The Python/C API

Version 3.7.4

Guido van Rossum
and the Python development team

septembre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

C'est la documentation de l'API utilisée par les développeurs C et C++ écrivant des modules d'extension ou intégrant Python. Elle va de pair avec extending-index, qui décrit les principes généraux de l'écriture d'extensions, mais ne rentre pas dans les détails de chaque fonction de l'API.

CHAPITRE 1

Introduction

The Application Programmer's Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write *extension modules* for specific purposes ; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application ; this technique is generally referred to as *embedding* Python in an application.

Writing an extension module is a relatively well-understood process, where a "cookbook" approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you're embedding or extending Python ; moreover, most applications that embed Python will need to provide a custom extension as well, so it's probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.

1.1 Coding standards

If you're writing C code for inclusion in CPython, you **must** follow the guidelines and standards defined in **PEP 7**. These guidelines apply regardless of the version of Python you are contributing to. Following these conventions is not necessary for your own third party extension modules, unless you eventually expect to contribute them to Python.

1.2 Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line :

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

This implies inclusion of the following standard headers : `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` and `<stdlib.h>` (if available).

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.

It is recommended to always define `PY_SSIZE_T_CLEAN` before including `Python.h`. See *Analyse des arguments et construction des valeurs* for a description of this macro.

All user visible names defined by Python.h (except those defined by the included standard headers) have one of the prefixes `Py` or `_Py`. Names beginning with `_Py` are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

Important : user code should never define names that begin with `Py` or `_Py`. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's **configure** script and *version* is `'%d.%d' % sys.version_info[:2]`. On Windows, the headers are installed in `prefix/include`, where `prefix` is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler's search path for includes. Do *not* place the parent directories on the search path and then use `#include <pythonX.Y/Python.h>`; this will break on multi-platform builds since the platform independent headers under `prefix` include the platform specific headers from `exec_prefix`.

C++ users should note that though the API is defined entirely using C, the header files do properly declare the entry points to be `extern "C"`, so there is no need to do anything special to use the API from C++.

1.3 Useful macros

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (e.g. *`Py_RETURN_NONE`*). Others of a more general utility are defined here. This is not necessarily a complete listing.

`Py_UNREACHABLE``()`

Use this when you have a code path that you do not expect to be reached. For example, in the `default:` clause in a `switch` statement for which all possible values are covered in `case` statements. Use this in places where you might be tempted to put an `assert(0)` or `abort()` call.

Nouveau dans la version 3.7.

`Py_ABS``(x)`

Return the absolute value of `x`.

Nouveau dans la version 3.3.

`Py_MIN``(x, y)`

Return the minimum value between `x` and `y`.

Nouveau dans la version 3.3.

`Py_MAX``(x, y)`

Return the maximum value between `x` and `y`.

Nouveau dans la version 3.3.

`Py_STRINGIFY``(x)`

Convert `x` to a C string. E.g. `Py_STRINGIFY(123)` returns `"123"`.

Nouveau dans la version 3.4.

Py_MEMBER_SIZE (type, member)
Return the size of a structure (`type`) `member` in bytes.

Nouveau dans la version 3.6.

Py_CHARMASK (c)
Argument must be a character or an integer in the range [-128, 127] or [0, 255]. This macro returns `c` cast to an `unsigned char`.

Py_GETENV (s)
Like `getenv(s)`, but returns *NULL* if `-E` was passed on the command line (i.e. if `Py_IgnoreEnvironmentFlag` is set).

Py_UNUSED (arg)
Use this for unused arguments in a function definition to silence compiler warnings, e.g. `PyObject* func(PyObject *Py_UNUSED(ignored))`.

Nouveau dans la version 3.4.

1.4 Objects, Types and Reference Counts

Most Python/C API functions have one or more arguments as well as a return value of type `PyObject*`. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap : you never declare an automatic or static variable of type `PyObject`, only pointer variables of type `PyObject*` can be declared. The sole exception are the type objects ; since these must never be deallocated, they are typically static `PyTypeObject` objects.

All Python objects (even Python integers) have a *type* and a *reference count*. An object's type determines what kind of object it is (e.g., an integer, a list, or a user-defined function ; there are many more as explained in types). For each of the well-known types there is a macro to check whether an object is of that type ; for instance, `PyList_Check(a)` is true if (and only if) the object pointed to by *a* is a Python list.

1.4.1 Reference Counts

The reference count is important because today's computers have a finite (and often severely limited) memory size ; it counts how many different places there are that have a reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When an object's reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference count is decremented. Those other objects may be deallocated in turn, if this decrement makes their reference count become zero, and so on. (There's an obvious problem with objects that reference each other here ; for now, the solution is "don't do that.")

Reference counts are always manipulated explicitly. The normal way is to use the macro `Py_INCREF()` to increment an object's reference count by one, and `Py_DECREF()` to decrement it by one. The `Py_DECREF()` macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator to be called. The deallocator is a function pointer contained in the object's type structure. The type-specific deallocator takes care of decrementing the reference counts for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that's needed. There's no chance that the reference count can overflow ; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming `sizeof(Py_ssize_t) >= sizeof(void*)`). Thus, the reference count increment is a simple operation.

It is not necessary to increment an object's reference count for every local variable that contains a pointer to an object. In theory, the object's reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn't changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to increment the reference count temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python ; the call mechanism guarantees to hold a reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without incrementing its reference count. Some other operation might conceivably remove the object from the list, decrementing its reference count and possible deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this ; there is a code path which allows control to flow back to the user from a *Py_DECREF()*, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with `PyObject_`, `PyNumber_`, `PySequence_` or `PyMapping_`). These operations always increment the reference count of the object they return. This leaves the caller with the responsibility to call *Py_DECREF()* when they are done with the result ; this soon becomes second nature.

Reference Count Details

The reference count behavior of functions in the Python/C API is best explained in terms of *ownership of references*. Ownership pertains to references, never to objects (objects are not owned : they are always shared). "Owning a reference" means being responsible for calling Py_DECREF on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually decref'ing it by calling *Py_DECREF()* or *Py_XDECREF()* when it's no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a *new* reference. When no ownership is transferred, the caller is said to *borrow* the reference. Nothing needs to be done for a borrowed reference.

Conversely, when a calling function passes in a reference to an object, there are two possibilities : the function *steals* a reference to the object, or it does not. *Stealing a reference* means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references ; the two notable exceptions are *PyList_SetItem()* and *PyTuple_SetItem()*, which steal a reference to the item (but not to the tuple or list into which the item is put !). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects ; for example, the code to create the tuple `(1, 2, "three")` could look like this (forgetting about error handling for the moment ; a better way to code this is shown below) :

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Here, *PyLong_FromLong()* returns a new reference which is immediately stolen by *PyTuple_SetItem()*. When you want to keep using an object although the reference to it will be stolen, use *Py_INCREF()* to grab another reference before calling the reference-stealing function.

Incidentally, *PyTuple_SetItem()* is the *only* way to set tuple items ; *PySequence_SetItem()* and *PyObject_SetItem()* refuse to do this since tuples are an immutable data type. You should only use *PyTuple_SetItem()* for tuples that you are creating yourself.

Equivalent code for populating a list can be written using *PyList_New()* and *PyList_SetItem()*.

However, in practice, you will rarely use these ways of creating and populating a tuple or list. There's a generic function, *Py_BuildValue()*, that can create most common objects from C values, directed by a *format string*. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking) :

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

It is much more common to use *PyObject_SetItem()* and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding reference counts is much saner, since you don't have to increment a reference count so you can give a reference away ("have it be stolen"). For example, this function sets all items of a list (actually, any mutable sequence) to a given item :

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple : in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like *PyObject_GetItem()* and *PySequence_GetItem()*, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only — *the plumage* (the type of the object passed as an argument to the function) *doesn't enter into it!* Thus, if you extract an item from a list using *PyList_GetItem()*, you don't own the reference — but if you obtain the same item from the same list using *PySequence_GetItem()* (which happens to take exactly the same arguments), you do own a reference to the returned object.

Here is an example of how you could write a function that computes the sum of the items in a list of integers ; once using *PyList_GetItem()*, and once using *PySequence_GetItem()*.

```
long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyLong_Check(item)) continue; /* Skip non-integers */
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyLong_Check(item)) {
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* Integer too big to fit in a C long, bail out */
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* Discard reference ownership */
        }
    }
    return total;
}
```

1.4.2 Types

There are few other data types that play a significant role in the Python/C API ; most are simple C types such as `int`, `long`, `double` and `char*`. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.

1.5 Exceptions

The Python programmer only needs to deal with exceptions if specific error handling is required ; unhandled exceptions are automatically propagated to the caller, then to the caller's caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function's documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator. If not documented otherwise, this indicator is either *NULL* or `-1`, depending on the function's return type. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with `PyErr_Occurred()`. These exceptions are always explicitly documented.

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states : an exception has occurred, or not. The function `PyErr_Occurred()` can be used to check for this : it returns a borrowed reference to the exception type object when an exception has occurred, and *NULL* otherwise. There are a number of functions to set the exception state : `PyErr_SetString()` is the most common (though not the most general) function to set the exception state, and `PyErr_Clear()` clears the exception state.

The full exception state consists of three objects (all of which can be *NULL*) : the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python result of `sys.exc_info()` ; however, they are not the same : the Python objects represent the last exception being handled by a Python `try` ... `except` statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter's main loop, which takes care of transferring it to `sys.exc_info()` and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function `sys.exc_info()`, which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread's exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled ; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should *not* set another exception — that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that this example doesn't need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code :

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Here is the corresponding C code, in all its glory :

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

 error:
    /* Cleanup code, shared by success and failure path */

    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}
```

This example represents an endorsed use of the `goto` statement in C ! It illustrates the use of `PyErr_ExceptionMatches()` and `PyErr_Clear()` to handle specific exceptions, and the use of `Py_XDECREF()` to dispose of owned references that may be *NULL* (note the `'X'` in the name ; `Py_DECREF()` would crash when confronted with a *NULL* reference). It is important that the variables used to hold owned references are initialized to *NULL* for this to work ; likewise, the proposed return value is initialized to `-1` (failure) and only set to success after the final call made is successful.

1.6 Embarquer Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules `builtins`, `__main__`, and `sys`. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the "script argument list" (`sys.argv`). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to `PySys_SetArgvEx(argc, argv, updatepath)` after the call to `Py_Initialize()`.

On most systems (in particular, on Unix and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/pythonX.Y` relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable `PATH`).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in `/usr/local/lib/pythonX.Y`. (In fact, this particular path is also the "fallback" location, used when no executable file named `python` is found along `PATH`.) The user can override this behavior by setting the environment variable `PYTHONHOME`, or insert additional directories in front of the standard path by setting `PYTHONPATH`.

The embedding application can steer the search by calling `Py_SetProgramName(file)` *before* calling `Py_Initialize()`. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, and `Py_GetProgramFullPath()` (all defined in `Modules/getpath.c`).

Sometimes, it is desirable to "uninitialize" Python. For instance, the application may want to start over (make another call to `Py_Initialize()`) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling `Py_FinalizeEx()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that `Py_FinalizeEx()` does *not* free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.

1.7 Debugging Builds

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

A full list of the various types of debugging builds is in the file `Misc/SpecialBuilds.txt` in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently-used builds will be described in the remainder of this section.

Compiling the interpreter with the `Py_DEBUG` macro defined produces what is generally meant by "a debug build" of Python. `Py_DEBUG` is enabled in the Unix build by adding `--with-pydebug` to the `./configure` command. It is also implied by the presence of the not-Python-specific `_DEBUG` macro. When `Py_DEBUG` is enabled in the Unix build, compiler optimization is disabled.

In addition to the reference count debugging described below, the following extra checks are performed :

- Extra checks are added to the object allocator.
- Extra checks are added to the parser and compiler.
- Downcasts from wide types to narrow types are checked for loss of information.
- A number of assertions are added to the dictionary and set implementations. In addition, the set object acquires a `test_c_api()` method.
- Sanity checks of the input arguments are added to frame creation.
- The storage for ints is initialized with a known invalid pattern to catch reference to uninitialized digits.
- Low-level tracing and extra exception checking are added to the runtime virtual machine.
- Extra checks are added to the memory arena implementation.
- Extra debugging is added to the thread module.

There may be additional checks not mentioned here.

Defining `Py_TRACE_REFS` enables reference tracing. When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every `PyObject`. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.) Implied by `Py_DEBUG`.

Please refer to `Misc/SpecialBuilds.txt` in the Python source distribution for more detailed information.

CHAPITRE 2

ABI Stable

L'API C de Python change à chaque version. La majorité de ces changement n'affecte cependant pas la compatibilité du code source. Typiquement, des API sont ajoutées, mais ni modifiées ni supprimées (bien que certaines interfaces puissent être supprimées, après avoir d'abord été dépréciées).

Malheureusement, la compatibilité de l'API ne s'étend pas à une compatibilité binaire (l'ABI). L'évolution des structures en est la raison principale : l'ajout de nouveaux attributs, ou le changement du type d'un attribut peut ne pas casser l'API mais casser l'ABI. Par conséquent, les modules d'extension doivent être recompilés à chaque nouvelle version de Python (ce n'est exceptionnellement pas nécessaire sur Unix, si aucune des interfaces modifiées n'est utilisée). De plus, sous Windows, les modules d'extension sont liés à un *pythonXY.dll* spécifique, ils est donc nécessaire de les recompiler pour les lier au nouveau DLL.

Depuis Python 3.2 il est garanti qu'une certaine partie de l'API gardera une ABI stable. Les modules d'extension souhaitant utiliser cette API (Appellée "API limitée") doivent définir `Py_LIMITED_API`. Des spécificités de l'interpréteur sont alors cachées au module, en contrepartie le module devient compatible avec toutes les versions de Python 3.x (x>=2) sans recompilation.

Dans certains cas, il est nécessaire d'étendre l'ABI stable avec de nouvelles fonctions. Les modules d'extension souhaitant utiliser ces nouvelles APIs doivent configurer `Py_LIMITED_API` à la valeur `Py_VERSION_HEX` correspondant à la plus ancienne version de Python qu'ils souhaitent supporter (voir *Version des API et ABI*, par exemple `0x03030000` pour Python 3.3). De tels modules fonctionneront dans toutes les versions ultérieures de Python, mais ne pourront pas se charger (dû à des symboles manquants) sur les versions plus anciennes.

Depuis Python 3.2, l'ensemble des fonctions exposées par l'API limitée est documentée dans la **PEP 384**. Dans la documentation de l'API C, les éléments ne faisant pas partie de l'API limitée sont notés "Ne faisant pas partie de l'API limitée" (*"Not part of the limited API"*).

CHAPITRE 3

The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, and `Py_single_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take `FILE*` parameters. One particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that `FILE*` parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

int **`Py_Main`** (int *argc*, wchar_t ***argv*)

The main program for the standard interpreter. This is made available for programs which embed Python. The *argc* and *argv* parameters should be prepared exactly as those which are passed to a C program's `main()` function (converted to wchar_t according to the user's locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the parameter list does not represent a valid Python command line.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `1`, but exit the process, as long as `Py_InspectFlag` is not set.

int **`PyRun_AnyFile`** (FILE **fp*, const char **filename*)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to *NULL*.

int **`PyRun_AnyFileFlags`** (FILE **fp*, const char **filename*, PyCompilerFlags **flags*)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *closeit* argument set to `0`.

int **`PyRun_AnyFileEx`** (FILE **fp*, const char **filename*, int *closeit*)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *flags* argument set to *NULL*.

int **`PyRun_AnyFileExFlags`** (FILE **fp*, const char **filename*, int *closeit*, PyCompilerFlags **flags*)

If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`.

filename is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *filename* is *NULL*, this function uses `"???"` as the filename.

int **PyRun_SimpleString** (const char **command*)
This is a simplified interface to *PyRun_SimpleStringFlags()* below, leaving the *PyCompilerFlags** argument set to NULL.

int **PyRun_SimpleStringFlags** (const char **command*, *PyCompilerFlags *flags*)
Executes the Python source code from *command* in the `__main__` module according to the *flags* argument. If `__main__` does not already exist, it is created. Returns `0` on success or `-1` if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of *flags*, see below.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `-1`, but exit the process, as long as `Py_InspectFlag` is not set.

int **PyRun_SimpleFile** (FILE **fp*, const char **filename*)
This is a simplified interface to *PyRun_SimpleFileExFlags()* below, leaving *closeit* set to `0` and *flags* set to *NULL*.

int **PyRun_SimpleFileEx** (FILE **fp*, const char **filename*, int *closeit*)
This is a simplified interface to *PyRun_SimpleFileExFlags()* below, leaving *flags* set to *NULL*.

int **PyRun_SimpleFileExFlags** (FILE **fp*, const char **filename*, int *closeit*, *PyCompilerFlags *flags*)
Similar to *PyRun_SimpleStringFlags()*, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *closeit* is true, the file is closed before PyRun_SimpleFileExFlags returns.

Note : On Windows, *fp* should be opened as binary mode (e.g. `fopen(filename, "rb")`. Otherwise, Python may not handle script file with LF line ending correctly.

---

int **PyRun_InteractiveOne** (FILE **fp*, const char **filename*)
This is a simplified interface to *PyRun_InteractiveOneFlags()* below, leaving *flags* set to *NULL*.

int **PyRun_InteractiveOneFlags** (FILE **fp*, const char **filename*, *PyCompilerFlags *flags*)
Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`).

Returns `0` when the input was executed successfully, `-1` if there was an exception, or an error code from the `errcode.h` include file distributed as part of Python if there was a parse error. (Note that `errcode.h` is not included by `Python.h`, so must be included specifically if needed.)

int **PyRun_InteractiveLoop** (FILE **fp*, const char **filename*)
This is a simplified interface to *PyRun_InteractiveLoopFlags()* below, leaving *flags* set to *NULL*.

int **PyRun_InteractiveLoopFlags** (FILE **fp*, const char **filename*, *PyCompilerFlags *flags*)
Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). Returns `0` at EOF or a negative number upon failure.

int **(*PyOS_InputHook)** (void)
Can be set to point to a function with the prototype `int func(void)`. The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the `Modules/_tkinter.c` in the Python source code.

char* **(*PyOS_ReadlineFunctionPointer)** (FILE *, FILE *, const char *)
Can be set to point to a function with the prototype `char *func(FILE *stdin, FILE *stdout, char *prompt)`, overriding the default function used to read a single line of input at the interpreter's prompt. The function is expected to output the string *prompt* if it's not *NULL*, and then read a line of input from the provided

standard input file, returning the resulting string. For example, The `readline` module sets this hook to provide line-editing and tab-completion features.

The result must be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or *NULL* if an error occurred.

Modifié dans la version 3.4 : The result must be allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, instead of being allocated by `PyMem_Malloc()` or `PyMem_Realloc()`.

struct _node* **PyParser_SimpleParseString** (const char **str*, int *start*)
This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving *filename* set to *NULL* and *flags* set to `0`.

struct _node* **PyParser_SimpleParseStringFlags** (const char **str*, int *start*, int *flags*)
This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving *filename* set to *NULL*.

struct _node* **PyParser_SimpleParseStringFlagsFilename** (const char **str*, const char **filename*, int *start*, int *flags*)
Parse Python source code from *str* using the start token *start* according to the *flags* argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`).

struct _node* **PyParser_SimpleParseFile** (FILE **fp*, const char **filename*, int *start*)
This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving *flags* set to `0`.

struct _node* **PyParser_SimpleParseFileFlags** (FILE **fp*, const char **filename*, int *start*, int *flags*)
Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from *fp* instead of an in-memory string.

*PyObject** **PyRun_String** (const char **str*, int *start*, *PyObject* **globals*, *PyObject* **locals*)
Return value : New reference. This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to *NULL*.

*PyObject** **PyRun_StringFlags** (const char **str*, int *start*, *PyObject* **globals*, *PyObject* **locals*, *PyCompilerFlags* **flags*)
Return value : New reference. Execute Python source code from *str* in the context specified by the objects *globals* and *locals* with the compiler flags specified by *flags*. *globals* must be a dictionary ; *locals* can be any object that implements the mapping protocol. The parameter *start* specifies the start token that should be used to parse the source code.

Returns the result of executing the code as a Python object, or *NULL* if an exception was raised.

*PyObject** **PyRun_File** (FILE **fp*, const char **filename*, int *start*, *PyObject* **globals*, *PyObject* **locals*)
Return value : New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0` and *flags* set to *NULL*.

*PyObject** **PyRun_FileEx** (FILE **fp*, const char **filename*, int *start*, *PyObject* **globals*, *PyObject* **locals*, int *closeit*)
Return value : New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *flags* set to *NULL*.

*PyObject** **PyRun_FileFlags** (FILE **fp*, const char **filename*, int *start*, *PyObject* **globals*, *PyObject* **locals*, *PyCompilerFlags* **flags*)
Return value : New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0`.

*PyObject** **PyRun_FileExFlags** (FILE **fp*, const char **filename*, int *start*, *PyObject* **globals*, *PyObject* **locals*, int *closeit*, *PyCompilerFlags* **flags*)
Return value : New reference. Similar to `PyRun_StringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *closeit* is true, the file is closed before `PyRun_FileExFlags()` returns.

*PyObject** **Py_CompileString** (const char **str*, const char **filename*, int *start*)

Return value : New reference. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to *NULL*.

*PyObject** **Py_CompileStringFlags** (const char **str*, const char **filename*, int *start*, *PyCompilerFlags* **flags*)

Return value : New reference. This is a simplified interface to `Py_CompileStringExFlags()` below, with *optimize* set to `-1`.

*PyObject** **Py_CompileStringObject** (const char **str*, *PyObject* **filename*, int *start*, *PyCompilerFlags* **flags*, int *optimize*)

Return value : New reference. Parse and compile the Python source code in *str*, returning the resulting code object. The start token is given by *start* ; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns *NULL* if the code cannot be parsed or compiled.

The integer *optimize* specifies the optimization level of the compiler ; a value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization ; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

Nouveau dans la version 3.4.

*PyObject** **Py_CompileStringExFlags** (const char **str*, const char **filename*, int *start*, *PyCompilerFlags* **flags*, int *optimize*)

Return value : New reference. Like `Py_CompileStringObject()`, but *filename* is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

Nouveau dans la version 3.2.

*PyObject** **PyEval_EvalCode** (*PyObject* **co*, *PyObject* **globals*, *PyObject* **locals*)

Return value : New reference. This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to *NULL*.

*PyObject** **PyEval_EvalCodeEx** (*PyObject* **co*, *PyObject* **globals*, *PyObject* **locals*, *PyObject* *const **args*, int *argcount*, *PyObject* *const **kws*, int *kwcount*, *PyObject* *const **defs*, int *defcount*, *PyObject* **kwdefs*, *PyObject* **closure*)

Return value : New reference. Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for *keyword-only* arguments and a closure tuple of cells.

PyFrameObject

The C structure of the objects used to describe frame objects. The fields of this type are subject to change at any time.

*PyObject** **PyEval_EvalFrame** (*PyFrameObject* **f*)

Return value : New reference. Evaluate an execution frame. This is a simplified interface to `PyEval_EvalFrameEx()`, for backward compatibility.

*PyObject** **PyEval_EvalFrameEx** (*PyFrameObject* **f*, int *throwflag*)

Return value : New reference. This is the main, unvarnished function of Python interpretation. It is literally 2000 lines long. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown ; this is used for the `throw()` methods of generator objects.

Modifié dans la version 3.4 : Cette fonction inclut maintenant une assertion de débogage afin d'assurer qu'elle ne passe pas sous silence une exception active.

int **PyEval_MergeCompilerFlags** (*PyCompilerFlags* **cf*)

This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

int **Py_eval_input**

The start symbol from the Python grammar for isolated expressions; for use with *Py_CompileString()*.

int **Py_file_input**

The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with *Py_CompileString()*. This is the symbol to use when compiling arbitrarily long Python source code.

int **Py_single_input**

The start symbol from the Python grammar for a single statement; for use with *Py_CompileString()*. This is the symbol used for the interactive interpreter loop.

struct **PyCompilerFlags**

This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, `from __future__ import` can modify *flags*.

Whenever `PyCompilerFlags *flags` is *NULL*, `cf_flags` is treated as equal to `0`, and any modification due to `from __future__ import` is discarded.

```
struct PyCompilerFlags {
    int cf_flags;
}
```

int **CO_FUTURE_DIVISION**

This bit can be set in *flags* to cause division operator `/` to be interpreted as "true division" according to **PEP 238**.

CHAPITRE 4

Reference Counting

The macros in this section are used for managing reference counts of Python objects.

void **Py_INCREF**(*PyObject *o*)

Increment the reference count for object *o*. The object must not be *NULL* ; if you aren't sure that it isn't *NULL*, use `Py_XINCREF()`.

void **Py_XINCREF**(*PyObject *o*)

Increment the reference count for object *o*. The object may be *NULL*, in which case the macro has no effect.

void **Py_DECREF**(*PyObject *o*)

Decrement the reference count for object *o*. The object must not be *NULL* ; if you aren't sure that it isn't *NULL*, use `Py_XDECREF()`. If the reference count reaches zero, the object's type's deallocation function (which must not be *NULL*) is invoked.

> **Avertissement :** The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a `__del__()` method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before `Py_DECREF()` is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call `Py_DECREF()` for the temporary variable.

void **Py_XDECREF**(*PyObject *o*)

Decrement the reference count for object *o*. The object may be *NULL*, in which case the macro has no effect ; otherwise the effect is the same as for `Py_DECREF()`, and the same warning applies.

void **Py_CLEAR**(*PyObject *o*)

Decrement the reference count for object *o*. The object may be *NULL*, in which case the macro has no effect ; otherwise the effect is the same as for `Py_DECREF()`, except that the argument is also set to *NULL*. The warning for `Py_DECREF()` does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to *NULL* before decrementing its reference count.

It is a good idea to use this macro whenever decrementing the value of a variable that might be traversed during garbage collection.

The following functions are for runtime dynamic embedding of Python : `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`. They are simply exported function versions of *Py_XINCREF()* and *Py_XDECREF()*, respectively.

The following functions or macros are only for use within the interpreter core : `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`, as well as the global variable `_Py_RefTotal`.

CHAPITRE 5

Gestion des exceptions

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the POSIX `errno` variable : there is a global indicator (per thread) of the last error that occurred. Most C API functions don't clear this on success, but will set it to indicate the cause of the error on failure. Most C API functions also return an error indicator, usually *NULL* if they are supposed to return a pointer, or `-1` if they return an integer (exception : the `PyArg_*()` functions return `1` for success and `0` for failure).

Concretely, the error indicator consists of three object pointers : the exception's type, the exception's value, and the traceback object. Any of those pointers can be NULL if non-set (although some combinations are forbidden, for example you can't have a non-NULL traceback if the exception type is NULL).

When a function must fail because some function it called failed, it generally doesn't set the error indicator ; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations) ; it should *not* continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

Note : The error indicator is **not** the result of `sys.exc_info()`. The former corresponds to an exception that is not yet caught (and is therefore still propagating), while the latter returns an exception after it is caught (and has therefore stopped propagating).

5.1 Printing and clearing

void **`PyErr_Clear`**()

Clear the error indicator. If the error indicator is not set, there is no effect.

void **`PyErr_PrintEx`**(int *set_sys_last_vars*)

Print a standard traceback to `sys.stderr` and clear the error indicator. **Unless** the error is a `SystemExit`. In that case the no traceback is printed and Python process will exit with the error code specified by the `SystemExit` instance.

Call this function **only** when the error indicator is set. Otherwise it will cause a fatal error !

If *set_sys_last_vars* is nonzero, the variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` will be set to the type, value and traceback of the printed exception, respectively.

void **PyErr_Print**()
Alias for `PyErr_PrintEx(1)`.

void **PyErr_WriteUnraisable**(*PyObject *obj*)
This utility function prints a warning message to `sys.stderr` when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an `__del__()` method.

The function is called with a single argument *obj* that identifies the context in which the unraisable exception occurred. If possible, the repr of *obj* will be printed in the warning message.

An exception must be set when calling this function.

5.2 Lever des exceptions

These functions help you set the current thread's error indicator. For convenience, some of these functions will always return a NULL pointer for use in a `return` statement.

void **PyErr_SetString**(*PyObject *type*, const char **message*)
This is the most common way to set the error indicator. The first argument specifies the exception type ; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not increment its reference count. The second argument is an error message ; it is decoded from `'utf-8`'.

void **PyErr_SetObject**(*PyObject *type*, *PyObject *value*)
This function is similar to `PyErr_SetString()` but lets you specify an arbitrary Python object for the "value" of the exception.

*PyObject** **PyErr_Format**(*PyObject *exception*, const char **format*, ...)
Return value : Always NULL. This function sets the error indicator and returns *NULL*. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message ; they have the same meaning and values as in `PyUnicode_FromFormat()`. *format* is an ASCII-encoded string.

*PyObject** **PyErr_FormatV**(*PyObject *exception*, const char **format*, va_list *vargs*)
Return value : Always NULL. Same as `PyErr_Format()`, but taking a `va_list` argument rather than a variable number of arguments.

Nouveau dans la version 3.5.

void **PyErr_SetNone**(*PyObject *type*)
This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()
This is a shorthand for `PyErr_SetString(PyExc_TypeError, message)`, where *message* indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.

*PyObject** **PyErr_NoMemory**()
Return value : Always NULL. This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)` ; it returns *NULL* so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

*PyObject** **PyErr_SetFromErrno**(*PyObject *type*)
Return value : Always NULL. This is a convenience function to raise an exception when a C library function has returned an error and set the C variable `errno`. It constructs a tuple object whose first item is the integer `errno` value and whose second item is the corresponding error message (gotten from `strerror()`), and

then calls `PyErr_SetObject(type, object)`. On Unix, when the `errno` value is `EINTR`, indicating an interrupted system call, this calls `PyErr_CheckSignals()`, and if that set the error indicator, leaves it set to that. The function always returns *NULL*, so a wrapper function around a system call can write `return PyErr_SetFromErrno(type);` when the system call returns an error.

*PyObject** **`PyErr_SetFromErrnoWithFilenameObject`**(*PyObject *type*, *PyObject *filenameObject*)

Return value : Always NULL. Similar to `PyErr_SetFromErrno()`, with the additional behavior that if *filenameObject* is not *NULL*, it is passed to the constructor of *type* as a third parameter. In the case of `OSError` exception, this is used to define the `filename` attribute of the exception instance.

*PyObject** **`PyErr_SetFromErrnoWithFilenameObjects`**(*PyObject *type*, *PyObject *filenameObject*, *PyObject *filenameObject2*)

Return value : Always NULL. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but takes a second filename object, for raising errors when a function that takes two filenames fails.

Nouveau dans la version 3.4.

*PyObject** **`PyErr_SetFromErrnoWithFilename`**(*PyObject *type*, const char **filename*)

Return value : Always NULL. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

*PyObject** **`PyErr_SetFromWindowsErr`**(int *ierr*)

Return value : Always NULL. This is a convenience function to raise `WindowsError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_WindowsError, object)`. This function always returns *NULL*.

Disponibilité : Windows.

*PyObject** **`PyErr_SetExcFromWindowsErr`**(*PyObject *type*, int *ierr*)

Return value : Always NULL. Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised.

Disponibilité : Windows.

*PyObject** **`PyErr_SetFromWindowsErrWithFilename`**(int *ierr*, const char **filename*)

Return value : Always NULL. Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

Disponibilité : Windows.

*PyObject** **`PyErr_SetExcFromWindowsErrWithFilenameObject`**(*PyObject *type*, int *ierr*, *PyObject *filename*)

Return value : Always NULL. Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, with an additional parameter specifying the exception type to be raised.

Disponibilité : Windows.

*PyObject** **`PyErr_SetExcFromWindowsErrWithFilenameObjects`**(*PyObject *type*, int *ierr*, *PyObject *filename*, *PyObject *filename2*)

Return value : Always NULL. Similar to `PyErr_SetExcFromWindowsErrWithFilenameObject()`, but accepts a second filename object.

Disponibilité : Windows.

Nouveau dans la version 3.4.

*PyObject** **`PyErr_SetExcFromWindowsErrWithFilename`**(*PyObject *type*, int *ierr*, const char **filename*)

Return value : Always NULL. Similar to `PyErr_SetFromWindowsErrWithFilename()`, with an additional parameter specifying the exception type to be raised.

Disponibilité : Windows.

*PyObject** **PyErr_SetImportError** (*PyObject *msg*, *PyObject *name*, *PyObject *path*)
Return value : Always NULL. This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

Nouveau dans la version 3.3.

void **PyErr_SyntaxLocationObject** (*PyObject *filename*, int *lineno*, int *col_offset*)
Set file, line, and offset information for the current exception. If the current exception is not a `SyntaxError`, then it sets additional attributes, which make the exception printing subsystem think the exception is a `SyntaxError`.

Nouveau dans la version 3.4.

void **PyErr_SyntaxLocationEx** (const char **filename*, int *lineno*, int *col_offset*)
Like `PyErr_SyntaxLocationObject()`, but *filename* is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

Nouveau dans la version 3.2.

void **PyErr_SyntaxLocation** (const char **filename*, int *lineno*)
Like `PyErr_SyntaxLocationEx()`, but the col_offset parameter is omitted.

void **PyErr_BadInternalCall** ()
This is a shorthand for `PyErr_SetString(PyExc_SystemError, message)`, where *message* indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

5.3 Issuing warnings

Use these functions to issue warnings from C code. They mirror similar functions exported by the Python `warnings` module. They normally print a warning message to *sys.stderr* ; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case they will raise an exception. It is also possible that the functions raise an exception because of a problem with the warning machinery. The return value is `0` if no exception is raised, or `-1` if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception ; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_DECREF()` owned references and return an error value).

int **PyErr_WarnEx** (*PyObject *category*, const char **message*, Py_ssize_t *stack_level*)
Issue a warning message. The *category* argument is a warning category (see below) or *NULL* ; the *message* argument is a UTF-8 encoded string. *stack_level* is a positive number giving a number of stack frames ; the warning will be issued from the currently executing line of code in that stack frame. A *stack_level* of 1 is the function calling `PyErr_WarnEx()`, 2 is the function above that, and so forth.

Warning categories must be subclasses of `PyExc_Warning`; `PyExc_Warning` is a subclass of `PyExc_Exception` ; the default warning category is `PyExc_RuntimeWarning`. The standard Python warning categories are available as global variables whose names are enumerated at *Standard Warning Categories*.

For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

*PyObject** **PyErr_SetImportErrorSubclass** (*PyObject *exception*, *PyObject *msg*, *PyObject *name*, *PyObject *path*)
Return value : Always NULL. Much like `PyErr_SetImportError()` but this function allows for specifying a subclass of `ImportError` to raise.

Nouveau dans la version 3.6.

int **PyErr_WarnExplicitObject** (*PyObject *category*, *PyObject *message*, *PyObject *filename*, int *lineno*, *PyObject *module*, *PyObject *registry*)
Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around

the Python function `warnings.warn_explicit()`, see there for more information. The *module* and *registry* arguments may be set to *NULL* to get the default effect described there.

Nouveau dans la version 3.4.

int **PyErr_WarnExplicit** (*PyObject* **category*, const char **message*, const char **filename*, int *lineno*, const char **module*, *PyObject* **registry*)

Similar to `PyErr_WarnExplicitObject()` except that *message* and *module* are UTF-8 encoded strings, and *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

int **PyErr_WarnFormat** (*PyObject* **category*, Py_ssize_t *stack_level*, const char **format*, ...)

Function similar to `PyErr_WarnEx()`, but use `PyUnicode_FromFormat()` to format the warning message. *format* is an ASCII-encoded string.

Nouveau dans la version 3.2.

int **PyErr_ResourceWarning** (*PyObject* **source*, Py_ssize_t *stack_level*, const char **format*, ...)

Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and pass *source* to `warnings.WarningMessage()`.

Nouveau dans la version 3.6.

5.4 Querying the error indicator

*PyObject** **PyErr_Occurred** ()

Return value : Borrowed reference. Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*()` functions or to `PyErr_Restore()`). If not set, return *NULL*. You do not own a reference to the return value, so you do not need to `Py_DECREF()` it.

Note : Do not compare the return value to a specific exception ; use `PyErr_ExceptionMatches()` instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

int **PyErr_ExceptionMatches** (*PyObject* **exc*)

Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. This should only be called when an exception is actually set ; a memory access violation will occur if no exception has been raised.

int **PyErr_GivenExceptionMatches** (*PyObject* **given*, *PyObject* **exc*)

Return true if the *given* exception matches the exception type in *exc*. If *exc* is a class object, this also returns true when *given* is an instance of a subclass. If *exc* is a tuple, all exception types in the tuple (and recursively in subtuples) are searched for a match.

void **PyErr_Fetch** (*PyObject* ***ptype*, *PyObject* ***pvalue*, *PyObject* ***ptraceback*)

Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to *NULL*. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be *NULL* even when the type object is not.

Note : This function is normally only used by code that needs to catch exceptions or by code that needs to save and restore the error indicator temporarily, e.g. :

```
{
   PyObject *type, *value, *traceback;
   PyErr_Fetch(&type, &value, &traceback);

   /* ... code that might produce other errors ... */
```

(suite sur la page suivante)

(suite de la page précédente)

```
    PyErr_Restore(type, value, traceback);
}
```

void **PyErr_Restore** (*PyObject *type*, *PyObject *value*, *PyObject *traceback*)

Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are *NULL*, the error indicator is cleared. Do not pass a *NULL* type and non-*NULL* value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object : you must own a reference to each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.)

Note : This function is normally only used by code that needs to save and restore the error indicator temporarily. Use `PyErr_Fetch()` to save the current error indicator.

void **PyErr_NormalizeException** (*PyObject***exc, *PyObject***val, *PyObject***tb)

Under certain circumstances, the values returned by `PyErr_Fetch()` below can be "unnormalized", meaning that `*exc` is a class object but `*val` is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

Note : This function *does not* implicitly set the `__traceback__` attribute on the exception value. If setting the traceback appropriately is desired, the following additional snippet is needed :

```
if (tb != NULL) {
  PyException_SetTraceback(val, tb);
}
```

void **PyErr_GetExcInfo** (*PyObject **ptype*, *PyObject **pvalue*, *PyObject **ptraceback*)

Retrieve the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be *NULL*. Does not modify the exception info state.

Note : This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_SetExcInfo()` to restore or clear the exception state.

Nouveau dans la version 3.3.

void **PyErr_SetExcInfo** (*PyObject *type*, *PyObject *value*, *PyObject *traceback*)

Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass *NULL* for all three arguments. For general rules about the three arguments, see `PyErr_Restore()`.

Note : This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_GetExcInfo()` to read the exception state.

Nouveau dans la version 3.3.

5.5 Traitement des signaux

int **PyErr_CheckSignals**()

This function interacts with Python's signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for `SIGINT` is to raise the `KeyboardInterrupt` exception. If an exception is raised the error indicator is set and the function returns `-1` ; otherwise the function returns `0`. The error indicator may or may not be cleared if it was previously set.

void **PyErr_SetInterrupt**()

Simulate the effect of a `SIGINT` signal arriving. The next time *PyErr_CheckSignals()* is called, the Python signal handler for `SIGINT` will be called.

If `SIGINT` isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), this function does nothing.

int **PySignal_SetWakeupFd**(int *fd*)

This utility function specifies a file descriptor to which the signal number is written as a single byte whenever a signal is received. *fd* must be non-blocking. It returns the previous such file descriptor.

The value `-1` disables the feature ; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be called from the main thread.

Modifié dans la version 3.5 : On Windows, the function now also supports socket handles.

5.6 Exception Classes

*PyObject** **PyErr_NewException**(const char **name*, *PyObject* **base*, *PyObject* **dict*)

Return value : New reference. This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally *NULL*. This creates a class object derived from `Exception` (accessible in C as `PyExc_Exception`).

The `__module__` attribute of the new class is set to the first part (up to the last dot) of the *name* argument, and the class name is set to the last part (after the last dot). The *base* argument can be used to specify alternate base classes ; it can either be only one class or a tuple of classes. The *dict* argument can be used to specify a dictionary of class variables and methods.

*PyObject** **PyErr_NewExceptionWithDoc**(const char **name*, const char **doc*, *PyObject* **base*, *PyObject* **dict*)

Return value : New reference. Same as *PyErr_NewException()*, except that the new exception class can easily be given a docstring : If *doc* is non-*NULL*, it will be used as the docstring for the exception class.

Nouveau dans la version 3.2.

5.7 Objets exception

*PyObject** **PyException_GetTraceback**(*PyObject* **ex*)

Return value : New reference. Return the traceback associated with the exception as a new reference, as accessible from Python through `__traceback__`. If there is no traceback associated, this returns *NULL*.

int **PyException_SetTraceback**(*PyObject* **ex*, *PyObject* **tb*)

Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

*PyObject** **PyException_GetContext**(*PyObject* **ex*)

Return value : New reference. Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through `__context__`. If there is no context associated, this returns *NULL*.

void **PyException_SetContext** (*PyObject *ex*, *PyObject *ctx*)
Set the context associated with the exception to *ctx*. Use *NULL* to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

*PyObject** **PyException_GetCause** (*PyObject *ex*)
Return value : New reference. Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through `__cause__`.

void **PyException_SetCause** (*PyObject *ex*, *PyObject *cause*)
Set the cause associated with the exception to *cause*. Use *NULL* to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.

`__suppress_context__` is implicitly set to `True` by this function.

5.8 Objets exception Unicode

The following functions are used to create and modify Unicode exceptions from C.

*PyObject** **PyUnicodeDecodeError_Create** (const char **encoding*, const char **object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char **reason*)
Return value : New reference. Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

*PyObject** **PyUnicodeEncodeError_Create** (const char **encoding*, const *Py_UNICODE *object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char **reason*)
Return value : New reference. Create a `UnicodeEncodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

*PyObject** **PyUnicodeTranslateError_Create** (const *Py_UNICODE *object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char **reason*)
Return value : New reference. Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*. *reason* is a UTF-8 encoded string.

*PyObject** **PyUnicodeDecodeError_GetEncoding** (*PyObject *exc*)
*PyObject** **PyUnicodeEncodeError_GetEncoding** (*PyObject *exc*)
Return value : New reference. Return the *encoding* attribute of the given exception object.

*PyObject** **PyUnicodeDecodeError_GetObject** (*PyObject *exc*)
*PyObject** **PyUnicodeEncodeError_GetObject** (*PyObject *exc*)
*PyObject** **PyUnicodeTranslateError_GetObject** (*PyObject *exc*)
Return value : New reference. Return the *object* attribute of the given exception object.

int **PyUnicodeDecodeError_GetStart** (*PyObject *exc*, Py_ssize_t **start*)
int **PyUnicodeEncodeError_GetStart** (*PyObject *exc*, Py_ssize_t **start*)
int **PyUnicodeTranslateError_GetStart** (*PyObject *exc*, Py_ssize_t **start*)
Get the *start* attribute of the given exception object and place it into **start*. *start* must not be *NULL*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetStart** (*PyObject *exc*, Py_ssize_t *start*)
int **PyUnicodeEncodeError_SetStart** (*PyObject *exc*, Py_ssize_t *start*)
int **PyUnicodeTranslateError_SetStart** (*PyObject *exc*, Py_ssize_t *start*)
Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_GetEnd** (*PyObject *exc*, Py_ssize_t **end*)
int **PyUnicodeEncodeError_GetEnd** (*PyObject *exc*, Py_ssize_t **end*)

int **PyUnicodeTranslateError_GetEnd** (*PyObject* *exc, Py_ssize_t *end)
Get the *end* attribute of the given exception object and place it into **end*. *end* must not be *NULL*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetEnd** (*PyObject* *exc, Py_ssize_t *end*)
int **PyUnicodeEncodeError_SetEnd** (*PyObject* *exc, Py_ssize_t *end*)
int **PyUnicodeTranslateError_SetEnd** (*PyObject* *exc, Py_ssize_t *end*)
Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

*PyObject** **PyUnicodeDecodeError_GetReason** (*PyObject* *exc)
*PyObject** **PyUnicodeEncodeError_GetReason** (*PyObject* *exc)
*PyObject** **PyUnicodeTranslateError_GetReason** (*PyObject* *exc)
Return value : New reference. Return the *reason* attribute of the given exception object.

int **PyUnicodeDecodeError_SetReason** (*PyObject* *exc, const char *reason)
int **PyUnicodeEncodeError_SetReason** (*PyObject* *exc, const char *reason)
int **PyUnicodeTranslateError_SetReason** (*PyObject* *exc, const char *reason)
Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

5.9 Contrôle de la récursion

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically).

int **Py_EnterRecursiveCall** (const char *where)
Marks a point where a recursive C-level call is about to be performed.

If `USE_STACKCHECK` is defined, this function checks if the OS stack overflowed using `PyOS_CheckStack()`. In this is the case, it sets a `MemoryError` and returns a nonzero value.

The function then checks if the recursion limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

where should be a string such as `" in instance check"` to be concatenated to the `RecursionError` message caused by the recursion depth limit.

void **Py_LeaveRecursiveCall** ()
Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

Properly implementing `tp_repr` for container types requires special recursion handling. In addition to protecting the stack, `tp_repr` also needs to track objects to prevent cycles. The following two functions facilitate this functionality. Effectively, these are the C equivalent to `reprlib.recursive_repr()`.

int **Py_ReprEnter** (*PyObject* *object)
Called at the beginning of the `tp_repr` implementation to detect cycles.

If the object has already been processed, the function returns a positive integer. In that case the `tp_repr` implementation should return a string object indicating a cycle. As examples, `dict` objects return `{...}` and `list` objects return `[...]`.

The function will return a negative integer if the recursion limit is reached. In that case the `tp_repr` implementation should typically return `NULL`.

Otherwise, the function returns zero and the `tp_repr` implementation can continue normally.

void **Py_ReprLeave** (*PyObject* *object)
Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.

5.10 Exceptions standards

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables :

Nom C	Nom Python	Notes
PyExc_BaseException	BaseException	(1)
PyExc_Exception	Exception	(1)
PyExc_ArithmeticError	ArithmeticError	(1)
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_BlockingIOError	BlockingIOError	
PyExc_BrokenPipeError	BrokenPipeError	
PyExc_BufferError	BufferError	
PyExc_ChildProcessError	ChildProcessError	
PyExc_ConnectionAbortedError	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedError	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FloatingPointError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	(1)
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	(1)
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_ReferenceError	RecursionError	
PyExc_ReferenceError	ReferenceError	(2)
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Nom C	Nom Python	Notes
PyExc_ImportError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

Nouveau dans la version 3.3 : `PyExc_BlockingIOError`, `PyExc_BrokenPipeError`, `PyExc_ChildProcessError`, `PyExc_ConnectionError`, `PyExc_ConnectionAbortedError`, `PyExc_ConnectionRefusedError`, `PyExc_ConnectionResetError`, `PyExc_FileExistsError`, `PyExc_FileNotFoundError`, `PyExc_InterruptedError`, `PyExc_IsADirectoryError`, `PyExc_NotADirectoryError`, `PyExc_PermissionError`, `PyExc_ProcessLookupError` and `PyExc_TimeoutError` were introduced following **PEP 3151**.

Nouveau dans la version 3.5 : `PyExc_StopAsyncIteration` et `PyExc_RecursionError`.

Nouveau dans la version 3.6 : `PyExc_ModuleNotFoundError`.

These are compatibility aliases to `PyExc_OSError` :

Nom C	Notes
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	(3)

Modifié dans la version 3.3 : These aliases used to be separate exception types.

Notes :

(1) C'est la classe de base pour les autres exceptions standards.

(2) Identique à `weakref.ReferenceError`.

(3) Only defined on Windows ; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

5.11 Standard Warning Categories

All standard Python warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type *`PyObject*`* ; they are all class objects. For completeness, here are all the variables :

Nom C	Nom Python	Notes
PyExc_Warning	Warning	(1)
PyExc_BytesWarning	BytesWarning	
PyExc_DeprecationWarning	DeprecationWarning	
PyExc_FutureWarning	FutureWarning	
PyExc_ImportWarning	ImportWarning	
PyExc_PendingDeprecationWarning	PendingDeprecationWarning	
PyExc_ResourceWarning	ResourceWarning	
PyExc_RuntimeWarning	RuntimeWarning	
PyExc_SyntaxWarning	SyntaxWarning	
PyExc_UnicodeWarning	UnicodeWarning	
PyExc_UserWarning	UserWarning	

Nouveau dans la version 3.2 : `PyExc_ResourceWarning`.

Notes :

(1) C'est la classe de base pour les autres catégories de *warning*.

CHAPITRE 6

Utilitaires

Les fonctions de ce chapitre sont utilitaires, certaines aident à rendre le code en C plus portable, d'autres à utiliser des modules Python depuis du C, analyser des arguments de fonctions, ou encore construire des valeurs Python à partir de valeurs C.

6.1 Operating System Utilities

*PyObject** **PyOS_FSPath**(*PyObject* **path*)

Return value : New reference. Return the file system representation for *path*. If the object is a `str` or `bytes` object, then its reference count is incremented. If the object implements the `os.PathLike` interface, then `__fspath__()` is returned as long as it is a `str` or `bytes` object. Otherwise `TypeError` is raised and `NULL` is returned.

Nouveau dans la version 3.6.

int **Py_FdIsInteractive**(FILE **fp*, const char **filename*)

Return true (nonzero) if the standard I/O file *fp* with name *filename* is deemed interactive. This is the case for files for which `isatty(fileno(fp))` is true. If the global flag `Py_InteractiveFlag` is true, this function also returns true if the *filename* pointer is *NULL* or if the name is equal to one of the strings `'<stdin>'` or `'???'`.

void **PyOS_BeforeFork**()

Function to prepare some internal state before a process fork. This should be called before calling `fork()` or any similar function that clones the current process. Only available on systems where `fork()` is defined.

Nouveau dans la version 3.7.

void **PyOS_AfterFork_Parent**()

Function to update some internal state after a process fork. This should be called from the parent process after calling `fork()` or any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where `fork()` is defined.

Nouveau dans la version 3.7.

void **PyOS_AfterFork_Child**()

Function to update internal interpreter state after a process fork. This must be called from the child process after calling `fork()`, or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where `fork()` is defined.

Nouveau dans la version 3.7.

Voir aussi :

`os.register_at_fork()` allows registering custom Python functions to be called by *PyOS_BeforeFork()*, *PyOS_AfterFork_Parent()* and *PyOS_AfterFork_Child()*.

void **PyOS_AfterFork**()

Function to update some internal state after a process fork ; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

Obsolète depuis la version 3.7 : This function is superseded by *PyOS_AfterFork_Child()*.

int **PyOS_CheckStack**()

Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when `USE_STACKCHECK` is defined (currently on Windows using the Microsoft Visual C++ compiler). `USE_STACKCHECK` will be defined automatically ; you should never change the definition in your own code.

PyOS_sighandler_t **PyOS_getsig**(int *i*)

Return the current signal handler for signal *i*. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly ! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

PyOS_sighandler_t **PyOS_setsig**(int *i*, PyOS_sighandler_t *h*)

Set the signal handler for signal *i* to be *h* ; return the old signal handler. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly ! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

wchar_t* **Py_DecodeLocale**(const char* *arg*, size_t **size*)

Decode a byte string from the locale encoding with the surrogateescape error handler : undecodable bytes are decoded as characters in range U+DC80..U+DCFF. If a byte sequence can be decoded as a surrogate character, escape the bytes using the surrogateescape error handler instead of decoding them.

Encoding, highest priority to lowest priority :

- `UTF-8` on macOS and Android ;
- `UTF-8` if the Python UTF-8 mode is enabled ;
- `ASCII` if the `LC_CTYPE` locale is `"C"`, `nl_langinfo(CODESET)` returns the `ASCII` encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the `ISO-8859-1` encoding.
- the current locale encoding.

Return a pointer to a newly allocated wide character string, use *PyMem_RawFree()* to free the memory. If size is not `NULL`, write the number of wide characters excluding the null character into `*size`

Return `NULL` on decoding error or memory allocation error. If *size* is not `NULL`, `*size` is set to `(size_t)-1` on memory error or set to `(size_t)-2` on decoding error.

Decoding errors should never happen, unless there is a bug in the C library.

Use the *Py_EncodeLocale()* function to encode the character string back to a byte string.

Voir aussi :

The *PyUnicode_DecodeFSDefaultAndSize()* and *PyUnicode_DecodeLocaleAndSize()* functions.

Nouveau dans la version 3.5.

Modifié dans la version 3.7 : The function now uses the UTF-8 encoding in the UTF-8 mode.

char* **Py_EncodeLocale**(const wchar_t **text*, size_t **error_pos*)

Encode a wide character string to the locale encoding with the surrogateescape error handler : surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority :

- `UTF-8` on macOS and Android ;
- `UTF-8` if the Python UTF-8 mode is enabled ;

— `ASCII` if the `LC_CTYPE` locale is `"C"`, `nl_langinfo(CODESET)` returns the `ASCII` encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the `ISO-8859-1` encoding.

— the current locale encoding.

The function uses the UTF-8 encoding in the Python UTF-8 mode.

Return a pointer to a newly allocated byte string, use *`PyMem_Free()`* to free the memory. Return `NULL` on encoding error or memory allocation error

If error_pos is not `NULL`, `*error_pos` is set to `(size_t)-1` on success, or set to the index of the invalid character on encoding error.

Use the *`Py_DecodeLocale()`* function to decode the bytes string back to a wide character string.

Modifié dans la version 3.7 : The function now uses the UTF-8 encoding in the UTF-8 mode.

Voir aussi :

The *`PyUnicode_EncodeFSDefault()`* and *`PyUnicode_EncodeLocale()`* functions.

Nouveau dans la version 3.5.

Modifié dans la version 3.7 : The function now supports the UTF-8 mode.

6.2 System Functions

These are utility functions that make functionality from the `sys` module accessible to C code. They all work with the current interpreter thread's `sys` module's dict, which is contained in the internal thread state structure.

PyObject ***`PySys_GetObject`**(const char **name*)

Return value : Borrowed reference. Return the object *name* from the `sys` module or *NULL* if it does not exist, without setting an exception.

int **`PySys_SetObject`**(const char **name*, *PyObject* **v*)

Set *name* in the `sys` module to *v* unless *v* is *NULL*, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

void **`PySys_ResetWarnOptions`**()

Reset `sys.warnoptions` to an empty list. This function may be called prior to *`Py_Initialize()`*.

void **`PySys_AddWarnOption`**(const wchar_t **s*)

Append *s* to `sys.warnoptions`. This function must be called prior to *`Py_Initialize()`* in order to affect the warnings filter list.

void **`PySys_AddWarnOptionUnicode`**(*PyObject* **unicode*)

Append *unicode* to `sys.warnoptions`.

Note : this function is not currently usable from outside the CPython implementation, as it must be called prior to the implicit import of `warnings` in *`Py_Initialize()`* to be effective, but can't be called until enough of the runtime has been initialized to permit the creation of Unicode objects.

void **`PySys_SetPath`**(const wchar_t **path*)

Set `sys.path` to a list object of paths found in *path* which should be a list of paths separated with the platform's search path delimiter (**`:`** on Unix, **`;`** on Windows).

void **`PySys_WriteStdout`**(const char **format*, ...)

Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted "%s" formats should occur ; these should be limited using "%.<N>s" where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for "%f", which can print hundreds of digits for very large numbers.

If a problem occurs, or `sys.stdout` is unset, the formatted message is written to the real (C level) *stdout*.

void **PySys_WriteStderr**(const char **format*, ...)

As *PySys_WriteStdout()*, but write to `sys.stderr` or *stderr* instead.

void **PySys_FormatStdout**(const char **format*, ...)

Function similar to PySys_WriteStdout() but format the message using *PyUnicode_FromFormatV()* and don't truncate the message to an arbitrary length.

Nouveau dans la version 3.2.

void **PySys_FormatStderr**(const char **format*, ...)

As *PySys_FormatStdout()*, but write to `sys.stderr` or *stderr* instead.

Nouveau dans la version 3.2.

void **PySys_AddXOption**(const wchar_t **s*)

Parse *s* as a set of `-X` options and add them to the current options mapping as returned by *PySys_GetXOptions()*. This function may be called prior to *Py_Initialize()*.

Nouveau dans la version 3.2.

PyObject ***PySys_GetXOptions**()

Return value : Borrowed reference. Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, *NULL* is returned and an exception is set.

Nouveau dans la version 3.2.

6.3 Process Control

void **Py_FatalError**(const char **message*)

Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter ; e.g., when the object administration appears to be corrupted. On Unix, the standard C library function `abort()` is called which will attempt to produce a `core` file.

void **Py_Exit**(int *status*)

Exit the current process. This calls *Py_FinalizeEx()* and then calls the standard C library function `exit(status)`. If *Py_FinalizeEx()* indicates an error, the exit status is set to 120.

Modifié dans la version 3.6 : Errors from finalization no longer ignored.

int **Py_AtExit**(void (**func*)())

Register a cleanup function to be called by *Py_FinalizeEx()*. The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, *Py_AtExit()* returns `0` ; on failure, it returns `-1`. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by *func*.

6.4 Importer des modules

*PyObject** **PyImport_ImportModule**(const char **name*)

Return value : New reference. This is a simplified interface to *PyImport_ImportModuleEx()* below, leaving the *globals* and *locals* arguments set to *NULL* and *level* set to 0. When the *name* argument contains a dot (when it specifies a submodule of a package), the *fromlist* argument is set to the list `['*']` so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when *name* in fact specifies a subpackage instead of a submodule : the submodules specified in the package's `__all__` variable are loaded.) Return a new reference to the imported module, or *NULL* with an exception set on failure. A failing import of a module doesn't leave the module in `sys.modules`.

This function always uses absolute imports.

*PyObject** **PyImport_ImportModuleNoBlock**(const char **name*)

Return value : New reference. This function is a deprecated alias of `PyImport_ImportModule()`.

Modifié dans la version 3.3 : This function used to fail immediately when the import lock was held by another thread. In Python 3.3 though, the locking scheme switched to per-module locks for most purposes, so this function's special behaviour isn't needed anymore.

*PyObject** **PyImport_ImportModuleEx**(const char **name*, *PyObject *globals*, *PyObject *locals*, *PyObject *fromlist*)

Return value : New reference. Import a module. This is best described by referring to the built-in Python function `__import__()`.

The return value is a new reference to the imported module or top-level package, or *NULL* with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Failing imports remove incomplete module objects, like with `PyImport_ImportModule()`.

*PyObject** **PyImport_ImportModuleLevelObject**(*PyObject *name*, *PyObject *globals*, *PyObject *locals*, *PyObject *fromlist*, int *level*)

Return value : New reference. Import a module. This is best described by referring to the built-in Python function `__import__()`, as the standard `__import__()` function calls this function directly.

The return value is a new reference to the imported module or top-level package, or *NULL* with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Nouveau dans la version 3.3.

*PyObject** **PyImport_ImportModuleLevel**(const char **name*, *PyObject *globals*, *PyObject *locals*, *PyObject *fromlist*, int *level*)

Return value : New reference. Similar to `PyImport_ImportModuleLevelObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

Modifié dans la version 3.3 : Negative values for *level* are no longer accepted.

*PyObject** **PyImport_Import**(*PyObject *name*)

Return value : New reference. This is a higher-level interface that calls the current "import hook function" (with an explicit *level* of 0, meaning absolute import). It invokes the `__import__()` function from the `__builtins__` of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

This function always uses absolute imports.

*PyObject** **PyImport_ReloadModule**(*PyObject *m*)

Return value : New reference. Reload a module. Return a new reference to the reloaded module, or *NULL* with an exception set on failure (the module still exists in this case).

*PyObject** **PyImport_AddModuleObject**(*PyObject *name*)

Return value : Borrowed reference. Return the module object corresponding to a module name. The *name* argument may be of the form `package.module`. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary. Return *NULL* with an exception set on failure.

Note : This function does not load or import the module ; if the module wasn't already loaded, you will get an empty module object. Use `PyImport_ImportModule()` or one of its variants to import a module. Package structures implied by a dotted name for *name* are not created if not already present.

Nouveau dans la version 3.3.

*PyObject** **PyImport_AddModule**(const char **name*)

Return value : Borrowed reference. Similar to `PyImport_AddModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

*PyObject** **PyImport_ExecCodeModule**(const char **name*, *PyObject *co*)

Return value : New reference. Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function `compile()`, load the module. Return a new reference to the module object, or *NULL* with an exception set if an error occurred. *name* is removed from `sys.modules` in error cases, even if *name* was already in `sys.modules` on entry to `PyImport_ExecCodeModule()`. Leaving incompletely initialized modules in `sys.modules` is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author's intents) state.

The module's `__spec__` and `__loader__` will be set, if not set already, with the appropriate values. The spec's loader will be set to the module's `__loader__` (if set) and to an instance of `SourceFileLoader` otherwise.

The module's `__file__` attribute will be set to the code object's `co_filename`. If applicable, `__cached__` will also be set.

This function will reload the module if it was already imported. See `PyImport_ReloadModule()` for the intended way to reload a module.

If *name* points to a dotted name of the form `package.module`, any package structures not already created will still not be created.

See also `PyImport_ExecCodeModuleEx()` and `PyImport_ExecCodeModuleWithPathnames()`.

*PyObject** **`PyImport_ExecCodeModuleEx`**(const char **name*, *PyObject* **co*, const char **pathname*)

Return value : New reference. Like `PyImport_ExecCodeModule()`, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

See also `PyImport_ExecCodeModuleWithPathnames()`.

*PyObject** **`PyImport_ExecCodeModuleObject`**(*PyObject* **name*, *PyObject* **co*, *PyObject* **pathname*, *PyObject* **cpathname*)

Return value : New reference. Like `PyImport_ExecCodeModuleEx()`, but the `__cached__` attribute of the module object is set to *cpathname* if it is non-`NULL`. Of the three functions, this is the preferred one to use.

Nouveau dans la version 3.3.

*PyObject** **`PyImport_ExecCodeModuleWithPathnames`**(const char **name*, *PyObject* **co*, const char **pathname*, const char **cpathname*)

Return value : New reference. Like `PyImport_ExecCodeModuleObject()`, but *name*, *pathname* and *cpathname* are UTF-8 encoded strings. Attempts are also made to figure out what the value for *pathname* should be from *cpathname* if the former is set to `NULL`.

Nouveau dans la version 3.2.

Modifié dans la version 3.3 : Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

long **`PyImport_GetMagicNumber`**()

Return the magic number for Python bytecode files (a.k.a. `.pyc` file). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order. Returns `-1` on error.

Modifié dans la version 3.3 : Return value of `-1` upon failure.

const char * **`PyImport_GetMagicTag`**()

Return the magic tag string for **PEP 3147** format Python bytecode file names. Keep in mind that the value at `sys.implementation.cache_tag` is authoritative and should be used instead of this function.

Nouveau dans la version 3.2.

*PyObject** **`PyImport_GetModuleDict`**()

Return value : Borrowed reference. Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

*PyObject** **`PyImport_GetModule`**(*PyObject* **name*)

Return value : New reference. Return the already imported module with the given name. If the module has not been imported yet then returns NULL but does not set an error. Returns NULL and sets an error if the lookup failed.

Nouveau dans la version 3.7.

*PyObject** **`PyImport_GetImporter`**(*PyObject* **path*)

Return value : New reference. Return a finder object for a `sys.path`/`pkg.__path__` item *path*, possibly by fetching it from the `sys.path_importer_cache` dict. If it wasn't yet cached, traverse `sys.path_hooks` until a hook is found that can handle the path item. Return `None` if no hook could; this tells our caller that the *path based finder* could not find a finder for this path item. Cache the result in `sys.path_importer_cache`. Return a new reference to the finder object.

void **`_PyImport_Init`**()
Initialize the import mechanism. For internal use only.

void **`PyImport_Cleanup`**()
Empty the module table. For internal use only.

void **`_PyImport_Fini`**()
Finalize the import mechanism. For internal use only.

int **`PyImport_ImportFrozenModuleObject`**(*PyObject *name*)
Return value : New reference. Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use `PyImport_ImportModule()`. (Note the misnomer — this function would reload the module if it was already imported.)

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : The `__file__` attribute is no longer set on the module.

int **`PyImport_ImportFrozenModule`**(const char **name*)
Similar to `PyImport_ImportFrozenModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

struct **`_frozen`**
This is the structure type definition for frozen module descriptors, as generated by the **freeze** utility (see `Tools/freeze/` in the Python source distribution). Its definition, found in `Include/import.h`, is :

```
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
};
```

const struct *_frozen** **`PyImport_FrozenModules`**
This pointer is initialized to point to an array of `struct _frozen` records, terminated by one whose members are all *NULL* or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

int **`PyImport_AppendInittab`**(const char **name*, *PyObject** (**initfunc*)(void))
Add a single module to the existing table of built-in modules. This is a convenience wrapper around `PyImport_ExtendInittab()`, returning `-1` if the table could not be extended. The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import. This should be called before `Py_Initialize()`.

struct **`_inittab`**
Structure describing a single entry in the list of built-in modules. Each of these structures gives the name and initialization function for a module built into the interpreter. The name is an ASCII encoded string. Programs which embed Python may use an array of these structures in conjunction with `PyImport_ExtendInittab()` to provide additional built-in modules. The structure is defined in `Include/import.h` as :

```
struct _inittab {
    const char *name;           /* ASCII encoded string */
    PyObject* (*initfunc)(void);
};
```

int **PyImport_ExtendInittab** (struct *_inittab* **newtab*)
Add a collection of modules to the table of built-in modules. The *newtab* array must end with a sentinel entry which contains *NULL* for the `name` field ; failure to provide the sentinel value can result in a memory fault. Returns `0` on success or `-1` if insufficient memory could be allocated to extend the internal table. In the event of failure, no modules are added to the internal table. This should be called before `Py_Initialize()`.

6.5 Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format : version 0 is the historical version, version 1 shares interned strings in the file, and upon unmarshalling. Version 2 uses a binary format for floating point numbers. *Py_MARSHAL_VERSION* indicates the current file format (currently 2).

void **PyMarshal_WriteLongToFile** (long *value*, FILE **file*, int *version*)
Marshal a `long` integer, *value*, to *file*. This will only write the least-significant 32 bits of *value* ; regardless of the size of the native `long` type. *version* indicates the file format.

void **PyMarshal_WriteObjectToFile** (*PyObject* **value*, FILE **file*, int *version*)
Marshal a Python object, *value*, to *file*. *version* indicates the file format.

*PyObject** **PyMarshal_WriteObjectToString** (*PyObject* **value*, int *version*)
Return value : New reference. Return a bytes object containing the marshalled representation of *value*. *version* indicates the file format.

The following functions allow marshalled values to be read back in.

long **PyMarshal_ReadLongFromFile** (FILE **file*)
Return a C `long` from the data stream in a `FILE*` opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of `long`.

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

int **PyMarshal_ReadShortFromFile** (FILE **file*)
Return a C `short` from the data stream in a `FILE*` opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of `short`.

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

*PyObject** **PyMarshal_ReadObjectFromFile** (FILE **file*)
Return value : New reference. Return a Python object from the data stream in a `FILE*` opened for reading.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns *NULL*.

*PyObject** **PyMarshal_ReadLastObjectFromFile** (FILE **file*)
Return value : New reference. Return a Python object from the data stream in a `FILE*` opened for reading. Unlike `PyMarshal_ReadObjectFromFile()`, this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won't be reading anything else from the file.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns *NULL*.

*PyObject** **PyMarshal_ReadObjectFromString** (const char **data*, Py_ssize_t *len*)
Return value : New reference. Return a Python object from the data stream in a byte buffer containing *len* bytes pointed to by *data*.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns *NULL*.

6.6 Analyse des arguments et construction des valeurs

Ces fonctions sont utiles pour créer vos propres fonctions et méthodes d'extensions. Des informations supplémentaires et des exemples sont disponibles ici : extending-index.

Dans Les trois premières de ces fonctions décrites, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, et `PyArg_Parse()`, toutes utilisent *des chaînes de format* qui sont utilisées pour indiquer à la fonction les arguments attendus. Les chaînes de format utilise la même syntaxe pour chacune de ces fonctions.

6.6.1 Analyse des arguments

Une chaîne de format se compose de zéro ou plusieurs "unités de format". Une unité de format décrit un objet Python, elle est généralement composée d'un seul caractère ou d'une séquence d'unités de format entre parenthèses. À quelques exceptions près, une unité de format qui n'est pas une séquence entre parenthèses correspond normalement à un argument d'une seule adresse pour ces fonctions. Dans la description qui suit, la forme entre guillemets est l'unité de format, l'entrée entre parenthèses est le type d'objet Python qui correspond à l'unité de format, et l'entrée entre crochets est le type de la variable C (ou des variables) dont l'adresse doit être donnée.

Chaînes et tampons

Ces formats permettent d'accéder à un objet sous forme d'un fragment de mémoire contigüe. Il n'est pas nécessaire d'allouer la mémoire pour l'*unicode* ou le *bytes* renvoyé.

In general, when a format sets a pointer to a buffer, the buffer is managed by the corresponding Python object, and the buffer shares the lifetime of this object. You won't have to release any memory yourself. The only exceptions are `es`, `es#`, `et` and `et#`.

Néanmoins, quand une structure `Py_buffer` est en cours de remplissage, le tampon sous-jacent est verrouillé pour permettre à l'appelant d'utiliser le tampon par la suite, même à l'intérieur d'un bloc `Py_BEGIN_ALLOW_THREADS`, sans risques de voir des données muables se faire redimensionner ou supprimer. En conséquence, **il vous appartient d'appeler** `PyBuffer_Release()` après avoir terminé de traiter les données (ou après une interruption prématurée du traitement de ces données).

Sauf indication contraire, les tampons ne se terminent pas par NUL.

Some formats require a read-only *bytes-like object*, and set a pointer instead of a buffer structure. They work by checking that the object's `PyBufferProcs.bf_releasebuffer` field is *NULL*, which disallows mutable objects such as `bytearray`.

Note : Pour toutes les variantes du marqueur `#` (`s#`, `y#`, etc), le type de l'argument *length* (*int* ou `Py_ssize_t`) est contrôlé en définissant la macro `PY_SSIZE_T_CLEAN` avant d'inclure le fichier `Python.h`. Si la macro est définie, la longueur est de type `Py_ssize_t` au lieu d'être de type `int`. Ce comportement changera dans une future version de Python, qui supportera seulement `Py_ssize_t` a la place de `int`. Il est préférable de toujours définir `PY_SSIZE_T_CLEAN`.

`s` (`str`) [`const char *`] Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points ; if it does, a `ValueError` exception is raised. Unicode objects are converted to C strings using `'utf-8'` encoding. If this conversion fails, a `UnicodeError` is raised.

Note : This format does not accept *bytes-like objects*. If you want to accept filesystem paths and convert them to C character strings, it is preferable to use the `O&` format with `PyUnicode_FSConverter()` as *converter*.

Modifié dans la version 3.5 : Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

`s*` (`str` ou *bytes-like object*) [`Py_buffer`] This format accepts Unicode objects as well as bytes-like objects. It fills a `Py_buffer` structure provided by the caller. In this case the resulting C string may contain embedded NUL bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`s#` (`str`, read-only *bytes-like object*) [const char *, int or `Py_ssize_t`] Like `s*`, except that it doesn't accept mutable objects. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`z` (`str` ou `None`) [`const char *`] Comme `s`, mais l'objet Python peut aussi être `None`, auquel cas le pointeur C devient *NULL*.

`z*` (`str`, *bytes-like object* ou `None`) [`Py_buffer`] Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the `Py_buffer` structure is set to *NULL*.

`z#` (`str`, read-only *bytes-like object* or `None`) [const char *, int] Comme `s#`, mais l'objet Python peut également être `None`, dans ce cas le pointeur C est définie à *NULL*.

`y` (read-only *bytes-like object*) [const char *] This format converts a bytes-like object to a C pointer to a character string ; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes ; if it does, a `ValueError` exception is raised.

Modifié dans la version 3.5 : Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes buffer.

`y*` (*bytes-like object*) [`Py_buffer`] This variant on `s*` doesn't accept Unicode objects, only bytes-like objects. **This is the recommended way to accept binary data.**

`y#` (read-only *bytes-like object*) [const char *, int] Cette variante de `s#` n'accepte pas les objets Unicode, uniquement des objets assimilés à des octets.

`S` (`bytes`) [`PyBytesObject *`] Requires that the Python object is a `bytes` object, without attempting any conversion. Raises `TypeError` if the object is not a bytes object. The C variable may also be declared as `PyObject*`.

`Y` (`bytearray`) [`PyByteArrayObject *`] Requires that the Python object is a `bytearray` object, without attempting any conversion. Raises `TypeError` if the object is not a `bytearray` object. The C variable may also be declared as `PyObject*`.

`u` (`str`) [`const Py_UNICODE *`] Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of Unicode characters. You must pass the address of a `Py_UNICODE` pointer variable, which will be filled with the pointer to an existing Unicode buffer. Please note that the width of a `Py_UNICODE` character depends on compilation options (it is either 16 or 32 bits). The Python string must not contain embedded null code points ; if it does, a `ValueError` exception is raised.

Modifié dans la version 3.5 : Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsWideCharString()`.

`u#` (`str`) [`const Py_UNICODE *, int`] This variant on `u` stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. This variant allows null code points.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsWideCharString()`.

`Z` (`str` ou `None`) [`const Py_UNICODE *`] Comme `u`, mais l'objet Python peut aussi être `None`, auquel cas le pointeur `Py_UNICODE` vaut *NULL*.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsWideCharString()`.

Z# (str ou None) [const Py_UNICODE *, int] Comme `u#`, mais l'objet Python peut également être `None`, auquel cas le pointeur `Py_UNICODE` vaut *NULL*.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsWideCharString()`.

U (str) [PyObject *] Requires that the Python object is a Unicode object, without attempting any conversion. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as `PyObject*`.

w* (lecture-écriture *bytes-like object*) [Py_buffer] This format accepts any object which implements the read-write buffer interface. It fills a `Py_buffer` structure provided by the caller. The buffer may contain embedded null bytes. The caller have to call `PyBuffer_Release()` when it is done with the buffer.

es (str) [const char *encoding, char **buffer] This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or *NULL*, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**` ; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

`PyArg_ParseTuple()` will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after use.

et (str, bytes or bytearray) [const char *encoding, char **buffer] Same as `es` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

es# (str) [const char *encoding, char **buffer, int *buffer_length] This variant on `s#` is used for encoding Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

It requires three arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or *NULL*, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**` ; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer ; the referenced integer will be set to the number of bytes in the output buffer.

Il existe deux modes de fonctionnement :

If **buffer* points a *NULL* pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

If **buffer* points to a non-*NULL* pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as the buffer and interpret the initial value of **buffer_length* as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `ValueError` will be set.

Dans les deux cas, **buffer_length* est la longueur des données encodées, sans l'octet NUL de fin.

et# (str, bytes or bytearray) [const char *encoding, char **buffer, int *buffer_length] Same as `es#` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

Les nombres

b (int) [unsigned char] Convertit un entier Python positif ou nul en un `unsigned tiny int`, stocké dans un `unsigned char` C.

B (int) [unsigned char] Convertit un entier Python en un `tiny int` sans vérifier le débordement, stocké dans un `unsigned char` C.

h (int) [short int] Convertit un entier Python en un `short int` C.

H (int) [unsigned short int] Convertit un entier Python en un `unsigned short int` C, sans contrôle de débordement.

i (int) [int] Convertit un entier Python en un `int` C.

I (int) [unsigned int] Convertit un entier Python en un `unsigned int` C, sans contrôle de le débordement.

l (int) [long int] Convertit un entier Python en un `long int`.

k (int) [unsigned long] Convertit un entier Python en un `unsigned long` C sans en vérifier le débordement.

L (int) [long long] Convert a Python integer to a C `long long`.

K (int) [unsigned long long] Convert a Python integer to a C `unsigned long long` without overflow checking.

n (int) [Py_ssize_t] Convertit un entier Python en un `short int` C.

c (bytes ou bytearray de longueur 1) [char] Convertit un *byte* Python, représenté comme un objet `bytes` ou `bytearray` de longueur 1, en un `char` C.

Modifié dans la version 3.3 : Allow `bytearray` objects.

C (str de longueur 1) [int] Convertit un caractère Python, représenté comme un objet `str` de longueur 1, en un `int` C.

f (float) [float] Convertit un nombre flottant Python vers un `float`.

d (float) [double] Convertit un nombre flottant Python vers un `double` C.

D (complex) [Py_complex] Convertit un nombre complexe Python vers une structure *`Py_complex`* C.

Autres objets

O (objet) [PyObject *] Stocke un objet Python (sans aucune conversion) en un pointeur sur un objet C. Ainsi, Le programme C reçoit l'objet réel qui a été passé. Le compteur de référence sur l'objet n'est pas incrémenté. Le pointeur stocké n'est pas *NULL*.

O! (objet) [*typeobject*, PyObject *] Store a Python object in a C object pointer. This is similar to `O`, but takes two C arguments : the first is the address of a Python type object, the second is the address of the C variable (of type *`PyObject*`*) into which the object pointer is stored. If the Python object does not have the required type, `TypeError` is raised.

O& (objet) [*converter*, *anything*] Convert a Python object to a C variable through a *converter* function. This takes two arguments : the first is a function, the second is the address of a C variable (of arbitrary type), converted to `void *`. The *converter* function in turn is called as follows :

```
status = converter(object, address);
```

where *object* is the Python object to be converted and *address* is the `void*` argument that was passed to the *`PyArg_Parse*()`* function. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception and leave the content of *address* unmodified.

If the *converter* returns `Py_CLEANUP_SUPPORTED`, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the *object* parameter will be NULL ; *address* will have the same value as in the original call.

Modifié dans la version 3.1 : `Py_CLEANUP_SUPPORTED` à été ajouté.

p (bool) [int] Tests the value passed in for truth (a boolean **p**redicate) and converts the result to its equivalent C true/false integer value. Sets the int to `1` if the expression was true and `0` if it was false. This accepts any valid Python value. See truth for more information about how Python tests values for truth.

Nouveau dans la version 3.3.

(items) (**tuple**) [*matching-items*] L'objet doit être une séquence Python dont la longueur est le nombre d'unités de formats dans *articles*. Les arguments C doivent correspondre à chaque unité de format particulière dans *articles*. Les unités de formats pour les séquences peuvent être imbriquées.

It is possible to pass "long" integers (integers whose value exceeds the platform's `LONG_MAX`) however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

Quelques autres caractères ont un sens dans une chaîne de format. On ne doit pas les trouvées dans des parenthèses imbriquées. Ce sont :

| Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, `PyArg_ParseTuple()` does not touch the contents of the corresponding C variable(s).

$ `PyArg_ParseTupleAndKeywords()` only : Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so `|` must always be specified before `$` in the format string.

Nouveau dans la version 3.3.

: The list of format units ends here ; the string after the colon is used as the function name in error messages (the "associated value" of the exception that `PyArg_ParseTuple()` raises).

; La liste des unités de format s'arrête ici ; la chaîne après le point-virgule est utilise comme message d'erreur *au lieu* du message d'erreur par défaut. **:** et **;** sont mutuellement exclusifs.

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

Les arguments additionnels qui sont donnés à ces fonctions doivent être des adresses de variables dont le type est déterminé par la chaîne de format. Elles sont utilisées pour stocker les valeurs du n-uplet d'entrée. Il y a quelques cas, comme décrit précédemment dans le liste des unités de formats, où ces paramètres sont utilisés comme valeurs d'entrée. Dans ce cas, ils devraient correspondre à ce qui est spécifié pour l'unité de format correspondante.

For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. On success, the `PyArg_Parse*()` functions return true, otherwise they return false and raise an appropriate exception. When the `PyArg_Parse*()` functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

Fonction de l'API

int **PyArg_ParseTuple**(*PyObject* *args*, const char *format*, ...)

Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success ; on failure, it returns false and raises the appropriate exception.

int **PyArg_VaParse**(*PyObject* *args*, const char *format*, va_list *vargs*)

Identical to `PyArg_ParseTuple()`, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ParseTupleAndKeywords**(*PyObject* *args*, *PyObject* *kw*, const char *format*, char *keywords[]*, ...)

Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a *NULL*-terminated array of keyword parameter names. Empty names denote *positional-only parameters*. Returns true on success ; on failure, it returns false and raises the appropriate exception.

Modifié dans la version 3.6 : Added support for *positional-only parameters*.

int **PyArg_VaParseTupleAndKeywords**(*PyObject* *args*, *PyObject* *kw*, const char *format*, char *keywords[]*, va_list *vargs*)

Identical to `PyArg_ParseTupleAndKeywords()`, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ValidateKeywordArguments** (*PyObject* *)

Ensure that the keys in the keywords argument dictionary are strings. This is only needed if `PyArg_ParseTupleAndKeywords()` is not used, since the latter already does this check.

Nouveau dans la version 3.2.

int **PyArg_Parse** (*PyObject* *args*, const char *format*, ...)

Function used to deconstruct the argument lists of "old-style" functions — these are functions which use the `METH_OLDARGS` parameter parsing method, which has been removed in Python 3. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

int **PyArg_UnpackTuple** (*PyObject* *args*, const char *name*, Py_ssize_t *min*, Py_ssize_t *max*, ...)

A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed as *args* ; it must actually be a tuple. The length of the tuple must be at least *min* and no more than *max* ; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a `PyObject*` variable ; these will be filled in with the values from *args* ; they will contain borrowed references. The variables which correspond to optional parameters not given by *args* will not be filled in ; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements ; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references :

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

The call to `PyArg_UnpackTuple()` in this example is entirely equivalent to this call to `PyArg_ParseTuple()` :

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

6.6.2 Construction des valeurs

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

Return value : New reference. Create a new value based on a format string similar to those accepted by the `PyArg_Parse*()` family of functions and a sequence of values. Returns the value or *NULL* in the case of an error ; an exception will be raised if *NULL* is returned.

`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.

When memory buffers are passed as parameters to supply data to build objects, as for the `s` and `s#` formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by `Py_BuildValue()`. In other words, if your code invokes `malloc()` and passes the allocated memory to

`Py_BuildValue()`, your code is responsible for calling `free()` for that memory once `Py_BuildValue()` returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

s (str ou None) [const char *] Convert a null-terminated C string to a Python `str` object using `'utf-8'` encoding. If the C string pointer is *NULL*, `None` is used.

s# (str ou None) [const char *, int] Convert a C string and its length to a Python `str` object using `'utf-8'` encoding. If the C string pointer is *NULL*, the length is ignored and `None` is returned.

y (bytes) [const char *] This converts a C string to a Python `bytes` object. If the C string pointer is *NULL*, `None` is returned.

y# (bytes) [const char *, int] This converts a C string and its lengths to a Python object. If the C string pointer is *NULL*, `None` is returned.

z (str ou None) [const char *] Same as `s`.

z# (str ou None) [const char *, int] Same as `s#`.

u (str) [const wchar_t *] Convert a null-terminated `wchar_t` buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is *NULL*, `None` is returned.

u# (str) [const wchar_t *, int] Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is *NULL*, the length is ignored and `None` is returned.

U (str ou None) [const char *] Same as `s`.

U# (str ou None) [const char *, int] Same as `s#`.

i (int) [int] Convert a plain C `int` to a Python integer object.

b (int) [char] Convert a plain C `char` to a Python integer object.

h (int) [short int] Convert a plain C `short int` to a Python integer object.

l (int) [long int] Convertit un `long int` en un *int* Python.

B (int) [unsigned char] Convert a C `unsigned char` to a Python integer object.

H (int) [unsigned short int] Convert a C `unsigned short int` to a Python integer object.

I (int) [unsigned int] Convert a C `unsigned int` to a Python integer object.

k (int) [unsigned long] Convert a C `unsigned long` to a Python integer object.

L (int) [long long] Convert a C `long long` to a Python integer object.

K (int) [unsigned long long] Convert a C `unsigned long long` to a Python integer object.

n (int) [Py_ssize_t] Convert a C `Py_ssize_t` to a Python integer.

c (bytes de taille 1) [char] Convert a C `int` representing a byte to a Python `bytes` object of length 1.

C (str de longueur 1) [int] Convert a C `int` representing a character to Python `str` object of length 1.

d (float) [double] Convert a C `double` to a Python floating point number.

f (float) [float] Convert a C `float` to a Python floating point number.

D (complex) [Py_complex *] Convert a C `Py_complex` structure to a Python complex number.

O (objet) [PyObject *] Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a *NULL* pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, `Py_BuildValue()` will return *NULL* but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

S (objet) [PyObject *] Same as `O`.

N (objet) [PyObject *] Same as `O`, except it doesn't increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.

O& **(objet) [*converter, anything*]** Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with `void *`) as its argument and should return a "new" Python object, or *NULL* if an error occurred.

(items) **(tuple) [*matching-items*]** Convert a sequence of C values to a Python tuple with the same number of items.

[items] **(list) [*matching-items*]** Convert a sequence of C values to a Python list with the same number of items.

{items} **(dict) [*matching-items*]** Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and *NULL* returned.

*PyObject** **Py_VaBuildValue** (const char **format*, va_list *vargs*)
Return value : New reference. Identical to `Py_BuildValue()`, except that it accepts a va_list rather than a variable number of arguments.

6.7 Conversion et formatage de chaînes

Fonctions de conversion pour les nombres et pour la sortie des chaînes formatées.

int **PyOS_snprintf** (char **str*, size_t *size*, const char **format*, ...)
Output not more than *size* bytes to *str* according to the format string *format* and the extra arguments. See the Unix man page `snprintf(2)`.

int **PyOS_vsnprintf** (char **str*, size_t *size*, const char **format*, va_list *va*)
Output not more than *size* bytes to *str* according to the format string *format* and the variable argument list *va*. Unix man page `vsnprintf(2)`.

`PyOS_snprintf()` and `PyOS_vsnprintf()` wrap the Standard C library functions `snprintf()` and `vsnprintf()`. Their purpose is to guarantee consistent behavior in corner cases, which the Standard C functions do not.

The wrappers ensure that *str*[*size*-1] is always `'\0'` upon return. They never write more than *size* bytes (including the trailing `'\0'`) into str. Both functions require that `str != NULL`, `size > 0` and `format != NULL`.

If the platform doesn't have `vsnprintf()` and the buffer size needed to avoid truncation exceeds *size* by more than 512 bytes, Python aborts with a *Py_FatalError*.

The return value (*rv*) for these functions should be interpreted as follows :

- When `0 <= rv < size`, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing `'\0'` byte at *str*[*rv*]).
- When `rv >= size`, the output conversion was truncated and a buffer with `rv + 1` bytes would have been needed to succeed. *str*[*size*-1] is `'\0'` in this case.
- When `rv < 0`, "something bad happened." *str*[*size*-1] is `'\0'` in this case too, but the rest of *str* is undefined. The exact cause of the error depends on the underlying platform.

The following functions provide locale-independent string to number conversions.

double **PyOS_string_to_double** (const char **s*, char ***endptr*, *PyObject* **overflow_exception*)
Convert a string `s` to a `double`, raising a Python exception on failure. The set of accepted strings corresponds to the set of strings accepted by Python's `float()` constructor, except that `s` must not have leading or trailing whitespace. The conversion is independent of the current locale.

If `endptr` is `NULL`, convert the whole string. Raise `ValueError` and return `-1.0` if the string is not a valid representation of a floating-point number.

If endptr is not `NULL`, convert as much of the string as possible and set `*endptr` to point to the first unconverted character. If no initial segment of the string is the valid representation of a floating-point number, set `*endptr` to point to the beginning of the string, raise ValueError, and return `-1.0`.

If `s` represents a value that is too large to store in a float (for example, `"1e500"` is such a string on many platforms) then if `overflow_exception` is `NULL` return `Py_HUGE_VAL` (with an appropriate sign) and don't set any exception. Otherwise, `overflow_exception` must point to a Python exception object ; raise that exception and return `-1.0`. In both cases, set `*endptr` to point to the first character after the converted value.

If any other error occurs during the conversion (for example an out-of-memory error), set the appropriate Python exception and return `-1.0`.

Nouveau dans la version 3.1.

char* **PyOS_double_to_string**(double *val*, char *format_code*, int *precision*, int *flags*, int **ptype*)

Convert a `double` *val* to a string using supplied *format_code*, *precision*, and *flags*.

format_code must be one of `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'r'`. For `'r'`, the supplied *precision* must be 0 and is ignored. The `'r'` format code specifies the standard `repr()` format.

flags can be zero or more of the values *Py_DTSF_SIGN*, *Py_DTSF_ADD_DOT_0*, or *Py_DTSF_ALT*, or-ed together :

— *Py_DTSF_SIGN* means to always precede the returned string with a sign character, even if *val* is non-negative.
— *Py_DTSF_ADD_DOT_0* means to ensure that the returned string will not look like an integer.
— *Py_DTSF_ALT* means to apply "alternate" formatting rules. See the documentation for the `PyOS_snprintf()` `'#'` specifier for details.

If *ptype* is non-NULL, then the value it points to will be set to one of *Py_DTST_FINITE*, *Py_DTST_INFINITE*, or *Py_DTST_NAN*, signifying that *val* is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to *buffer* with the converted string or *NULL* if the conversion failed. The caller is responsible for freeing the returned string by calling `PyMem_Free()`.

Nouveau dans la version 3.1.

int **PyOS_stricmp**(const char **s1*, const char **s2*)

Case insensitive comparison of strings. The function works almost identically to `strcmp()` except that it ignores the case.

int **PyOS_strnicmp**(const char **s1*, const char **s2*, Py_ssize_t *size*)

Case insensitive comparison of strings. The function works almost identically to `strncmp()` except that it ignores the case.

6.8 Réflexion

*PyObject** **PyEval_GetBuiltins**()

Return value : Borrowed reference. Renvoie un dictionnaire des fonctions natives de la *frame* en cours d'exécution, ou si aucune *frame* n'est exécutée, les fonctions natives du *thread* indiqué par le *thread state*.

*PyObject** **PyEval_GetLocals**()

Return value : Borrowed reference. Renvoie un dictionnaire des variables locales de la *frame* en cours d'exécution, ou *NULL* si aucune *frame* n'est en cours d'exécution.

*PyObject** **PyEval_GetGlobals**()

Return value : Borrowed reference. Renvoie un dictionnaire des variables globales de la *frame* en cours d'exécution ou *NULL* si aucune *frame* n'est en cours d'exécution.

*PyFrameObject** **PyEval_GetFrame**()

Return value : Borrowed reference. Renvoie la *frame* actuelle selon le *thread state*, qui est *NULL* si aucune *frame* n'est en cours d'exécution.

int **PyFrame_GetLineNumber**(*PyFrameObject* **frame*)

Renvoie le numéro de ligne que *frame* est en train d'exécuter

const char* **PyEval_GetFuncName**(*PyObject* **func*)

Renvoie le nom de *func* s'il s'agit d'une fonction, d'une classe ou d'un objet d'instance, sinon le nom du type de *func* const char* **PyEval_GetFuncDesc** (*PyObject* **func*)

Renvoie une description en chaîne de caractères, en fonction du type de *func*. Les valeurs renvoyées peuvent être `"()"` pour les fonction et les méthodes, `\" constructor\"`, `\" instance\"`, `\" object\"`. Concaténé avec le résultat de `PyEval_GetFuncName()`, le résultat sera une description de *func*

6.9 Codec registry and support functions

int **PyCodec_Register** (*PyObject* **search_function*)

Register a new codec search function.

As side effect, this tries to load the `encodings` package, if not yet done, to make sure that it is always first in the list of search functions.

int **PyCodec_KnownEncoding** (const char **encoding*)

Return `1` or `0` depending on whether there is a registered codec for the given *encoding*. This function always succeeds.

*PyObject** **PyCodec_Encode** (*PyObject* **object*, const char **encoding*, const char **errors*)

Return value : New reference. Generic codec based encoding API.

object is passed through the encoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be *NULL* to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

*PyObject** **PyCodec_Decode** (*PyObject* **object*, const char **encoding*, const char **errors*)

Return value : New reference. Generic codec based decoding API.

object is passed through the decoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be *NULL* to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

6.9.1 Codec lookup API

In the following functions, the *encoding* string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a `KeyError` is set and *NULL* returned.

*PyObject** **PyCodec_Encoder** (const char **encoding*)

Return value : New reference. Get an encoder function for the given *encoding*.

*PyObject** **PyCodec_Decoder** (const char **encoding*)

Return value : New reference. Get a decoder function for the given *encoding*.

*PyObject** **PyCodec_IncrementalEncoder** (const char **encoding*, const char **errors*)

Return value : New reference. Get an `IncrementalEncoder` object for the given *encoding*.

*PyObject** **PyCodec_IncrementalDecoder** (const char **encoding*, const char **errors*)

Return value : New reference. Get an `IncrementalDecoder` object for the given *encoding*.

*PyObject** **PyCodec_StreamReader** (const char **encoding*, *PyObject* **stream*, const char **errors*)

Return value : New reference. Get a `StreamReader` factory function for the given *encoding*.

*PyObject** **PyCodec_StreamWriter** (const char **encoding*, *PyObject* **stream*, const char **errors*)

Return value : New reference. Get a `StreamWriter` factory function for the given *encoding*.

6.9.2 Registry API for Unicode encoding error handlers

int **PyCodec_RegisterError**(const char **name*, *PyObject* **error*)

Register the error handling callback function *error* under the given *name*. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and *name* is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of `UnicodeEncodeError`, `UnicodeDecodeError` or `UnicodeTranslateError` that holds information about the problematic sequence of characters or bytes and their offset in the original string (see *Objets exception Unicode* for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return `0` on success, `-1` on error.

*PyObject** **PyCodec_LookupError**(const char **name*)

Return value : New reference. Lookup the error handling callback function registered under *name*. As a special case *NULL* can be passed, in which case the error handling callback for "strict" will be returned.

*PyObject** **PyCodec_StrictErrors**(*PyObject* **exc*)

Return value : Always NULL. Raise *exc* as an exception.

*PyObject** **PyCodec_IgnoreErrors**(*PyObject* **exc*)

Return value : New reference. Ignore the unicode error, skipping the faulty input.

*PyObject** **PyCodec_ReplaceErrors**(*PyObject* **exc*)

Return value : New reference. Replace the unicode encode error with `?` or `U+FFFD`.

*PyObject** **PyCodec_XMLCharRefReplaceErrors**(*PyObject* **exc*)

Return value : New reference. Replace the unicode encode error with XML character references.

*PyObject** **PyCodec_BackslashReplaceErrors**(*PyObject* **exc*)

Return value : New reference. Replace the unicode encode error with backslash escapes (`\x`, `\u` and `\U`).

*PyObject** **PyCodec_NameReplaceErrors**(*PyObject* **exc*)

Return value : New reference. Replace the unicode encode error with `\N{...}` escapes.

Nouveau dans la version 3.5.

CHAPITRE 7

Couche d'abstraction des objets

Dans ce chapitre, les fonctions s'appliquent à des objets Python sans tenir compte de leur type, ou des classes d'objets au sens large (par exemple, tous les types numériques, ou tous les types de séquence). Quand ils sont utilisés sur des types d'objets qui ne correspondent pas, ils lèveront une exception Python.

Il n'est pas possible d'utiliser ces fonctions sur des objets qui n'ont pas été correctement initialisés, comme un objet liste qui a été créé avec `PyList_New()` mais dont les éléments n'ont pas encore été mis à une valeur non-`NULL`.

7.1 Protocole Objet

*PyObject** **`Py_NotImplemented`**
Le singleton `NotImplemented`, utilisé pour signaler qu'une opération n'est pas implémentée pour la combinaison de types en question.

`Py_RETURN_NOTIMPLEMENTED`
Traite proprement le renvoi de `Py_NotImplemented` depuis l'intérieur d'une fonction C (c'est-à-dire, incrémente le compteur de référence de *NotImplemented* et le renvoie).

int **`PyObject_Print`**(*PyObject *o*, FILE **fp*, int *flags*)
Écrit un objet *o*, dans le fichier *fp*. Renvoie `-1` en cas d'erreur. L'argument *flags* est utilisé pour permettre certaines options de rendu. La seule option actuellement gérée est `Py_PRINT_RAW` ; si cet argument est fourni, le `str()` de l'objet est utilisé pour le rendu à la place de `repr()`.

int **`PyObject_HasAttr`**(*PyObject *o*, *PyObject *attr_name*)
Renvoie `1` si *o* a l'attribut *attr_name*, et `0` sinon. Ceci est équivalent à l'expression Python `hasattr(o, attr_name)`. Cette fonction réussit toujours.

Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods will get suppressed. To get error reporting use `PyObject_GetAttr()` instead.

int **`PyObject_HasAttrString`**(*PyObject *o*, const char **attr_name*)
Renvoie `1` si *o* a l'attribut *attr_name*, et `0` sinon. Ceci est équivalent à l'expression Python `hasattr(o, attr_name)`. Cette fonction réussit toujours.

Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods and creating a temporary string object will get suppressed. To get error reporting use `PyObject_GetAttrString()` instead.

*PyObject** **PyObject_GetAttr** (*PyObject *o*, *PyObject *attr_name*)
Return value : New reference. Récupère l'attribut nommé *attr_name* de l'objet *o*. Renvoie la valeur de l'attribut en cas de succès, ou *NULL* en cas d'échec. Ceci est équivalent à l'expression Python `o.attr_name`.

*PyObject** **PyObject_GetAttrString** (*PyObject *o*, const char **attr_name*)
Return value : New reference. Récupère un attribut nommé *attr_name* de l'objet *o*. Renvoie la valeur de l'attribut en cas de succès, ou *NULL* en cas d'échec. Ceci est équivalent à l'expression Python `o.attr_name`.

*PyObject** **PyObject_GenericGetAttr** (*PyObject *o*, *PyObject *name*)
Return value : New reference. Accesseur d'attribut générique destiné à être mis dans le *slot* `tp_getattro` d'un objet type. Recherche un descripteur dans le dictionnaire de classes du MRO de l'objet ainsi qu'un attribut dans le `__dict__` de l'objet (si présent). Comme défini dans descriptors, les descripteurs de données sont prioritaires sur les attributs d'instance, contrairement aux autres descripteurs. Sinon, une `AttributeError` est levée.

int **PyObject_SetAttr** (*PyObject *o*, *PyObject *attr_name*, *PyObject *v*)
Définit la valeur de l'attribut nommé *attr_name*, pour l'objet *o*, à la valeur *v*. Lève une exception et renvoie `-1` en cas d'échec ; renvoie `0` en cas de succès. Ceci est équivalent à l'instruction Python `o.attr_name = v`.

Si *v* est *NULL*, l'attribut est supprimé. Cette fonctionnalité est obsolète,nous vous conseillons d'utiliser `PyObject_DelAttr()`.

int **PyObject_SetAttrString** (*PyObject *o*, const char **attr_name*, *PyObject *v*)
Définit la valeur de l'attribut nommé *attr_name*, pour l'objet *o*, à la valeur *v*. Lève une exception et renvoie `-1` en cas d'échec ; renvoie `0` en cas de succès. Ceci est équivalent à l'instruction Python `o.attr_name = v`.

Si *v* est *NULL*, l'attribut est supprimé. Cette fonctionnalité est obsolète, nous vous conseillons d'utiliser `PyObject_DelAttr()`.

int **PyObject_GenericSetAttr** (*PyObject *o*, *PyObject *name*, *PyObject *value*)
Accesseur et suppresseur générique d'attributs qui est fait pour être mis dans le `tp_setattro` d'un objet type. Il cherche un descripteur de données dans le dictionnaire de classes dans le MRO de l'objet et, si ce descripteur est trouvé, c'est lui qui est utilisé de préférence pour la suppression et la définition de l'attribut dans le dictionnaire d'instance. Sinon, l'attribut est défini ou supprimé dans le `__dict__` de l'objet (si présent). En cas de succès, `0` est renvoyé, sinon une `AttributeError` est levée et `-1` est renvoyé.

int **PyObject_DelAttr** (*PyObject *o*, *PyObject *attr_name*)
Supprime l'attribut nommé *attr_name*, pour l'objet *o*. Renvoie `-1` en cas d'échec. Ceci est l'équivalent de l'expression Python `del o.attr_name`.

int **PyObject_DelAttrString** (*PyObject *o*, const char **attr_name*)
Supprime l'attribut nommé *attr_name*, pour l'objet *o*. Renvoie `-1` en cas d'échec. Ceci est l'équivalent de l'expression Python `del o.attr_name`.

*PyObject** **PyObject_GenericGetDict** (*PyObject *o*, void **context*)
Return value : New reference. Une implémentation générique de l'accesseur d'un descripteur d'un `__dict__`. Crée le dictionnaire si nécessaire.

Nouveau dans la version 3.3.

int **PyObject_GenericSetDict** (*PyObject *o*, void **context*)
Une implémentation générique du mutateur d'un descripteur de `__dict__`. Cette implémentation n'autorise pas la suppression du dictionnaire.

Nouveau dans la version 3.3.

*PyObject** **PyObject_RichCompare** (*PyObject *o1*, *PyObject *o2*, int *opid*)
Return value : New reference. Compare les valeurs de *o1* et *o2* en utilisant l'opération spécifiée par *opid*, qui doit être `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, ou `Py_GE`, correspondant à <, <=, ==, `!=`, >, ou >= respectivement.

Ceci est l'équivalent de l'expression Python `o1 op o2`, où `op` est l'opérateur correspondant à *opid*. Renvoie la valeur de la comparaison en cas de succès, ou *NULL* en cas d'échec.

int **PyObject_RichCompareBool** (*PyObject* **o1*, *PyObject* **o2*, int *opid*)
Compare les valeurs de *o1* et *o2* en utilisant l'opération spécifiée par *opid*, qui doit être `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, ou `Py_GE`, correspondant à <, <=, ==, !=, >, ou >= respectivement. Renvoie `-1` en cas d'erreur, `0` si le résultat est faux, et `1` sinon. Ceci est l'équivalent de l'expression Python `o1 op o2`, où `op` est l'opérateur correspondant à *opid*.

Note : Si *o1* et *o2* sont le même objet, `PyObject_RichCompareBool()` renvoie toujours `1` pour `Py_EQ` et `0` pour `Py_NE`.

*PyObject** **PyObject_Repr** (*PyObject* **o*)
Return value : New reference. Calcule une représentation en chaîne de caractères de l'objet *o*. Renvoie la représentation en chaîne de caractères en cas de succès, *NULL* en cas d'échec. Ceci est l'équivalent de l'expression Python `repr(o)`. Appelé par la fonction native `repr()`.

Modifié dans la version 3.4 : Cette fonction inclut maintenant une assertion de débogage afin d'assurer qu'elle ne passe pas sous silence une exception active.

*PyObject** **PyObject_ASCII** (*PyObject* **o*)
Return value : New reference. Comme `PyObject_Repr()`, calcule une représentation en chaîne de caractères de l'objet *o*, mais échappe les caractères non ASCII dans la chaîne de caractères renvoyée par `PyObject_Repr()` avec' `\x`, `\u` ou `\U`. Cela génère une chaîne de caractères similaire à celle renvoyée par `PyObject_Repr()` en Python 2. Appelée par la fonction native `ascii()`.

*PyObject** **PyObject_Str** (*PyObject* **o*)
Return value : New reference. Calcule une représentation en chaîne de caractères de l'objet *o*. Renvoie la représentation en chaîne de caractères en cas de succès, *NULL* en cas d'échec. Ceci est l'équivalent de l'expression Python `str(o)`. Appelée par la fonction native `str()`, et, par conséquent, par la fonction `print()`.

Modifié dans la version 3.4 : Cette fonction inclut maintenant une assertion de débogage afin d'assurer qu'elle ne passe pas sous silence une exception active.

*PyObject** **PyObject_Bytes** (*PyObject* **o*)
Return value : New reference. Calcule une représentation en octets de l'objet *o*. *NULL* est renvoyé en cas d'échec, un objet séquence d'octets est renvoyé en cas de succès. Ceci est l'équivalent de l'expression Python `bytes(o)`, quand *o* n'est pas un entier. Contrairement à `bytes(o)`, une exception *TypeError* est levée lorsque *o* est un entier au lieu d'un objet octet initialisé avec des zéros.

int **PyObject_IsSubclass** (*PyObject* **derived*, *PyObject* **cls*)
Renvoie `1` si la classe *derived* est identique à ou dérivée de la classe *cls*, renvoie `0` sinon. En cas d'erreur, renvoie `-1`.

Si *cls* est un tuple, la vérification est menée sur chaque entrée de *cls*. Le résultat sera `1` quand au moins une des vérifications renvoie `1`, sinon ce sera `0`.

Si *cls* a une méthode `__subclasscheck__()`, elle est appelée pour déterminer le statut de la sous-classe comme décrit dans **PEP 3119**. Sinon, *derived* est une sous-classe de *cls* si c'est une sous-classe directe ou indirecte, c'est-à-dire contenue dans `cls.__mro__`.

Normalement seulement les classes objets, c'est-à-dire les instances de `type` ou d'une classe dérivée, sont considérées classes. Cependant, les objets peuvent surcharger cela en ayant un attribut `__bases__` (qui doit être un tuple de classes de bases).

int **PyObject_IsInstance** (*PyObject* **inst*, *PyObject* **cls*)
Renvoie `1` si *inst* est une instance de la classe *cls* ou une sous-classe de *cls*, ou `0` sinon. En cas d'erreur, renvoie `-1` et initialise une exception.

Si *cls* est un tuple, la vérification est menée sur chaque entrée de *cls*. Le résultat sera `1` quand au moins une des vérifications renvoie `1`, sinon ce sera `0`.

Si *cls* a une méthode `__subclasscheck__()`, elle sera appelée pour déterminer le statut de la sous-classe comme décrit dans **PEP 3119**. Sinon, *inst* est une instance *cls* si sa classe est une sous-classe de *cls*.

Une instance *inst* peut surcharger ce qui est considéré comme sa classe en ayant un attribut `__class__`.

Un objet *cls* peut surcharger s'il est considéré comme une classe, et ce que ses classes de bases sont, en ayant un attribut `__bases__` (qui doit être un tuple des classes de base).

int **PyCallable_Check**(*PyObject *o*)

Détermine si l'objet *o* est appelable. Renvoie `1` si c'est le cas, et `0` sinon. Cette fonction réussit toujours.

*PyObject** **PyObject_Call**(*PyObject *callable*, *PyObject *args*, *PyObject *kwargs*)

Return value : New reference. Appelle un objet Python appelable *callable*, avec des arguments donnés par le tuple *args*, et des arguments nommés donnés par le dictionnaire *kwargs*.

args ne doit pas être égal à *NULL*, utilisez un tuple vide si aucun argument n'est nécessaire. Si aucun argument nommé n'est nécessaire, *kwargs* peut être égal à *NULL*.

Renvoie le résultat de l'appel en cas de succès, ou lève une exception et renvoie *NULL* en cas d'échec.

Ceci est l'équivalent de l'expression Python : `callable(*args, **kwargs)`.

*PyObject** **PyObject_CallObject**(*PyObject *callable*, *PyObject *args*)

Return value : New reference. Appelle un objet Python appelable *callable*, avec des arguments donnés par le tuple *args*. Si aucun argument n'est nécessaire, alors *args* peut être égal à *NULL*.

Renvoie le résultat de l'appel en cas de succès, ou lève une exception et renvoie *NULL* en cas d'échec.

Ceci est l'équivalent de l'expression Python : `callable(*args)`.

*PyObject** **PyObject_CallFunction**(*PyObject *callable*, const char **format*, ...)

Return value : New reference. Appelle un objet Python appelable, avec un nombre variable d'arguments C. Les arguments C sont décrits par une chaîne de caractères de format de type `Py_BuildValue()`. Le format peut être *NULL*, indiquant qu'aucun argument n'est donné.

Renvoie le résultat de l'appel en cas de succès, ou lève une exception et renvoie *NULL* en cas d'échec.

Ceci est l'équivalent de l'expression Python : `callable(*args)`.

Note that if you only pass `PyObject *` args, `PyObject_CallFunctionObjArgs()` is a faster alternative.

Modifié dans la version 3.4 : The type of *format* was changed from `char *`.

*PyObject** **PyObject_CallMethod**(*PyObject *obj*, const char **name*, const char **format*, ...)

Return value : New reference. Call the method named *name* of object *obj* with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple.

The format can be *NULL*, indicating that no arguments are provided.

Renvoie le résultat de l'appel en cas de succès, ou lève une exception et renvoie *NULL* en cas d'échec.

This is the equivalent of the Python expression : `obj.name(arg1, arg2, ...)`.

Note that if you only pass `PyObject *` args, `PyObject_CallMethodObjArgs()` is a faster alternative.

Modifié dans la version 3.4 : The types of *name* and *format* were changed from `char *`.

*PyObject** **PyObject_CallFunctionObjArgs**(*PyObject *callable*, ..., NULL)

Return value : New reference. Call a callable Python object *callable*, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Renvoie le résultat de l'appel en cas de succès, ou lève une exception et renvoie *NULL* en cas d'échec.

This is the equivalent of the Python expression : `callable(arg1, arg2, ...)`.

*PyObject** **PyObject_CallMethodObjArgs**(*PyObject *obj*, *PyObject *name*, ..., NULL)

Return value : New reference. Calls a method of the Python object *obj*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Renvoie le résultat de l'appel en cas de succès, ou lève une exception et renvoie *NULL* en cas d'échec.

Py_hash_t **PyObject_Hash**(*PyObject *o*)

Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

Modifié dans la version 3.2 : The return type is now Py_hash_t. This is a signed integer the same size as Py_ssize_t.

Py_hash_t **PyObject_HashNotImplemented** (*PyObject *o*)
Set a `TypeError` indicating that `type(o)` is not hashable and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int **PyObject_IsTrue** (*PyObject *o*)
Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int **PyObject_Not** (*PyObject *o*)
Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

*PyObject** **PyObject_Type** (*PyObject *o*)
Return value : New reference. When *o* is non-*NULL*, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns *NULL*. This is equivalent to the Python expression `type(o)`. This function increments the reference count of the return value. There's really no reason to use this function instead of the common expression `o->ob_type`, which returns a pointer of type `PyTypeObject*`, except when the incremented reference count is needed.

int **PyObject_TypeCheck** (*PyObject *o*, *PyTypeObject *type*)
Return true if the object *o* is of type *type* or a subtype of *type*. Both parameters must be non-*NULL*.

Py_ssize_t **PyObject_Size** (*PyObject *o*)
Py_ssize_t **PyObject_Length** (*PyObject *o*)
Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

Py_ssize_t **PyObject_LengthHint** (*PyObject *o*, Py_ssize_t *default*)
Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, default)`.

Nouveau dans la version 3.4.

*PyObject** **PyObject_GetItem** (*PyObject *o*, *PyObject *key*)
Return value : New reference. Return element of *o* corresponding to the object *key* or *NULL* on failure. This is the equivalent of the Python expression `o[key]`.

int **PyObject_SetItem** (*PyObject *o*, *PyObject *key*, *PyObject *v*)
Map the object *key* to the value *v*. Raise an exception and return `-1` on failure ; return `0` on success. This is the equivalent of the Python statement `o[key] = v`.

int **PyObject_DelItem** (*PyObject *o*, *PyObject *key*)
Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

*PyObject** **PyObject_Dir** (*PyObject *o*)
Return value : New reference. This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or *NULL* if there was an error. If the argument is *NULL*, this is like the Python `dir()`, returning the names of the current locals ; in this case, if no execution frame is active then *NULL* is returned but `PyErr_Occurred()` will return false.

*PyObject** **PyObject_GetIter** (*PyObject *o*)
Return value : New reference. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns *NULL* if the object cannot be iterated.

7.2 Number Protocol

int **PyNumber_Check** (*PyObject *o*)

Returns 1 if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

*PyObject** **PyNumber_Add** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Returns the result of adding *o1* and *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 + o2`.

*PyObject** **PyNumber_Subtract** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Returns the result of subtracting *o2* from *o1*, or *NULL* on failure. This is the equivalent of the Python expression `o1 - o2`.

*PyObject** **PyNumber_Multiply** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Returns the result of multiplying *o1* and *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 * o2`.

*PyObject** **PyNumber_MatrixMultiply** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Returns the result of matrix multiplication on *o1* and *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 @ o2`.

Nouveau dans la version 3.5.

*PyObject** **PyNumber_FloorDivide** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Return the floor of *o1* divided by *o2*, or *NULL* on failure. This is equivalent to the "classic" division of integers.

*PyObject** **PyNumber_TrueDivide** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or *NULL* on failure. The return value is "approximate" because binary floating point numbers are approximate ; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers.

*PyObject** **PyNumber_Remainder** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. Returns the remainder of dividing *o1* by *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 % o2`.

*PyObject** **PyNumber_Divmod** (*PyObject *o1*, *PyObject *o2*)

Return value : New reference. See the built-in function `divmod()`. Returns *NULL* on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

*PyObject** **PyNumber_Power** (*PyObject *o1*, *PyObject *o2*, *PyObject *o3*)

Return value : New reference. See the built-in function `pow()`. Returns *NULL* on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass `Py_None` in its place (passing *NULL* for *o3* would cause an illegal memory access).

*PyObject** **PyNumber_Negative** (*PyObject *o*)

Return value : New reference. Returns the negation of *o* on success, or *NULL* on failure. This is the equivalent of the Python expression `-o`.

*PyObject** **PyNumber_Positive** (*PyObject *o*)

Return value : New reference. Returns *o* on success, or *NULL* on failure. This is the equivalent of the Python expression `+o`.

*PyObject** **PyNumber_Absolute** (*PyObject *o*)

Return value : New reference. Returns the absolute value of *o*, or *NULL* on failure. This is the equivalent of the Python expression `abs(o)`.

*PyObject** **PyNumber_Invert** (*PyObject *o*)

Return value : New reference. Returns the bitwise negation of *o* on success, or *NULL* on failure. This is the equivalent of the Python expression `~o`.

*PyObject** **PyNumber_Lshift** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of left shifting *o1* by *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 << o2`.

*PyObject** **PyNumber_Rshift** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of right shifting *o1* by *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 >> o2`.

*PyObject** **PyNumber_And** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the "bitwise and" of *o1* and *o2* on success and *NULL* on failure. This is the equivalent of the Python expression `o1 & o2`.

*PyObject** **PyNumber_Xor** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 ^ o2`.

*PyObject** **PyNumber_Or** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the "bitwise or" of *o1* and *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 | o2`.

*PyObject** **PyNumber_InPlaceAdd** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of adding *o1* and *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

*PyObject** **PyNumber_InPlaceSubtract** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of subtracting *o2* from *o1*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

*PyObject** **PyNumber_InPlaceMultiply** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of multiplying *o1* and *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

*PyObject** **PyNumber_InPlaceMatrixMultiply** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of matrix multiplication on *o1* and *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

Nouveau dans la version 3.5.

*PyObject** **PyNumber_InPlaceFloorDivide** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the mathematical floor of dividing *o1* by *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

*PyObject** **PyNumber_InPlaceTrueDivide** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or *NULL* on failure. The return value is "approximate" because binary floating point numbers are approximate ; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done *in-place* when *o1* supports it.

*PyObject** **PyNumber_InPlaceRemainder** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the remainder of dividing *o1* by *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

*PyObject** **PyNumber_InPlacePower** (*PyObject* *o1*, *PyObject* *o2*, *PyObject* *o3*)
Return value : New reference. See the built-in function `pow()`. Returns *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is `Py_None`, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass `Py_None` in its place (passing *NULL* for *o3* would cause an illegal memory access).

*PyObject** **PyNumber_InPlaceLshift** (*PyObject* *o1*, *PyObject* *o2*)
Return value : New reference. Returns the result of left shifting *o1* by *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

*PyObject** **PyNumber_InPlaceRshift** (*PyObject *o1*, *PyObject *o2*)
Return value : New reference. Returns the result of right shifting *o1* by *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

*PyObject** **PyNumber_InPlaceAnd** (*PyObject *o1*, *PyObject *o2*)
Return value : New reference. Returns the "bitwise and" of *o1* and *o2* on success and *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

*PyObject** **PyNumber_InPlaceXor** (*PyObject *o1*, *PyObject *o2*)
Return value : New reference. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

*PyObject** **PyNumber_InPlaceOr** (*PyObject *o1*, *PyObject *o2*)
Return value : New reference. Returns the "bitwise or" of *o1* and *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

*PyObject** **PyNumber_Long** (*PyObject *o*)
Return value : New reference. Returns the *o* converted to an integer object on success, or *NULL* on failure. This is the equivalent of the Python expression `int(o)`.

*PyObject** **PyNumber_Float** (*PyObject *o*)
Return value : New reference. Returns the *o* converted to a float object on success, or *NULL* on failure. This is the equivalent of the Python expression `float(o)`.

*PyObject** **PyNumber_Index** (*PyObject *o*)
Return value : New reference. Returns the *o* converted to a Python int on success or *NULL* with a `TypeError` exception raised on failure.

*PyObject** **PyNumber_ToBase** (*PyObject *n*, int *base*)
Return value : New reference. Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of `'0b'`, `'0o'`, or `'0x'`, respectively. If *n* is not a Python int, it is converted with `PyNumber_Index()` first.

Py_ssize_t **PyNumber_AsSsize_t** (*PyObject *o*, *PyObject *exc*)
Returns *o* converted to a Py_ssize_t value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python int but the attempt to convert to a Py_ssize_t value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is *NULL*, then the exception is cleared and the value is clipped to *PY_SSIZE_T_MIN* for a negative integer or *PY_SSIZE_T_MAX* for a positive integer.

int **PyIndex_Check** (*PyObject *o*)
Returns `1` if *o* is an index integer (has the nb_index slot of the tp_as_number structure filled in), and `0` otherwise. This function always succeeds.

7.3 Sequence Protocol

int **PySequence_Check** (*PyObject *o*)
Return `1` if the object provides sequence protocol, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method unless they are `dict` subclasses since in general case it is impossible to determine what the type of keys it supports. This function always succeeds.

Py_ssize_t **PySequence_Size** (*PyObject *o*)
Py_ssize_t **PySequence_Length** (*PyObject *o*)
Returns the number of objects in sequence *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

*PyObject** **PySequence_Concat** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Return the concatenation of *o1* and *o2* on success, and *NULL* on failure. This is the equivalent of the Python expression `o1 + o2`.

*PyObject** **PySequence_Repeat** (*PyObject *o*, Py_ssize_t *count*)
Return value: New reference. Return the result of repeating sequence object *o count* times, or *NULL* on failure. This is the equivalent of the Python expression `o * count`.

*PyObject** **PySequence_InPlaceConcat** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Return the concatenation of *o1* and *o2* on success, and *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

*PyObject** **PySequence_InPlaceRepeat** (*PyObject *o*, Py_ssize_t *count*)
Return value: New reference. Return the result of repeating sequence object *o count* times, or *NULL* on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

*PyObject** **PySequence_GetItem** (*PyObject *o*, Py_ssize_t *i*)
Return value: New reference. Return the *i*th element of *o*, or *NULL* on failure. This is the equivalent of the Python expression `o[i]`.

*PyObject** **PySequence_GetSlice** (*PyObject *o*, Py_ssize_t *i1*, Py_ssize_t *i2*)
Return value: New reference. Return the slice of sequence object *o* between *i1* and *i2*, or *NULL* on failure. This is the equivalent of the Python expression `o[i1:i2]`.

int **PySequence_SetItem** (*PyObject *o*, Py_ssize_t *i*, *PyObject *v*)
Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *v*.

If *v* is *NULL*, the element is deleted, however this feature is deprecated in favour of using *`PySequence_DelItem()`*.

int **PySequence_DelItem** (*PyObject *o*, Py_ssize_t *i*)
Delete the *i*th element of object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i]`.

int **PySequence_SetSlice** (*PyObject *o*, Py_ssize_t *i1*, Py_ssize_t *i2*, *PyObject *v*)
Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. This is the equivalent of the Python statement `o[i1:i2] = v`.

int **PySequence_DelSlice** (*PyObject *o*, Py_ssize_t *i1*, Py_ssize_t *i2*)
Delete the slice in sequence object *o* from *i1* to *i2*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i1:i2]`.

Py_ssize_t **PySequence_Count** (*PyObject *o*, *PyObject *value*)
Return the number of occurrences of *value* in *o*, that is, return the number of keys for which `o[key] == value`. On failure, return `-1`. This is equivalent to the Python expression `o.count(value)`.

int **PySequence_Contains** (*PyObject *o*, *PyObject *value*)
Determine if *o* contains *value*. If an item in *o* is equal to *value*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `value in o`.

Py_ssize_t **PySequence_Index** (*PyObject *o*, *PyObject *value*)
Return the first index *i* for which `o[i] == value`. On error, return `-1`. This is equivalent to the Python expression `o.index(value)`.

*PyObject** **PySequence_List** (*PyObject *o*)
Return value: New reference. Return a list object with the same contents as the sequence or iterable *o*, or *NULL* on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

*PyObject** **PySequence_Tuple** (*PyObject *o*)
Return value: New reference. Return a tuple object with the same contents as the sequence or iterable *o*, or *NULL*

on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

*PyObject** **PySequence_Fast** (*PyObject* **o*, const char **m*)
Return value : New reference. Return the sequence or iterable *o* as a list, unless it is already a tuple or list, in which case *o* is returned. Use `PySequence_Fast_GET_ITEM()` to access the members of the result. Returns *NULL* on failure. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text.

Py_ssize_t **PySequence_Fast_GET_SIZE** (*PyObject* **o*)
Returns the length of *o*, assuming that *o* was returned by `PySequence_Fast()` and that *o* is not *NULL*. The size can also be gotten by calling `PySequence_Size()` on *o*, but `PySequence_Fast_GET_SIZE()` is faster because it can assume *o* is a list or tuple.

*PyObject** **PySequence_Fast_GET_ITEM** (*PyObject* **o*, Py_ssize_t *i*)
Return value : Borrowed reference. Return the *i*th element of *o*, assuming that *o* was returned by `PySequence_Fast()`, *o* is not *NULL*, and that *i* is within bounds.

*PyObject*** **PySequence_Fast_ITEMS** (*PyObject* **o*)
Return the underlying array of PyObject pointers. Assumes that *o* was returned by `PySequence_Fast()` and *o* is not *NULL*.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

*PyObject** **PySequence_ITEM** (*PyObject* **o*, Py_ssize_t *i*)
Return value : New reference. Return the *i*th element of *o* or *NULL* on failure. Macro form of `PySequence_GetItem()` but without checking that `PySequence_Check()` on *o* is true and without adjustment for negative indices.

7.4 Mapping Protocol

See also `PyObject_GetItem()`, `PyObject_SetItem()` and `PyObject_DelItem()`.

int **PyMapping_Check** (*PyObject* **o*)
Return `1` if the object provides mapping protocol or supports slicing, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method since in general case it is impossible to determine what the type of keys it supports. This function always succeeds.

Py_ssize_t **PyMapping_Size** (*PyObject* **o*)
Py_ssize_t **PyMapping_Length** (*PyObject* **o*)
Returns the number of keys in object *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

*PyObject** **PyMapping_GetItemString** (*PyObject* **o*, const char **key*)
Return value : New reference. Return element of *o* corresponding to the string *key* or *NULL* on failure. This is the equivalent of the Python expression `o[key]`. See also `PyObject_GetItem()`.

int **PyMapping_SetItemString** (*PyObject* **o*, const char **key*, *PyObject* **v*)
Map the string *key* to the value *v* in object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `o[key] = v`. See also `PyObject_SetItem()`.

int **PyMapping_DelItem** (*PyObject* **o*, *PyObject* **key*)
Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`. This is an alias of `PyObject_DelItem()`.

int **PyMapping_DelItemString** (*PyObject* **o*, const char **key*)
Remove the mapping for the string *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int **PyMapping_HasKey** (*PyObject* *o, *PyObject* *key)

Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. This function always succeeds.

Note that exceptions which occur while calling the `__getitem__()` method will get suppressed. To get error reporting use `PyObject_GetItem()` instead.

int **PyMapping_HasKeyString** (*PyObject* *o, const char *key)

Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. This function always succeeds.

Note that exceptions which occur while calling the `__getitem__()` method and creating a temporary string object will get suppressed. To get error reporting use `PyMapping_GetItemString()` instead.

*PyObject** **PyMapping_Keys** (*PyObject* *o)

Return value : New reference. On success, return a list of the keys in object *o*. On failure, return *NULL*.

Modifié dans la version 3.7 : Previously, the function returned a list or a tuple.

*PyObject** **PyMapping_Values** (*PyObject* *o)

Return value : New reference. On success, return a list of the values in object *o*. On failure, return *NULL*.

Modifié dans la version 3.7 : Previously, the function returned a list or a tuple.

*PyObject** **PyMapping_Items** (*PyObject* *o)

Return value : New reference. On success, return a list of the items in object *o*, where each item is a tuple containing a key-value pair. On failure, return *NULL*.

Modifié dans la version 3.7 : Previously, the function returned a list or a tuple.

7.5 Protocole d'itération

Il existe deux fonctions dédiées à l'interaction avec les itérateurs.

int **PyIter_Check** (*PyObject* *o)

Renvoie vrai si l'objet *o* supporte le protocole d'itération.

*PyObject** **PyIter_Next** (*PyObject* *o)

Return value : New reference. Renvoie la valeur suivante d'une itération de *o*. L'objet doit être un itérateur (c'est à l'appelant de faire cette vérification). Renvoie *NULL* s'il n'y a plus de valeurs, sans déclarer d'exception. Renvoie *NULL* en déclarant une exception si une erreur survient lors de la récupération d'un élément.

Pour écrire une boucle itérant un itérateur, le code C devrait ressembler à :

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while (item = PyIter_Next(iterator)) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```

7.6 Protocole tampon

Certains objets Python enveloppent l'accès à un tableau de mémoire sous-jacente (nommée zone tampon ou simplement tampon, *buffer* en anglais). Les objets natifs `bytes` et `bytearray` en sont des exemples, ainsi que quelques types d'extension comme `array.array`. Les bibliothèques tierces peuvent définir leurs propres types à des fins spéciales, telles que le traitement d'image ou l'analyse numérique.

Alors que chacun de ces types a sa propre sémantique, ils partagent la caractéristique commune d'être soutenus par un tampon de mémoire important. Il est donc souhaitable, dans certains cas, d'accéder à cette mémoire directement sans l'étape intermédiaire de copie.

Python fournit une telle facilité au niveau du C sous la forme de *protocole tampon*. Ce protocole comporte deux aspects :

— du côté producteur, un type peut exporter une "interface tampon" qui permet aux objets de ce type d'exposer des informations concernant leur tampon sous-jacent. Cette interface est décrite dans la section *Buffer Object Structures* ;

— du côté consommateur, plusieurs moyens sont disponibles pour obtenir un pointeur vers les données sous-jacentes brutes d'un objet (par exemple un paramètre de méthode).

Des objets simples tels que `bytes` et `bytearray` exposent leur tampon sous-jacent dans un format orienté octet. D'autres formes sont possibles ; par exemple, les éléments exposés par un `array.array` peuvent être des valeurs multi-octets.

Un exemple de consommateur de l'interface tampon est la méthode `write()` des objets fichiers : tout objet qui peut exporter une série d'octets à travers l'interface tampon peut être écrit dans un fichier. Alors que `write()` n'a besoin que d'un accès lecture au contenu interne de l'objet qui lui est passé, d'autres méthodes telles que `readinto()` nécessitent un accès écriture au contenu de leur argument. L'interface *buffer* permet aux objets d'autoriser ou de rejeter sélectivement l'exportation de tampons en mode lecture-écriture et en mode lecture seule.

Un consommateur de l'interface tampon peut acquérir un tampon sur un objet cible de deux manières :

— appelez `PyObject_GetBuffer()` avec les paramètres appropriés ;

— appelez `PyArg_ParseTuple()` (ou l'un de ses fonctions sœurs) avec l'un des `y*`, `w*` ou `s*` *format codes*.

Dans les deux cas, `PyBuffer_Release()` doit être appelée quand le tampon n'est plus nécessaire. Ne pas le faire peut conduire à divers problèmes tels que des fuites de ressources.

7.6.1 La structure *buffer*

Les structures tampons (ou simplement les "tampons", *buffers* en anglais) sont utiles pour exposer les données binaires d'un autre objet au programmeur Python. Elles peuvent également être utilisées comme un mécanisme de découpage sans copie. En utilisant leur capacité à référencer un bloc de mémoire, il est possible d'exposer toutes les données au programmeur Python assez facilement. La mémoire peut être un grand tableau constant dans une extension C, il peut s'agir d'un bloc brut de mémoire à manipuler avant de passer à une bibliothèque de système d'exploitation ou être utilisé pour transmettre des données structurées dans son format natif en mémoire.

Contrairement à la plupart des types de données exposés par l'interpréteur Python, les tampons ne sont pas de simples pointeurs vers `PyObject` mais plutôt des structures C simples. Cela leur permet d'être créés et copiés très simplement.

lorsque vous avez besoin d'une enveloppe générique (*wrapper* en anglais) pour un tampon, un objet *memoryview* peut être créé.

For short instructions how to write an exporting object, see *Buffer Object Structures*. For obtaining a buffer, see `PyObject_GetBuffer()`.

Py_buffer

void ***buf**

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative `strides` the value may point to the end of the memory block.

For *contiguous* arrays, the value points to the beginning of the memory block.

void ***obj**

A new reference to the exporting object. The reference is owned by the consumer and automatically decremented and set to *NULL* by `PyBuffer_Release()`. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by `PyMemoryView_FromBuffer()` or `PyBuffer_FillInfo()` this field is *NULL*. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t **len**

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be `PyBUF_SIMPLE` or `PyBUF_WRITABLE`.

int **readonly**

An indicator of whether the buffer is read-only. This field is controlled by the `PyBUF_WRITABLE` flag.

Py_ssize_t **itemsize**

Item size in bytes of a single element. Same as the value of `struct.calcsize()` called on non-NULL `format` values.

Important exception : If a consumer requests a buffer without the `PyBUF_FORMAT` flag, `format` will be set to *NULL*, but `itemsize` still has the value for the original format.

If `shape` is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use `itemsize` to navigate the buffer.

If `shape` is *NULL* as a result of a `PyBUF_SIMPLE` or a `PyBUF_WRITABLE` request, the consumer must disregard `itemsize` and assume `itemsize == 1`.

const char ***format**

A *NUL* terminated string in `struct` module style syntax describing the contents of a single item. If this is *NULL*, `"B"` (unsigned bytes) is assumed.

This field is controlled by the `PyBUF_FORMAT` flag.

int **ndim**

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, `buf` points to a single item representing a scalar. In this case, `shape`, `strides` and `suboffsets` MUST be *NULL*.

The macro `PyBUF_MAX_NDIM` limits the maximum number of dimensions to 64. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions.

Py_ssize_t ***shape**

An array of `Py_ssize_t` of length `ndim` indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to `len`.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See *complex arrays* for further information.

The shape array is read-only for the consumer.

Py_ssize_t ***strides**

An array of `Py_ssize_t` of length *ndim* giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See *complex arrays* for further information.

The strides array is read-only for the consumer.

Py_ssize_t ***suboffsets**

An array of `Py_ssize_t` of length *ndim*. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be NULL (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See *complex arrays* for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

7.6.2 Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via *PyObject_GetBuffer()*. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All *Py_buffer* fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values : *obj*, *buf*, *len*, *itemsize*, *ndim*.

readonly, format

PyBUF_WRITABLE

Controls the *readonly* field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers.

PyBUF_FORMAT

Controls the *format* field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be *NULL*.

PyBUF_WRITABLE can be l'd to any of the flags in the next section. Since *PyBUF_SIMPLE* is defined as 0, *PyBUF_WRITABLE* can be used as a stand-alone flag to request a simple writable buffer.

PyBUF_FORMAT can be l'd to any of the flags except *PyBUF_SIMPLE*. The latter already implies format `B` (unsigned bytes).

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT	oui	oui	if needed
PyBUF_STRIDES	oui	oui	NULL
PyBUF_ND	oui	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

contiguity requests

C or Fortran *contiguity* can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	oui	oui	NULL	C
PyBUF_F_CONTIGUOUS	oui	oui	NULL	F
PyBUF_ANY_CONTIGUOUS	oui	oui	NULL	C or F
PyBUF_ND	oui	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call `PyBuffer_IsContiguous()` to determine contiguity.

Request	shape	strides	suboffsets	contig	readonly	format
PyBUF_FULL	oui	oui	if needed	U	0	oui
PyBUF_FULL_RO	oui	oui	if needed	U	1 or 0	oui
PyBUF_RECORDS	oui	oui	NULL	U	0	oui
PyBUF_RECORDS_RO	oui	oui	NULL	U	1 or 0	oui
PyBUF_STRIDED	oui	oui	NULL	U	0	NULL
PyBUF_STRIDED_RO	oui	oui	NULL	U	1 or 0	NULL
PyBUF_CONTIG	oui	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	oui	NULL	NULL	C	1 or 0	NULL

7.6.3 Complex arrays

NumPy-style : shape and strides

The logical structure of NumPy-style arrays is defined by `itemsize`, `ndim`, `shape` and `strides`.

If `ndim == 0`, the memory location pointed to by `buf` is interpreted as a scalar of size `itemsize`. In that case, both `shape` and `strides` are *NULL*.

If `strides` is *NULL*, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows :

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

As noted above, `buf` can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function :

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-style : shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays : `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of *buf*, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-NULL strides and suboffsets :

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.6.4 Fonctions relatives aux tampons

int **PyObject_CheckBuffer**(*PyObject* *obj*)

Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that `PyObject_GetBuffer()` will succeed. This function always succeeds.

int **PyObject_GetBuffer**(*PyObject* *exporter*, *Py_buffer* *view*, int *flags*)

Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `PyExc_BufferError`, set `view->obj` to *NULL* and return `-1`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See *Buffer Object Structures*).

Successful calls to `PyObject_GetBuffer()` must be paired with calls to `PyBuffer_Release()`, similar to `malloc()` and `free()`. Thus, after the consumer is done with the buffer, `PyBuffer_Release()` must be called exactly once.

void **PyBuffer_Release** (*Py_buffer *view*)

Release the buffer *view* and decrement the reference count for `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via `PyObject_GetBuffer()`.

Py_ssize_t **PyBuffer_SizeFromFormat** (const char *)

Return the implied `itemsize` from `format`. This function is not yet implemented.

int **PyBuffer_IsContiguous** (*Py_buffer *view*, char *order*)

Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) *contiguous* or either one (*order* is `'A'`). Return `0` otherwise. This function always succeeds.

void* **PyBuffer_GetPointer** (*Py_buffer *view*, Py_ssize_t **indices*)

Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

int **PyBuffer_ToContiguous** (void **buf*, *Py_buffer *src*, Py_ssize_t *len*, char *order*)

Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

void **PyBuffer_FillContiguousStrides** (int *ndims*, Py_ssize_t **shape*, Py_ssize_t **strides*, int *itemsize*, char *order*)

Fill the *strides* array with byte-strides of a *contiguous* (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo** (*Py_buffer *view*, *PyObject *exporter*, void **buf*, Py_ssize_t *len*, int *readonly*, int *flags*)

Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and `PyBUF_WRITABLE` is set in *flags*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `PyExc_BufferError`, set `view->obj` to *NULL* and return `-1`;

If this function is used as part of a *getbufferproc*, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be NULL.

7.7 Ancien Protocole Tampon

Obsolète depuis la version 3.0.

Ces fonctions faisaient partie de l'API de l'ancien protocole de tampons dans Python 2. Dans Python 3, ce protocole n'existe plus, mais les fonctions sont toujours exposées pour simplifier le portage de code Python 2.x. Elles se comportent comme une abstraction de compatibilité du *nouveau protocole de tampons*, mais sans vous donner de contrôle sur la durée de vie des ressources acquises lorsqu'un tampon est exporté.

Il est donc recommandé d'appeler `PyObject_GetBuffer()` (ou les *codes* `y*` ou `w*` à la famille de fonctions `PyArg_ParseTuple()`) pour obtenir une vue d'un tampon sur un objet, et `PyBuffer_Release()` lorsque la vue peut être libérée.

int **PyObject_AsCharBuffer** (*PyObject *obj*, const char ***buffer*, Py_ssize_t **buffer_len*)

Retourne un pointeur vers un emplacement de mémoire en lecture seule utilisable en tant qu'entrée basée sur des caractères. L'argument *obj* doit prendre en charge l'interface de tampon de caractère à segment unique. En cas de succès, retourne `0`, définit *buffer* à l'emplacement de la mémoire et *buffer_len* à la longueur de la mémoire tampon. Retourne `-1` et affecte une exception `TypeError` en cas d'erreur.

int **PyObject_AsReadBuffer**(*PyObject *obj*, const void ***buffer*, Py_ssize_t **buffer_len*)

Retourne un pointeur vers un emplacement de mémoire en lecture seule contenant des données arbitraires. L'argument *obj* doit prendre en charge l'interface de tampon lisible à segment unique. En cas de succès, retourne `0`, définit *buffer* à l'emplacement de la mémoire et *buffer_len* à la longueur de la mémoire tampon. Renvoie `-1` et affecte l'exception `TypeError` en cas d'erreur.

int **PyObject_CheckReadBuffer**(*PyObject *o*)

Retourne `1` si *o* prend en charge l'interface de mémoire tampon lisible à segment unique. Sinon, renvoie `0`. Cette fonction réussit toujours.

Notez que cette fonction tente d'obtenir et de libérer une mémoire tampon, et les exceptions qui se produisent lors de l'appel des fonctions correspondantes seront supprimées. Pour que les erreurs vous soient signalées, utilisez `PyObject_GetBuffer()` à la place.

int **PyObject_AsWriteBuffer**(*PyObject *obj*, void ***buffer*, Py_ssize_t **buffer_len*)

Retourne un pointeur vers un emplacement de mémoire accessible en écriture. L'argument *obj* doit prendre en charge l'interface de mémoire tampon de caractère à segment unique. En cas de succès, retourne `0`, définit *buffer* à l'emplacement de la mémoire et *buffer_len* à la longueur de la mémoire tampon. Renvoie `-1` et affecte l'exception `TypeError` en cas d'erreur.

CHAPITRE 8

Couche des objets concrets

Les fonctions de ce chapitre sont spécifiques à certains types d'objets Python. Leur donner un objet du mauvais type n'est pas une bonne idée, si vous recevez un objet d'un programme Python, et que vous n'êtes pas sûr qu'il soit du bon type, vous devez vérifier son type en premier. Par exemple, pour vérifier qu'un objet est un dictionnaire, utilisez `PyDict_Check()`. Ce chapitre est organisé comme un arbre généalogique de types d'objets Python.

> **Avertissement :** Tandis que les fonctions décrites dans ce chapitre vérifient avec soin le type des objets qui leur sont passés, beaucoup d'entre elles ne vérifient pas que *NULL* est passé au lieu d'un objet valide. Autoriser *NULL* à être passé peut provoquer des violations d'accès à la mémoire et ainsi terminer immédiatement l'interpréteur.

8.1 Objets fondamentaux

Cette section décrit les objets de type Python et l'objet singleton `None`.

8.1.1 Objets type

`PyTypeObject`

The C structure of the objects used to describe built-in types.

*PyObject** **`PyType_Type`**

This is the type object for type objects ; it is the same object as `type` in the Python layer.

int **`PyType_Check`** (*PyObject *o*)

Return true if the object *o* is a type object, including instances of types derived from the standard type object. Return false in all other cases.

int **`PyType_CheckExact`** (*PyObject *o*)

Return true if the object *o* is a type object, but not a subtype of the standard type object. Return false in all other cases.

unsigned int **PyType_ClearCache**()
Clear the internal lookup cache. Return the current version tag.

unsigned long **PyType_GetFlags**(*PyTypeObject* type*)
Return the `tp_flags` member of *type*. This function is primarily meant for use with *Py_LIMITED_API* ; the individual flag bits are guaranteed to be stable across Python releases, but access to `tp_flags` itself is not part of the limited API.

Nouveau dans la version 3.2.

Modifié dans la version 3.4 : The return type is now `unsigned long` rather than `long`.

void **PyType_Modified**(*PyTypeObject *type*)
Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

int **PyType_HasFeature**(*PyTypeObject *o*, int *feature*)
Return true if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

int **PyType_IS_GC**(*PyTypeObject *o*)
Return true if the type object includes support for the cycle detector ; this tests the type flag `Py_TPFLAGS_HAVE_GC`.

int **PyType_IsSubtype**(*PyTypeObject *a*, *PyTypeObject *b*)
Return true if *a* is a subtype of *b*.

This function only checks for actual subtypes, which means that `__subclasscheck__()` is not called on *b*. Call `PyObject_IsSubclass()` to do the same check that `issubclass()` would do.

*PyObject** **PyType_GenericAlloc**(*PyTypeObject *type*, Py_ssize_t *nitems*)
Return value : New reference. Generic handler for the `tp_alloc` slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to *NULL*.

*PyObject** **PyType_GenericNew**(*PyTypeObject *type*, *PyObject *args*, *PyObject *kwds*)
Return value : New reference. Generic handler for the `tp_new` slot of a type object. Create a new instance using the type's `tp_alloc` slot.

int **PyType_Ready**(*PyTypeObject *type*)
Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return `0` on success, or return `-1` and sets an exception on error.

*PyObject** **PyType_FromSpec**(PyType_Spec **spec*)
Return value : New reference. Creates and returns a heap type object from the *spec* passed to the function.

*PyObject** **PyType_FromSpecWithBases**(PyType_Spec **spec*, *PyObject *bases*)
Return value : New reference. Creates and returns a heap type object from the *spec*. In addition to that, the created heap type contains all types contained by the *bases* tuple as base types. This allows the caller to reference other heap types as base types.

Nouveau dans la version 3.3.

void* **PyType_GetSlot**(*PyTypeObject *type*, int *slot*)
Return the function pointer stored in the given slot. If the result is *NULL*, this indicates that either the slot is *NULL*, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

Nouveau dans la version 3.4.

8.1.2 L'objet `None`

Notez que le `PyTypeObject` de `None` n'est pas directement exposé via l'API Python/C. Puisque `None` est un singleton, tester son identité (en utilisant == en C) est suffisant. Il n'existe pas de fonction `PyNone_Check()` pour la même raison.

*PyObject** **Py_None**

L'objet Python `None`, exprimant l'absence de valeur. Cet objet n'a aucune méthode. Il doit être traité exactement comme les autres objets en terme de comptage de références.

Py_RETURN_NONE

Renvoie, de la bonne manière, `Py_None` depuis une fonction C (c'est à dire en incrémentant les références à `None` avant de le donner).

8.2 Objets numériques

8.2.1 Objets *Integer*

All integers are implemented as "long" integer objects of arbitrary size.

On error, most `PyLong_As*` APIs return `(return type)-1` which cannot be distinguished from a number. Use `PyErr_Occurred()` to disambiguate.

PyLongObject

This subtype of `PyObject` represents a Python integer object.

PyTypeObject **PyLong_Type**

This instance of `PyTypeObject` represents the Python integer type. This is the same object as `int` in the Python layer.

int **PyLong_Check** (*PyObject *p*)

Return true if its argument is a `PyLongObject` or a subtype of `PyLongObject`.

int **PyLong_CheckExact** (*PyObject *p*)

Return true if its argument is a `PyLongObject`, but not a subtype of `PyLongObject`.

*PyObject** **PyLong_FromLong** (long *v*)

Return value : New reference. Return a new `PyLongObject` object from *v*, or *NULL* on failure.

The current implementation keeps an array of integer objects for all integers between `-5` and `256`, when you create an int in that range you actually just get back a reference to the existing object. So it should be possible to change the value of `1`. I suspect the behaviour of Python in this case is undefined. :-)

*PyObject** **PyLong_FromUnsignedLong** (unsigned long *v*)

Return value : New reference. Return a new `PyLongObject` object from a C `unsigned long`, or *NULL* on failure.

*PyObject** **PyLong_FromSsize_t** (Py_ssize_t *v*)

Return value : New reference. Return a new `PyLongObject` object from a C `Py_ssize_t`, or *NULL* on failure.

*PyObject** **PyLong_FromSize_t** (size_t *v*)

Return value : New reference. Return a new `PyLongObject` object from a C `size_t`, or *NULL* on failure.

*PyObject** **PyLong_FromLongLong** (long long *v*)

Return value : New reference. Return a new `PyLongObject` object from a C `long long`, or *NULL* on failure.

*PyObject** **PyLong_FromUnsignedLongLong** (unsigned long long *v*)

Return value : New reference. Return a new `PyLongObject` object from a C `unsigned long long`, or *NULL* on failure.

*PyObject** **PyLong_FromDouble** (double *v*)

Return value : New reference. Return a new `PyLongObject` object from the integer part of *v*, or *NULL* on failure.

*PyObject** **PyLong_FromString** (const char **str*, char ***pend*, int *base*)

Return value : New reference. Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*. If *pend* is non-*NULL*, **pend* will point to the first character in *str* which follows the representation of the number. If *base* is `0`, *str* is interpreted using the integers definition ; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading spaces and single underscores after a base specifier and between digits are ignored. If there are no digits, `ValueError` will be raised.

*PyObject** **PyLong_FromUnicode** (*Py_UNICODE* **u*, Py_ssize_t *length*, int *base*)

Return value : New reference. Convert a sequence of Unicode digits to a Python integer value. The Unicode string is first encoded to a byte string using `PyUnicode_EncodeDecimal()` and then converted using `PyLong_FromString()`.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyLong_FromUnicodeObject()`.

*PyObject** **PyLong_FromUnicodeObject** (*PyObject* **u*, int *base*)

Return value : New reference. Convert a sequence of Unicode digits in the string *u* to a Python integer value. The Unicode string is first encoded to a byte string using `PyUnicode_EncodeDecimal()` and then converted using `PyLong_FromString()`.

Nouveau dans la version 3.3.

*PyObject** **PyLong_FromVoidPtr** (void **p*)

Return value : New reference. Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

long **PyLong_AsLong** (*PyObject* **obj*)

Return a C `long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__int__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

long **PyLong_AsLongAndOverflow** (*PyObject* **obj*, int **overflow*)

Return a C `long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__int__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1` ; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

long long **PyLong_AsLongLong** (*PyObject* **obj*)

Return a C `long long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__int__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

long long **PyLong_AsLongLongAndOverflow** (*PyObject* **obj*, int **overflow*)

Return a C `long long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__int__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than `PY_LLONG_MAX` or less than `PY_LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1` ; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Nouveau dans la version 3.2.

Py_ssize_t **PyLong_AsSsize_t** (*PyObject* **pylong*)

Return a C `Py_ssize_t` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `Py_ssize_t`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

unsigned long **PyLong_AsUnsignedLong**(*PyObject *pylong*)

Return a C `unsigned long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `unsigned long`.

Returns `(unsigned long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

size_t **PyLong_AsSize_t**(*PyObject *pylong*)

Return a C `size_t` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `size_t`.

Returns `(size_t)-1` on error. Use *PyErr_Occurred()* to disambiguate.

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject *pylong*)

Return a C `unsigned long long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for an `unsigned long long`.

Returns `(unsigned long long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

Modifié dans la version 3.1 : A negative *pylong* now raises `OverflowError`, not `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject *obj*)

Return a C `unsigned long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__int__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`.

Returns `(unsigned long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject *obj*)

Return a C `unsigned long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__int__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long long`, return the reduction of that value modulo `PY_ULLONG_MAX + 1`.

Returns `(unsigned long long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

double **PyLong_AsDouble**(*PyObject *pylong*)

Return a C `double` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `double`.

Returns `-1.0` on error. Use *PyErr_Occurred()* to disambiguate.

void* **PyLong_AsVoidPtr**(*PyObject *pylong*)

Convert a Python integer *pylong* to a C `void` pointer. If *pylong* cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable `void` pointer for values created with *PyLong_FromVoidPtr()*.

Returns *NULL* on error. Use *PyErr_Occurred()* to disambiguate.

8.2.2 Les objets booléens

Les booléens en Python sont implémentés comme une classe dérivée des entiers. Il y a seulement deux booléens, `Py_False` et `Py_True`. Comme tel, les fonctions de création de suppression ne s'appliquent pas aux booléens. Toutefois, les macros suivantes sont disponibles.

int **PyBool_Check**(*PyObject *o*)

Renvoie vrai si *o* est de type `PyBook_Type`.

*PyObject** **Py_False**

L'objet Python `False`. Cet objet n'a pas de méthodes. En ce qui concerne le comptage de référence, il doit être traité comme n'importe quel autre objet.

*PyObject** **Py_True**
L'objet Python `True`. Cet objet n'a pas de méthodes. En ce qui concerne le comptage de références, il doit être traité comme n'importe quel autre objet.

Py_RETURN_FALSE
Renvoie `Py_False` depuis une fonction tout en incrémentant son nombre de références.

Py_RETURN_TRUE
Renvoie `Py_True` depuis une fonction, en incrémentant son nombre de références.

*PyObject** **PyBool_FromLong**(long *v*)
Return value : New reference. Renvoie une nouvelle référence de `Py_True` ou `Py_False` en fonction de la valeur de *v*.

8.2.3 Objets représentant les nombres à virgule flottante

PyFloatObject
Ce sous-type de l'objet `PyObject` représente un nombre à virgule flottante en Python.

PyTypeObject **PyFloat_Type**
Cette instance de l'objet `PyTypeObject` représente le type nombre à virgule flottante en Python. C'est le même objet que la classe `float` de la couche Python.

int **PyFloat_Check**(*PyObject* *p*)
Renvoie vrai si l'argument est de type `PyFloatObject` ou un sous-type de `PyFloatObject`.

int **PyFloat_CheckExact**(*PyObject* *p*)
Renvoie vrai si l'argument est de type `PyFloatObject`, mais pas un sous-type de `PyFloatObject`.

*PyObject** **PyFloat_FromString**(*PyObject* *str*)
Return value : New reference. Crée un objet `PyFloatObject` à partir de la valeur de la chaîne de caractères *str*, ou *NULL* en cas d'échec.

*PyObject** **PyFloat_FromDouble**(double *v*)
Return value : New reference. Crée un objet `PyFloatObject` à partir de *v*, ou *NULL* en cas d'échec.

double **PyFloat_AsDouble**(*PyObject* *pyfloat*)
Renvoie une représentation du contenu d'un *pyfloat* sous la forme d'un `double` en C. Si le *pyfloat* n'est pas un nombre à virgule flottante mais contient une méthode `__float__()`, elle est d'abord appelée pour convertir le *pyfloat* en nombre à virgule flottante. Cette méthode renvoie `-1.0` en cas d'échec, il faut appeler `PyErr_Occurred()` pour vérifier les erreurs.

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat*)
Renvoie une représentation du contenu d'un *pyfloat* sous la forme d'un `double` en C, sans vérifier les erreurs.

*PyObject** **PyFloat_GetInfo**(void)
Return value : New reference. Renvoie une instance *structseq* qui contient les informations sur la précision et les valeurs minimales et maximales pour un nombre à virgule flottante. C'est une enveloppe autour du fichier d'entête `float.h`.

double **PyFloat_GetMax**()
Renvoie le nombre à virgule flottante fini maximal *DBL_MAX* sous la forme d'un `double` en C.

double **PyFloat_GetMin**()
Renvoie le nombre à virgule flottante minimal normalisé *DBL_MIN* sous la forme `double` en C.

int **PyFloat_ClearFreeList**()
Libère la mémoire de la *free list* des nombres à virgule flottante. Renvoie le nombre d'éléments qui n'ont pas pu être libérés.

8.2.4 Objets représentant des nombres complexes

Les nombres complexes Python sont implémentés comme deux types distincts, lorsqu'ils sont vus de l'API C : l'un est l'objet Python tel qu'il est vu par les programmes Python, et l'autre est une structure C qui représente la valeur exacte du nombre complexe. L'API fournit des fonctions pour travailler avec ces deux représentations.

Nombres complexes en tant que structures C

Les fonctions qui acceptent ces structures comme paramètres et les renvoient comme résultats le font en fonction de leur *valeur* au lieu de les dé-référencer en utilisant des pointeurs. C'est constant dans toute l'API.

Py_complex

Structure C représentant la valeur d'un nombre complexe Python. La majorité des fonctions qui traitent des nombres complexes utilisent cette structure en entrée ou en sortie, selon le cas. Elle est définie par :

```
typedef struct {
   double real;
   double imag;
} Py_complex;
```

Py_complex **_Py_c_sum** (*Py_complex left*, *Py_complex right*)

Renvoie la somme de deux nombres complexes, sous la forme d'un `Py_complex` en C.

Py_complex **_Py_c_diff** (*Py_complex left*, *Py_complex right*)

Renvoie la différence de deux nombres complexes, sous la forme d'un `Py_complex` en C.

Py_complex **_Py_c_neg** (*Py_complex complex*)

Renvoie l'opposé du nombre complexe *complex*, sous la forme d'un `Py_complex` en C.

Py_complex **_Py_c_prod** (*Py_complex left*, *Py_complex right*)

Renvoie le produit de deux nombres complexes, sous la forme d'un `Py_complex` en C.

Py_complex **_Py_c_quot** (*Py_complex dividend*, *Py_complex divisor*)

Renvoie le quotient de deux nombres complexes, sous la forme d'un `Py_complex` en C.

Si *divisor* est nul, cette méthode renvoie zéro et assigne `EDOM` à `errno`.

Py_complex **_Py_c_pow** (*Py_complex num*, *Py_complex exp*)

Renvoie *num* à la puissance *exp*, sous la forme d'un `Py_complex` en C.

Si *num* est nul et *exp* n'est pas un nombre réel positif, cette méthode renvoie zéro et assigne `EDOM` à `errno`.

Nombres complexes en tant qu'objets Python

PyComplexObject

Ce sous-type de l'objet `PyObject` représente un nombre complexe en Python.

PyTypeObject **PyComplex_Type**

Cette instance de `PyTypeObject` représente le type nombre complexe Python. C'est le même objet que la classe `complex` de la couche Python.

int **PyComplex_Check** (*PyObject *p*)

Renvoie vrai si l'argument est de type `PyComplexObject` ou un sous-type de `PyComplexObject`.

int **PyComplex_CheckExact** (*PyObject *p*)

Renvoie vrai si l'argument est de type `PyComplexObject`, mais pas un sous-type de `PyComplexObject`.

*PyObject** **PyComplex_FromCComplex** (*Py_complex v*)

Return value : New reference. Crée un nouveau nombre complexe à partir de la valeur d'un `Py_complex` en C.

*PyObject** **PyComplex_FromDoubles** (double *real*, double *imag*)
Return value : New reference. Renvoie un nouveau `PyComplexObject` à partir de *real* et de *imag*.

double **PyComplex_RealAsDouble** (*PyObject *op*)
Renvoie la partie réelle du nombre complexe *op* sous la forme d'un `double` en C.

double **PyComplex_ImagAsDouble** (*PyObject *op*)
Renvoie la partie imaginaire du nombre complexe *op* sous la forme d'un `double` en C.

Py_complex **PyComplex_AsCComplex** (*PyObject *op*)
Renvoie la valeur du nombre complexe *op* sous la forme d'un `Py_complex` en C.

Si *op* n'est pas un nombre complexe Python mais a une méthode `__complex__()`, cette méthode est d'abord appelée pour convertir *op* en nombre complexe Python. En cas d'échec, cette méthode renvoie `-1.0` en tant que nombre réel.

8.3 Objets séquences

Les opérations génériques sur les objets séquences ont été discutées dans le chapitre précédent. Cette section traite des genres spécifiques d'objets séquences qui sont intrinsèques au langage Python.

8.3.1 Objets *bytes*

These functions raise `TypeError` when expecting a bytes parameter and are called with a non-bytes parameter.

PyBytesObject
This subtype of `PyObject` represents a Python bytes object.

PyTypeObject **PyBytes_Type**
This instance of `PyTypeObject` represents the Python bytes type ; it is the same object as `bytes` in the Python layer.

int **PyBytes_Check** (*PyObject *o*)
Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type.

int **PyBytes_CheckExact** (*PyObject *o*)
Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type.

*PyObject** **PyBytes_FromString** (const char **v*)
Return value : New reference. Return a new bytes object with a copy of the string *v* as value on success, and *NULL* on failure. The parameter *v* must not be *NULL* ; it will not be checked.

*PyObject** **PyBytes_FromStringAndSize** (const char **v*, Py_ssize_t *len*)
Return value : New reference. Return a new bytes object with a copy of the string *v* as value and length *len* on success, and *NULL* on failure. If *v* is *NULL*, the contents of the bytes object are uninitialized.

*PyObject** **PyBytes_FromFormat** (const char **format*, ...)
Return value : New reference. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed :

Format Characters	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	*int*	A single byte, represented as a C int.
`%d`	*int*	Equivalent to `printf("%d")`. [1]
`%u`	`unsigned int`	Equivalent to `printf("%u")`. [1]
`%ld`	long	Equivalent to `printf("%ld")`. [1]
`%lu`	`unsigned long`	Equivalent to `printf("%lu")`. [1]
`%zd`	Py_ssize_t	Equivalent to `printf("%zd")`. [1]
`%zu`	size_t	Equivalent to `printf("%zu")`. [1]
`%i`	*int*	Equivalent to `printf("%i")`. [1]
`%x`	*int*	Equivalent to `printf("%x")`. [1]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

*PyObject** **PyBytes_FromFormatV** (const char **format*, va_list *vargs*)
Return value : New reference. Identical to `PyBytes_FromFormat()` except that it takes exactly two arguments.

*PyObject** **PyBytes_FromObject** (*PyObject* **o*)
Return value : New reference. Return the bytes representation of object *o* that implements the buffer protocol.

Py_ssize_t **PyBytes_Size** (*PyObject* **o*)
Return the length of the bytes in bytes object *o*.

Py_ssize_t **PyBytes_GET_SIZE** (*PyObject* **o*)
Macro form of `PyBytes_Size()` but without error checking.

char* **PyBytes_AsString** (*PyObject* **o*)
Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of `len(o) + 1` bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *o* is not a bytes object at all, `PyBytes_AsString()` returns *NULL* and raises `TypeError`.

char* **PyBytes_AS_STRING** (*PyObject* **string*)
Macro form of `PyBytes_AsString()` but without error checking.

int **PyBytes_AsStringAndSize** (*PyObject* **obj*, char ***buffer*, Py_ssize_t **length*)
Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*.

If *length* is *NULL*, the bytes object may not contain embedded null bytes ; if it does, the function returns `-1` and a `ValueError` is raised.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *obj* is not a bytes object at all, `PyBytes_AsStringAndSize()` returns `-1` and raises `TypeError`.

Modifié dans la version 3.5 : Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

void **PyBytes_Concat** (*PyObject* ***bytes*, *PyObject* **newpart*)
Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes* ; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old

1. For integer specifiers (d, u, ld, lu, zd, zu, i, x) : the 0-conversion flag has effect even when a precision is given.

reference to *bytes* will still be discarded and the value of **bytes* will be set to *NULL* ; the appropriate exception will be set.

void **PyBytes_ConcatAndDel** (*PyObject* ***bytes*, *PyObject* **newpart*)
Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version decrements the reference count of *newpart*.

int **_PyBytes_Resize** (*PyObject* ***bytes*, Py_ssize_t *newsize*)
A way to resize a bytes object even though it is "immutable". Only use this to build up a brand new bytes object ; don't use this if the bytes may already be known in other parts of the code. It is an error to call this function if the refcount on the input bytes object is not one. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned ; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to *NULL*, `MemoryError` is set, and `-1` is returned.

8.3.2 Objets tableau d'octets

PyByteArrayObject
Ce sous-type de `PyObject` représente un objet `bytearray` Python.

PyTypeObject **PyByteArray_Type**
Cette instance de `PyTypeObject` représente le type Python *bytearray*, c'est le même que `bytearray` côté Python.

Macros de vérification de type

int **PyByteArray_Check** (*PyObject* **o*)
Renvoie vrai si l'objet *o* est un `bytearray` ou une instance d'un sous-type du type `bytearray`.

int **PyByteArray_CheckExact** (*PyObject* **o*)
Renvoie vrai si l'objet *o* est un `bytearray`, mais pas une instance d'un sous-type du type `bytearray`.

Fonctions directes sur l'API

*PyObject** **PyByteArray_FromObject** (*PyObject* **o*)
Return value : New reference. Renvoie un nouvel objet *bytearray* depuis n'importe quel objet, *o*, qui implémente le *protocole buffer*.

*PyObject** **PyByteArray_FromStringAndSize** (const char **string*, Py_ssize_t *len*)
Return value : New reference. Crée un nouvel objet `bytearray` à partir d'un objet *string* et de sa longueur, *len*. En cas d'échec, *NULL* est renvoyé.

*PyObject** **PyByteArray_Concat** (*PyObject* **a*, *PyObject* **b*)
Return value : New reference. Concatène les `bytearrays` *a* et *b* et renvoie un nouveau `bytearray` avec le résultat.

Py_ssize_t **PyByteArray_Size** (*PyObject* **bytearray*)
Renvoie la taille de *bytearray* après vérification de la présence d'un pointeur *NULL*.

char* **PyByteArray_AsString** (*PyObject* **bytearray*)
Renvoie le contenu de *bytearray* sous forme d'un tableau de caractères, en vérifiant que ce n'est pas un pointeur *NULL*. Le tableau renvoyé se termine toujours par un octet *null*.

int **PyByteArray_Resize** (*PyObject* **bytearray*, Py_ssize_t *len*)
Redimensionne le tampon interne de *bytearray* à la taille *len*.

Macros

Ces macros sont taillées pour la vitesse d'exécution et ne vérifient pas les pointeurs.

char* **PyByteArray_AS_STRING** (*PyObject *bytearray*)
Version macro de `PyByteArray_AsString()`.

Py_ssize_t **PyByteArray_GET_SIZE** (*PyObject *bytearray*)
Version macro de `PyByteArray_Size()`.

8.3.3 Unicode Objects and Codecs

Unicode Objects

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536 ; otherwise, code points must be below 1114112 (which is the full Unicode range).

`Py_UNICODE*` and UTF-8 representations are created on demand and cached in the Unicode object. The `Py_UNICODE*` representation is deprecated and inefficient ; it should be avoided in performance- or memory-sensitive situations.

Due to the transition between the old APIs and the new APIs, Unicode objects can internally be in two states depending on how they were created :

— "canonical" Unicode objects are all objects created by a non-deprecated Unicode API. They use the most efficient representation allowed by the implementation.
— "legacy" Unicode objects have been created through one of the deprecated APIs (typically `PyUnicode_FromUnicode()`) and only bear the `Py_UNICODE*` representation ; you will have to call `PyUnicode_READY()` on them before calling any other API.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python :

Py_UCS4
Py_UCS2
Py_UCS1
These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use `Py_UCS4`.

Nouveau dans la version 3.3.

Py_UNICODE
This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

Modifié dans la version 3.3 : In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a "narrow" or "wide" Unicode version of Python at build time.

PyASCIIObject
PyCompactUnicodeObject
PyUnicodeObject
These subtypes of `PyObject` represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return `PyObject` pointers.

Nouveau dans la version 3.3.

PyTypeObject **PyUnicode_Type**
This instance of `PyTypeObject` represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects :

int **PyUnicode_Check**(*PyObject* **o*)
Return true if the object *o* is a Unicode object or an instance of a Unicode subtype.

int **PyUnicode_CheckExact**(*PyObject* **o*)
Return true if the object *o* is a Unicode object, but not an instance of a subtype.

int **PyUnicode_READY**(*PyObject* **o*)
Ensure the string object *o* is in the "canonical" representation. This is required before using any of the access macros described below.

Returns `0` on success and `-1` with an exception set on failure, which in particular happens if memory allocation fails.

Nouveau dans la version 3.3.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* **o*)
Return the length of the Unicode string, in code points. *o* has to be a Unicode object in the "canonical" representation (not checked).

Nouveau dans la version 3.3.

*Py_UCS1** **PyUnicode_1BYTE_DATA**(*PyObject* **o*)
*Py_UCS2** **PyUnicode_2BYTE_DATA**(*PyObject* **o*)
*Py_UCS4** **PyUnicode_4BYTE_DATA**(*PyObject* **o*)
Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size ; use `PyUnicode_KIND()` to select the right macro. Make sure `PyUnicode_READY()` has been called before accessing this.

Nouveau dans la version 3.3.

PyUnicode_WCHAR_KIND
PyUnicode_1BYTE_KIND
PyUnicode_2BYTE_KIND
PyUnicode_4BYTE_KIND
Return values of the `PyUnicode_KIND()` macro.

Nouveau dans la version 3.3.

int **PyUnicode_KIND**(*PyObject* **o*)
Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *o* has to be a Unicode object in the "canonical" representation (not checked).

Nouveau dans la version 3.3.

void* **PyUnicode_DATA**(*PyObject* **o*)
Return a void pointer to the raw Unicode buffer. *o* has to be a Unicode object in the "canonical" representation (not checked).

Nouveau dans la version 3.3.

void **PyUnicode_WRITE**(int *kind*, void **data*, Py_ssize_t *index*, *Py_UCS4* *value*)
Write into a canonical representation *data* (as obtained with `PyUnicode_DATA()`). This macro does not do any sanity checks and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other macro calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

Nouveau dans la version 3.3.

Py_UCS4 **PyUnicode_READ** (int *kind*, void **data*, Py_ssize_t *index*)
Read a code point from a canonical representation *data* (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

Nouveau dans la version 3.3.

Py_UCS4 **PyUnicode_READ_CHAR** (*PyObject* **o*, Py_ssize_t *index*)
Read a character from a Unicode object *o*, which must be in the "canonical" representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

Nouveau dans la version 3.3.

PyUnicode_MAX_CHAR_VALUE (*PyObject* **o*)
Return the maximum code point that is suitable for creating another string based on *o*, which must be in the "canonical" representation. This is always an approximation but more efficient than iterating over the string.

Nouveau dans la version 3.3.

int **PyUnicode_ClearFreeList** ()
Clear the free list. Return the total number of freed items.

Py_ssize_t **PyUnicode_GET_SIZE** (*PyObject* **o*)
Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units). *o* has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

Py_ssize_t **PyUnicode_GET_DATA_SIZE** (*PyObject* **o*)
Return the size of the deprecated `Py_UNICODE` representation in bytes. *o* has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

*Py_UNICODE** **PyUnicode_AS_UNICODE** (*PyObject* **o*)
const char* **PyUnicode_AS_DATA** (*PyObject* **o*)
Return a pointer to a `Py_UNICODE` representation of the object. The returned buffer is always terminated with an extra null code point. It may also contain embedded null code points, which would cause the string to be truncated when used in most C functions. The `AS_DATA` form casts the pointer to `const char *`. The *o* argument has to be a Unicode object (not checked).

Modifié dans la version 3.3 : This macro is now inefficient – because in many cases the `Py_UNICODE` representation does not exist and needs to be created – and can fail (return *NULL* with an exception set). Try to port the code to use the new `PyUnicode_nBYTE_DATA()` macros or use `PyUnicode_WRITE()` or `PyUnicode_READ()`.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style Unicode API, please migrate to using the `PyUnicode_nBYTE_DATA()` family of macros.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE** (*Py_UNICODE* *ch*)
Return `1` or `0` depending on whether *ch* is a whitespace character.

int **Py_UNICODE_ISLOWER** (*Py_UNICODE* *ch*)
Return `1` or `0` depending on whether *ch* is a lowercase character.

int **Py_UNICODE_ISUPPER** (*Py_UNICODE* *ch*)
Return `1` or `0` depending on whether *ch* is an uppercase character.

int **Py_UNICODE_ISTITLE** (*Py_UNICODE* *ch*)
Return `1` or `0` depending on whether *ch* is a titlecase character.

int **Py_UNICODE_ISLINEBREAK** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a linebreak character.

int **Py_UNICODE_ISDECIMAL** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a decimal character.

int **Py_UNICODE_ISDIGIT** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a numeric character.

int **Py_UNICODE_ISALPHA** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is an alphabetic character.

int **Py_UNICODE_ISALNUM** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int **Py_UNICODE_ISPRINTABLE** (*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions :

Py_UNICODE **Py_UNICODE_TOLOWER** (*Py_UNICODE ch*)
Return the character *ch* converted to lower case.

Obsolète depuis la version 3.3 : This function uses simple case mappings.

Py_UNICODE **Py_UNICODE_TOUPPER** (*Py_UNICODE ch*)
Return the character *ch* converted to upper case.

Obsolète depuis la version 3.3 : This function uses simple case mappings.

Py_UNICODE **Py_UNICODE_TOTITLE** (*Py_UNICODE ch*)
Return the character *ch* converted to title case.

Obsolète depuis la version 3.3 : This function uses simple case mappings.

int **Py_UNICODE_TODECIMAL** (*Py_UNICODE ch*)
Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This macro does not raise exceptions.

int **Py_UNICODE_TODIGIT** (*Py_UNICODE ch*)
Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This macro does not raise exceptions.

double **Py_UNICODE_TONUMERIC** (*Py_UNICODE ch*)
Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This macro does not raise exceptions.

These APIs can be used to work with surrogates :

Py_UNICODE_IS_SURROGATE (ch)
Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

Py_UNICODE_IS_HIGH_SURROGATE (ch)
Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

Py_UNICODE_IS_LOW_SURROGATE (ch)
Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

`Py_UNICODE_JOIN_SURROGATES` (high, low)
Join two surrogate characters and return a single Py_UCS4 value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair.

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs :

*PyObject** **`PyUnicode_New`** (Py_ssize_t *size*, *Py_UCS4* *maxchar*)
Return value : New reference. Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

Nouveau dans la version 3.3.

*PyObject** **`PyUnicode_FromKindAndData`** (int *kind*, const void **buffer*, Py_ssize_t *size*)
Return value : New reference. Create a new Unicode object with the given *kind* (possible values are `PyUnicode_1BYTE_KIND` etc., as returned by `PyUnicode_KIND()`). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

Nouveau dans la version 3.3.

*PyObject** **`PyUnicode_FromStringAndSize`** (const char **u*, Py_ssize_t *size*)
Return value : New reference. Create a Unicode object from the char buffer *u*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not *NULL*, the return value might be a shared object, i.e. modification of the data is not allowed.

If *u* is *NULL*, this function behaves like `PyUnicode_FromUnicode()` with the buffer set to *NULL*. This usage is deprecated in favor of `PyUnicode_New()`.

PyObject ***`PyUnicode_FromString`** (const char **u*)
Return value : New reference. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

*PyObject** **`PyUnicode_FromFormat`** (const char **format*, ...)
Return value : New reference. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed :

Format Characters	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	*int*	A single character, represented as a C int.
`%d`	*int*	Equivalent to `printf("%d")`.[1]
`%u`	`unsigned int`	Equivalent to `printf("%u")`.[1]
`%ld`	long	Equivalent to `printf("%ld")`.[1]
`%li`	long	Equivalent to `printf("%li")`.[1]
`%lu`	`unsigned long`	Equivalent to `printf("%lu")`.[1]
`%lld`	long long	Equivalent to `printf("%lld")`.[1]
`%lli`	long long	Equivalent to `printf("%lli")`.[1]
`%llu`	`unsigned long long`	Equivalent to `printf("%llu")`.[1]
`%zd`	Py_ssize_t	Equivalent to `printf("%zd")`.[1]
`%zi`	Py_ssize_t	Equivalent to `printf("%zi")`.[1]
`%zu`	size_t	Equivalent to `printf("%zu")`.[1]
`%i`	*int*	Equivalent to `printf("%i")`.[1]
`%x`	*int*	Equivalent to `printf("%x")`.[1]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`%A`	PyObject*	The result of calling `ascii()`.
`%U`	PyObject*	A Unicode object.
`%V`	PyObject*, const char*	A Unicode object (which may be *NULL*) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is *NULL*).
`%S`	PyObject*	The result of calling `PyObject_Str()`.
`%R`	PyObject*	The result of calling `PyObject_Repr()`.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Note : The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for `"%s"` and `"%V"` (if the `PyObject*` argument is NULL), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not NULL).

Modifié dans la version 3.2 : Support for `"%lld"` and `"%llu"` added.

Modifié dans la version 3.3 : Support for `"%li"`, `"%lli"` and `"%zi"` added.

Modifié dans la version 3.4 : Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"` added.

*PyObject** **`PyUnicode_FromFormatV`**(const char **format*, va_list *vargs*)

Return value : New reference. Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

*PyObject** **`PyUnicode_FromEncodedObject`**(*PyObject* **obj*, const char **encoding*, const char **errors*)

Return value : New reference. Decode an encoded object *obj* to a Unicode object.

`bytes`, `bytearray` and other *bytes-like objects* are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be *NULL* to have the interface use the default values (see *Built-in Codecs* for details).

1. For integer specifiers (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x) : the 0-conversion flag has effect even when a precision is given.

All other objects, including Unicode objects, cause a `TypeError` to be set.

The API returns *NULL* if there was an error. The caller is responsible for decref'ing the returned objects.

Py_ssize_t **PyUnicode_GetLength** (*PyObject *unicode*)

Return the length of the Unicode object, in code points.

Nouveau dans la version 3.3.

Py_ssize_t **PyUnicode_CopyCharacters** (*PyObject *to*, Py_ssize_t *to_start*, *PyObject *from*, Py_ssize_t *from_start*, Py_ssize_t *how_many*)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

Nouveau dans la version 3.3.

Py_ssize_t **PyUnicode_Fill** (*PyObject *unicode*, Py_ssize_t *start*, Py_ssize_t *length*, *Py_UCS4 fill_char*)

Fill a string with a character : write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return `-1` and raise an exception on error.

Nouveau dans la version 3.3.

int **PyUnicode_WriteChar** (*PyObject *unicode*, Py_ssize_t *index*, *Py_UCS4 character*)

Write a character to a string. The string must have been created through `PyUnicode_New()`. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

Nouveau dans la version 3.3.

Py_UCS4 **PyUnicode_ReadChar** (*PyObject *unicode*, Py_ssize_t *index*)

Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to the macro version `PyUnicode_READ_CHAR()`.

Nouveau dans la version 3.3.

*PyObject** **PyUnicode_Substring** (*PyObject *str*, Py_ssize_t *start*, Py_ssize_t *end*)

Return value : New reference. Return a substring of *str*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported.

Nouveau dans la version 3.3.

*Py_UCS4** **PyUnicode_AsUCS4** (*PyObject *u*, *Py_UCS4 *buffer*, Py_ssize_t *buflen*, int *copy_null*)

Copy the string *u* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns *NULL* and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *u*). *buffer* is returned on success.

Nouveau dans la version 3.3.

*Py_UCS4** **PyUnicode_AsUCS4Copy** (*PyObject *u*)

Copy the string *u* into a new UCS4 buffer that is allocated using `PyMem_Malloc()`. If this fails, *NULL* is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

Nouveau dans la version 3.3.

Deprecated Py_UNICODE APIs

Deprecated since version 3.3, will be removed in version 4.0.

These API functions are deprecated with the implementation of **PEP 393**. Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

*PyObject** **PyUnicode_FromUnicode** (const *Py_UNICODE* **u*, Py_ssize_t *size*)

Return value : New reference. Create a Unicode object from the Py_UNICODE buffer *u* of the given size. *u* may be *NULL* which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not *NULL*, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when *u* is *NULL*.

If the buffer is *NULL*, `PyUnicode_READY()` must be called once the string content has been filled before using any of the access macros such as `PyUnicode_KIND()`.

Please migrate to using `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()` or `PyUnicode_New()`.

*Py_UNICODE** **PyUnicode_AsUnicode** (*PyObject* **unicode*)

Return a read-only pointer to the Unicode object's internal `Py_UNICODE` buffer, or *NULL* on error. This will create the `Py_UNICODE*` representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting `Py_UNICODE` string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

*PyObject** **PyUnicode_TransformDecimalToASCII** (*Py_UNICODE* **s*, Py_ssize_t *size*)

Return value : New reference. Create a Unicode object by replacing all decimal digits in `Py_UNICODE` buffer of the given *size* by ASCII digits 0–9 according to their decimal value. Return *NULL* if an exception occurs.

*Py_UNICODE** **PyUnicode_AsUnicodeAndSize** (*PyObject* **unicode*, Py_ssize_t **size*)

Like `PyUnicode_AsUnicode()`, but also saves the `Py_UNICODE()` array length (excluding the extra null terminator) in *size*. Note that the resulting `Py_UNICODE*` string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Nouveau dans la version 3.3.

*Py_UNICODE** **PyUnicode_AsUnicodeCopy** (*PyObject* **unicode*)

Create a copy of a Unicode string ending with a null code point. Return *NULL* and raise a `MemoryError` exception on memory allocation failure, otherwise return a new allocated buffer (use `PyMem_Free()` to free the buffer). Note that the resulting `Py_UNICODE*` string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Nouveau dans la version 3.2.

Please migrate to using `PyUnicode_AsUCS4Copy()` or similar new APIs.

Py_ssize_t **PyUnicode_GetSize** (*PyObject* **unicode*)

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units).

Please migrate to using `PyUnicode_GetLength()`.

*PyObject** **PyUnicode_FromObject** (*PyObject* **obj*)

Return value : New reference. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return the reference with incremented refcount.

Objects other than Unicode or its subtypes will cause a `TypeError`.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

*PyObject** **PyUnicode_DecodeLocaleAndSize** (const char **str*, Py_ssize_t *len*, const char **errors*)

Return value : New reference. Decode a string from UTF-8 on Android, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a string from `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

Voir aussi :

The `Py_DecodeLocale()` function.

Nouveau dans la version 3.3.

Modifié dans la version 3.7 : The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_DecodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

*PyObject** **PyUnicode_DecodeLocale** (const char **str*, const char **errors*)

Return value : New reference. Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

Nouveau dans la version 3.3.

*PyObject** **PyUnicode_EncodeLocale** (*PyObject* **unicode*, const char **errors*)

Return value : New reference. Encode a Unicode object to UTF-8 on Android, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use `PyUnicode_EncodeFSDefault()` to encode a string to `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

Voir aussi :

The `Py_EncodeLocale()` function.

Nouveau dans la version 3.3.

Modifié dans la version 3.7 : The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_EncodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the error handler (**PEP 383** and **PEP 529**). To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function :

int **PyUnicode_FSConverter** (*PyObject** *obj*, void* *result*)

ParseTuple converter : encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using `PyUnicode_EncodeFSDefault()` ; `bytes` objects are output as-is. *result* must be a `PyBytesObject*` which must be released when it is no longer used.

Nouveau dans la version 3.1.

Modifié dans la version 3.6 : Accepte un *path-like object*.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSDecoder()` as the conversion function :

int **PyUnicode_FSDecoder** (*PyObject** *obj*, void* *result*)

ParseTuple converter : decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using `PyUnicode_DecodeFSDefaultAndSize()` ; `str` objects are output as-is. *result* must be a `PyUnicodeObject*` which must be released when it is no longer used.

Nouveau dans la version 3.2.

Modifié dans la version 3.6 : Accepte un *path-like object*.

*PyObject** **PyUnicode_DecodeFSDefaultAndSize** (const char **s*, Py_ssize_t *size*)

Return value : New reference. Decode a string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

Voir aussi :

The `Py_DecodeLocale()` function.

Modifié dans la version 3.6 : Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject** **PyUnicode_DecodeFSDefault** (const char **s*)

Return value : New reference. Decode a null-terminated string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use `PyUnicode_DecodeFSDefaultAndSize()` if you know the string length.

Modifié dans la version 3.6 : Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject** **PyUnicode_EncodeFSDefault** (*PyObject* **unicode*)

Return value : New reference. Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return `bytes`. Note that the resulting `bytes` object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

Voir aussi :

The `Py_EncodeLocale()` function.

Nouveau dans la version 3.2.

Modifié dans la version 3.6 : Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it :

*PyObject** **PyUnicode_FromWideChar** (const wchar_t **w*, Py_ssize_t *size*)

Return value : New reference. Create a Unicode object from the `wchar_t` buffer *w* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using wcslen. Return *NULL* on failure.

Py_ssize_t **PyUnicode_AsWideChar** (*PyObject* **unicode*, wchar_t **w*, Py_ssize_t *size*)

Copy the Unicode object contents into the `wchar_t` buffer *w*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error. Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t* **PyUnicode_AsWideCharString** (*PyObject* **unicode*, Py_ssize_t **size*)

Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not *NULL*, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is *NULL* and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_Alloc()` (use `PyMem_Free()` to free it) on success. On error, returns *NULL* and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

Nouveau dans la version 3.2.

Modifié dans la version 3.7 : Raises a `ValueError` if *size* is *NULL* and the `wchar_t*` string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to *NULL* causes the default encoding to be used which is ASCII. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only : on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes setlocale).

Error handling is set by errors which may also be set to *NULL* meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs :

*PyObject** **`PyUnicode_Decode`**(const char **s*, Py_ssize_t *size*, const char **encoding*, const char **errors*)
Return value : New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

*PyObject** **`PyUnicode_AsEncodedString`**(*PyObject* **unicode*, const char **encoding*, const char **errors*)
Return value : New reference. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

*PyObject** **`PyUnicode_Encode`**(const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **encoding*, const char **errors*)
Return value : New reference. Encode the `Py_UNICODE` buffer *s* of the given *size* and return a Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsEncodedString()`.

UTF-8 Codecs

These are the UTF-8 codec APIs :

*PyObject** **PyUnicode_DecodeUTF8** (const char **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF8Stateful** (const char **s*, Py_ssize_t *size*, const char **errors*, Py_ssize_t **consumed*)

Return value : New reference. If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF8()`. If *consumed* is not *NULL*, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF8String** (*PyObject* **unicode*)

Return value : New reference. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

const char* **PyUnicode_AsUTF8AndSize** (*PyObject* **unicode*, Py_ssize_t **size*)

Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be *NULL* ; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

In the case of an error, *NULL* is returned with an exception set and no *size* is stored.

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer.

Nouveau dans la version 3.3.

Modifié dans la version 3.7 : The return type is now `const char *` rather of `char *`.

const char* **PyUnicode_AsUTF8** (*PyObject* **unicode*)

As `PyUnicode_AsUTF8AndSize()`, but does not store the size.

Nouveau dans la version 3.3.

Modifié dans la version 3.7 : The return type is now `const char *` rather of `char *`.

*PyObject** **PyUnicode_EncodeUTF8** (const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Encode the `Py_UNICODE` buffer *s* of the given *size* using UTF-8 and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsUTF8String()`, `PyUnicode_AsUTF8AndSize()` or `PyUnicode_AsEncodedString()`.

UTF-32 Codecs

These are the UTF-32 codec APIs :

*PyObject** **PyUnicode_DecodeUTF32** (const char **s*, Py_ssize_t *size*, const char **errors*, int **byteorder*)

Return value : New reference. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-*NULL*) defines the error handling. It defaults to "strict".

If *byteorder* is non-*NULL*, the decoder starts decoding using the given byte order :

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is *NULL*, the codec starts in native order mode.

Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF32Stateful** (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value : New reference. If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF32()`. If *consumed* is not *NULL*, `PyUnicode_DecodeUTF32Stateful()` will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF32String** (*PyObject* *unicode)

Return value : New reference. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUTF32** (const *Py_UNICODE* *s, Py_ssize_t size, const char *errors, int byteorder)

Return value : New reference. Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in *s*. Output is written according to the following byte order :

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If *Py_UNICODE_WIDE* is not defined, surrogate pairs will be output as a single code point.

Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsUTF32String()` or `PyUnicode_AsEncodedString()`.

UTF-16 Codecs

These are the UTF-16 codec APIs :

*PyObject** **PyUnicode_DecodeUTF16** (const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value : New reference. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-*NULL*) defines the error handling. It defaults to "strict".

If *byteorder* is non-*NULL*, the decoder starts decoding using the given byte order :

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is *NULL*, the codec starts in native order mode.

Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF16Stateful** (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value : New reference. If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF16()`. If *consumed* is not *NULL*, `PyUnicode_DecodeUTF16Stateful()` will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF16String** (*PyObject* *unicode)

Return value : New reference. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUTF16**(const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*, int *byteorder*)

Return value : New reference. Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in *s*. Output is written according to the following byte order :

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If *Py_UNICODE_WIDE* is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as a UCS-2 character.

Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsUTF16String()` or `PyUnicode_AsEncodedString()`.

UTF-7 Codecs

These are the UTF-7 codec APIs :

*PyObject** **PyUnicode_DecodeUTF7**(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF7Stateful**(const char **s*, Py_ssize_t *size*, const char **errors*, Py_ssize_t **consumed*)

Return value : New reference. If *consumed* is *NULL*, behave like `PyUnicode_DecodeUTF7()`. If *consumed* is not *NULL*, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_EncodeUTF7**(const *Py_UNICODE* **s*, Py_ssize_t *size*, int *base64SetO*, int *base64WhiteSpace*, const char **errors*)

Return value : New reference. Encode the `Py_UNICODE` buffer of the given size using UTF-7 and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

If *base64SetO* is nonzero, "Set O" (punctuation that has no otherwise special meaning) will be encoded in base-64. If *base64WhiteSpace* is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python "utf-7" codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsEncodedString()`.

Unicode-Escape Codecs

These are the "Unicode Escape" codec APIs :

*PyObject** **PyUnicode_DecodeUnicodeEscape**(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_AsUnicodeEscapeString**(*PyObject* **unicode*)

Return value : New reference. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUnicodeEscape** (const *Py_UNICODE* *s, Py_ssize_t *size*)
Return value : New reference. Encode the `Py_UNICODE` buffer of the given *size* using Unicode-Escape and return a bytes object. Return *NULL* if an exception was raised by the codec.
Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsUnicodeEscapeString()`.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs :

*PyObject** **PyUnicode_DecodeRawUnicodeEscape** (const char *s, Py_ssize_t *size*, const char **errors*)
Return value : New reference. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_AsRawUnicodeEscapeString** (*PyObject* **unicode*)
Return value : New reference. Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeRawUnicodeEscape** (const *Py_UNICODE* *s, Py_ssize_t *size*)
Return value : New reference. Encode the `Py_UNICODE` buffer of the given *size* using Raw-Unicode-Escape and return a bytes object. Return *NULL* if an exception was raised by the codec.
Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsRawUnicodeEscapeString()` or `PyUnicode_AsEncodedString()`.

Latin-1 Codecs

These are the Latin-1 codec APIs : Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

*PyObject** **PyUnicode_DecodeLatin1** (const char *s, Py_ssize_t *size*, const char **errors*)
Return value : New reference. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_AsLatin1String** (*PyObject* **unicode*)
Return value : New reference. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeLatin1** (const *Py_UNICODE* *s, Py_ssize_t *size*, const char **errors*)
Return value : New reference. Encode the `Py_UNICODE` buffer of the given *size* using Latin-1 and return a Python bytes object. Return *NULL* if an exception was raised by the codec.
Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsLatin1String()` or `PyUnicode_AsEncodedString()`.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

*PyObject** **PyUnicode_DecodeASCII** (const char *s, Py_ssize_t *size*, const char **errors*)
Return value : New reference. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_AsASCIIString** (*PyObject* **unicode*)
Return value : New reference. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeASCII**(const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Encode the `Py_UNICODE` buffer of the given *size* using ASCII and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsASCIIString()` or `PyUnicode_AsEncodedString()`.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mapping to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface ; dictionaries and sequences work well.

These are the mapping codec APIs :

*PyObject** **PyUnicode_DecodeCharmap**(const char **data*, Py_ssize_t *size*, *PyObject* **mapping*, const char **errors*)

Return value : New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s* using the given *mapping* object. Return *NULL* if an exception was raised by the codec.

If *mapping* is *NULL*, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes – ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

*PyObject** **PyUnicode_AsCharmapString**(*PyObject* **unicode*, *PyObject* **mapping*)

Return value : New reference. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as "undefined mapping" and cause an error.

*PyObject** **PyUnicode_EncodeCharmap**(const *Py_UNICODE* **s*, Py_ssize_t *size*, *PyObject* **mapping*, const char **errors*)

Return value : New reference. Encode the `Py_UNICODE` buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsCharmapString()` or `PyUnicode_AsEncodedString()`.

The following codec API is special in that maps Unicode to Unicode.

*PyObject** **PyUnicode_Translate**(*PyObject* **unicode*, *PyObject* **mapping*, const char **errors*)

Return value : New reference. Translate a Unicode object using the given *mapping* object and return the resulting Unicode object. Return *NULL* if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None` (causing deletion of the character). Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

*PyObject** **PyUnicode_TranslateCharmap**(const *Py_UNICODE* **s*, Py_ssize_t *size*, *PyObject* **mapping*, const char **errors*)

Return value : New reference. Translate a `Py_UNICODE` buffer of the given *size* by applying a character *mapping* table to it and return the resulting Unicode object. Return *NULL* when an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_Translate()`. or *generic codec based API*

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

*PyObject** **PyUnicode_DecodeMBCS** (const char **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Create a Unicode object by decoding *size* bytes of the MBCS encoded string *s*. Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeMBCSStateful** (const char **s*, Py_ssize_t *size*, const char **errors*, Py_ssize_t **consumed*)

Return value : New reference. If *consumed* is *NULL*, behave like `PyUnicode_DecodeMBCS()`. If *consumed* is not *NULL*, `PyUnicode_DecodeMBCSStateful()` will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsMBCSString** (*PyObject* **unicode*)

Return value : New reference. Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeCodePage** (int *code_page*, *PyObject* **unicode*, const char **errors*)

Return value : New reference. Encode the Unicode object using the specified code page and return a Python bytes object. Return *NULL* if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

Nouveau dans la version 3.3.

*PyObject** **PyUnicode_EncodeMBCS** (const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*)

Return value : New reference. Encode the `Py_UNICODE` buffer of the given *size* using MBCS and return a Python bytes object. Return *NULL* if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0 : Part of the old-style `Py_UNICODE` API ; please migrate to using `PyUnicode_AsMBCSString()`, `PyUnicode_EncodeCodePage()` or `PyUnicode_AsEncodedString()`.

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return *NULL* or `-1` if an exception occurs.

*PyObject** **PyUnicode_Concat** (*PyObject* **left*, *PyObject* **right*)

Return value : New reference. Concat two strings giving a new Unicode string.

*PyObject** **PyUnicode_Split** (*PyObject* **s*, *PyObject* **sep*, Py_ssize_t *maxsplit*)

Return value : New reference. Split a string giving a list of Unicode strings. If *sep* is *NULL*, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

*PyObject** **PyUnicode_Splitlines** (*PyObject* **s*, int *keepend*)

Return value : New reference. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepend* is `0`, the Line break characters are not included in the resulting strings.

*PyObject** **PyUnicode_Translate** (*PyObject* **str*, *PyObject* **table*, const char **errors*)

Translate a string by applying a character mapping table to it and return the resulting Unicode object.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface ; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be *NULL* which indicates to use the default error handling.

*PyObject** **PyUnicode_Join** (*PyObject *separator*, *PyObject *seq*)

Return value : New reference. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch** (*PyObject *str*, *PyObject *substr*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)

Return `1` if *substr* matches `str[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **PyUnicode_Find** (*PyObject *str*, *PyObject *substr*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)

Return the first position of *substr* in `str[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match ; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar** (*PyObject *str*, *Py_UCS4 ch*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)

Return the first position of the character *ch* in `str[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match ; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Nouveau dans la version 3.3.

Modifié dans la version 3.7 : *start* and *end* are now adjusted to behave like `str[start:end]`.

Py_ssize_t **PyUnicode_Count** (*PyObject *str*, *PyObject *substr*, Py_ssize_t *start*, Py_ssize_t *end*)

Return the number of non-overlapping occurrences of *substr* in `str[start:end]`. Return `-1` if an error occurred.

*PyObject** **PyUnicode_Replace** (*PyObject *str*, *PyObject *substr*, *PyObject *replstr*, Py_ssize_t *maxcount*)

Return value : New reference. Replace at most *maxcount* occurrences of *substr* in *str* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **PyUnicode_Compare** (*PyObject *left*, *PyObject *right*)

Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call `PyErr_Occurred()` to check for errors.

int **PyUnicode_CompareWithASCIIString** (*PyObject *uni*, const char **string*)

Compare a Unicode object, *uni*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

*PyObject** **PyUnicode_RichCompare** (*PyObject *left*, *PyObject *right*, int *op*)

Return value : New reference. Rich compare two Unicode strings and return one of the following :

— `NULL` in case an exception was raised
— `Py_True` or `Py_False` for successful comparisons
— `Py_NotImplemented` in case the type combination is unknown

Possible values for *op* are `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE`.

*PyObject** **PyUnicode_Format** (*PyObject *format*, *PyObject *args*)

Return value : New reference. Return a new string object from *format* and *args* ; this is analogous to `format % args`.

int **PyUnicode_Contains** (*PyObject *container*, *PyObject *element*)

Check whether *element* is contained in *container* and return true or false accordingly.

element has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace** (*PyObject* ***string*)
Intern the argument **string* in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as **string*, it sets **string* to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves **string* alone and interns it (incrementing its reference count). (Clarification : even though there is a lot of talk about reference counts, think of this function as reference-count-neutral ; you own the object after the call if and only if you owned it before the call.)

*PyObject** **PyUnicode_InternFromString** (const char **v*)
Return value : New reference. A combination of `PyUnicode_FromString()` and `PyUnicode_InternInPlace()`, returning either a new Unicode string object that has been interned, or a new ("owned") reference to an earlier interned string object with the same value.

8.3.4 Tuple Objects

PyTupleObject
This subtype of `PyObject` represents a Python tuple object.

PyTypeObject **PyTuple_Type**
This instance of `PyTypeObject` represents the Python tuple type ; it is the same object as `tuple` in the Python layer.

int **PyTuple_Check** (*PyObject* **p*)
Return true if *p* is a tuple object or an instance of a subtype of the tuple type.

int **PyTuple_CheckExact** (*PyObject* **p*)
Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type.

*PyObject** **PyTuple_New** (Py_ssize_t *len*)
Return value : New reference. Return a new tuple object of size *len*, or *NULL* on failure.

*PyObject** **PyTuple_Pack** (Py_ssize_t *n*, ...)
Return value : New reference. Return a new tuple object of size *n*, or *NULL* on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Py_ssize_t **PyTuple_Size** (*PyObject* **p*)
Take a pointer to a tuple object, and return the size of that tuple.

Py_ssize_t **PyTuple_GET_SIZE** (*PyObject* **p*)
Return the size of the tuple *p*, which must be non-*NULL* and point to a tuple ; no error checking is performed.

*PyObject** **PyTuple_GetItem** (*PyObject* **p*, Py_ssize_t *pos*)
Return value : Borrowed reference. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is out of bounds, return *NULL* and sets an `IndexError` exception.

*PyObject** **PyTuple_GET_ITEM** (*PyObject* **p*, Py_ssize_t *pos*)
Return value : Borrowed reference. Like `PyTuple_GetItem()`, but does no checking of its arguments.

*PyObject** **PyTuple_GetSlice** (*PyObject* **p*, Py_ssize_t *low*, Py_ssize_t *high*)
Return value : New reference. Take a slice of the tuple pointed to by *p* from *low* to *high* and return it as a new tuple.

int **PyTuple_SetItem** (*PyObject* **p*, Py_ssize_t *pos*, *PyObject* **o*)
Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success.

Note : This function "steals" a reference to *o*.

void **PyTuple_SET_ITEM**(*PyObject* *p, Py_ssize_t *pos*, *PyObject* *o)
Like `PyTuple_SetItem()`, but does no error checking, and should *only* be used to fill in brand new tuples.

Note : This function "steals" a reference to *o*.

int **_PyTuple_Resize**(*PyObject* **p, Py_ssize_t *newsize*)
Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to *NULL*, and raises `MemoryError` or `SystemError`.

int **PyTuple_ClearFreeList**()
Clear the free list. Return the total number of freed items.

8.3.5 Struct Sequence Objects

Struct sequence objects are the C equivalent of `namedtuple()` objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

*PyTypeObject** **PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)
Return value : New reference. Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with `PyStructSequence_New()`.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)
Initializes a struct sequence type *type* from *desc* in place.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)
The same as `PyStructSequence_InitType`, but returns `0` on success and `-1` on failure.
Nouveau dans la version 3.4.

PyStructSequence_Desc
Contains the meta information of a struct sequence type to create.

Field	Type C	Signification
`name`	`const char *`	name of the struct sequence type
`doc`	`const char *`	pointer to docstring for the type or NULL to omit
`fields`	`PyStructSequence_Field *`	pointer to *NULL*-terminated array with field names of the new type
`n_in_sequence`	`int`	number of fields visible to the Python side (if used as tuple)

PyStructSequence_Field
Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as `PyObject*`. The index in the `fields` array of the `PyStructSequence_Desc` determines which field of the struct sequence is described.

Field	Type C	Signification
`name`	`const char *`	name for the field or *NULL* to end the list of named fields, set to PyStructSequence_UnnamedField to leave unnamed
`doc`	`const char *`	field docstring or *NULL* to omit

char* **PyStructSequence_UnnamedField**
Special value for a field name to leave it unnamed.

*PyObject** **PyStructSequence_New** (*PyTypeObject *type*)
Return value : New reference. Creates an instance of *type*, which must have been created with `PyStructSequence_NewType()`.

*PyObject** **PyStructSequence_GetItem** (*PyObject *p*, Py_ssize_t *pos*)
Return value : Borrowed reference. Return the object at position *pos* in the struct sequence pointed to by *p*. No bounds checking is performed.

*PyObject** **PyStructSequence_GET_ITEM** (*PyObject *p*, Py_ssize_t *pos*)
Return value : Borrowed reference. Macro equivalent of `PyStructSequence_GetItem()`.

void **PyStructSequence_SetItem** (*PyObject *p*, Py_ssize_t *pos*, *PyObject *o*)
Sets the field at index *pos* of the struct sequence *p* to value *o*. Like `PyTuple_SET_ITEM()`, this should only be used to fill in brand new instances.

Note : This function "steals" a reference to *o*.

void **PyStructSequence_SET_ITEM** (*PyObject *p*, Py_ssize_t *pos*, *PyObject *o*)
Macro equivalent of `PyStructSequence_SetItem()`.

Note : This function "steals" a reference to *o*.

8.3.6 List Objects

PyListObject
This subtype of `PyObject` represents a Python list object.

PyTypeObject **PyList_Type**
This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

int **PyList_Check** (*PyObject *p*)
Return true if *p* is a list object or an instance of a subtype of the list type.

int **PyList_CheckExact** (*PyObject *p*)
Return true if *p* is a list object, but not an instance of a subtype of the list type.

*PyObject** **PyList_New** (Py_ssize_t *len*)
Return value : New reference. Return a new list of length *len* on success, or *NULL* on failure.

Note : If *len* is greater than zero, the returned list object's items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()`.

Py_ssize_t **PyList_Size** (*PyObject *list*)
Return the length of the list object in *list* ; this is equivalent to `len(list)` on a list object.

Py_ssize_t **PyList_GET_SIZE** (*PyObject *list*)
Macro form of `PyList_Size()` without error checking.

*PyObject** **PyList_GetItem** (*PyObject *list*, Py_ssize_t *index*)
Return value : Borrowed reference. Return the object at position *index* in the list pointed to by *list*. The position must

be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (<0 or >=len(list)), return *NULL* and set an `IndexError` exception.

*PyObject** **PyList_GET_ITEM**(*PyObject *list*, Py_ssize_t *i*)
Return value : Borrowed reference. Macro form of `PyList_GetItem()` without error checking.

int **PyList_SetItem**(*PyObject *list*, Py_ssize_t *index*, *PyObject *item*)
Set the item at index *index* in list to *item*. Return `0` on success or `-1` on failure.

Note : This function "steals" a reference to *item* and discards a reference to an item already in the list at the affected position.

void **PyList_SET_ITEM**(*PyObject *list*, Py_ssize_t *i*, *PyObject *o*)
Macro form of `PyList_SetItem()` without error checking. This is normally only used to fill in new lists where there is no previous content.

Note : This macro "steals" a reference to *item*, and, unlike `PyList_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in *list* at position *i* will be leaked.

int **PyList_Insert**(*PyObject *list*, Py_ssize_t *index*, *PyObject *item*)
Insert the item *item* into list *list* in front of index *index*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.insert(index, item)`.

int **PyList_Append**(*PyObject *list*, *PyObject *item*)
Append the object *item* at the end of list *list*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.append(item)`.

*PyObject** **PyList_GetSlice**(*PyObject *list*, Py_ssize_t *low*, Py_ssize_t *high*)
Return value : New reference. Return a list of the objects in *list* containing the objects *between low* and *high*. Return *NULL* and set an exception if unsuccessful. Analogous to `list[low:high]`. Negative indices, as when slicing from Python, are not supported.

int **PyList_SetSlice**(*PyObject *list*, Py_ssize_t *low*, Py_ssize_t *high*, *PyObject *itemlist*)
Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be *NULL*, indicating the assignment of an empty list (slice deletion). Return `0` on success, `-1` on failure. Negative indices, as when slicing from Python, are not supported.

int **PyList_Sort**(*PyObject *list*)
Sort the items of *list* in place. Return `0` on success, `-1` on failure. This is equivalent to `list.sort()`.

int **PyList_Reverse**(*PyObject *list*)
Reverse the items of *list* in place. Return `0` on success, `-1` on failure. This is the equivalent of `list.reverse()`.

*PyObject** **PyList_AsTuple**(*PyObject *list*)
Return value : New reference. Return a new tuple object containing the contents of *list*; equivalent to `tuple(list)`.

int **PyList_ClearFreeList**()
Clear the free list. Return the total number of freed items.

Nouveau dans la version 3.3.

8.4 Objets conteneurs

8.4.1 Objets dictionnaires

PyDictObject
This subtype of *PyObject* represents a Python dictionary object.

PyTypeObject **PyDict_Type**
This instance of *PyTypeObject* represents the Python dictionary type. This is the same object as `dict` in the Python layer.

int **PyDict_Check**(*PyObject* *p)
Return true if *p* is a dict object or an instance of a subtype of the dict type.

int **PyDict_CheckExact**(*PyObject* *p)
Return true if *p* is a dict object, but not an instance of a subtype of the dict type.

*PyObject** **PyDict_New**()
Return value : New reference. Return a new empty dictionary, or *NULL* on failure.

*PyObject** **PyDictProxy_New**(*PyObject* *mapping)
Return value : New reference. Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

void **PyDict_Clear**(*PyObject* *p)
Empty an existing dictionary of all key-value pairs.

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)
Determine if dictionary *p* contains *key*. If an item in *p* is matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `key in p`.

*PyObject** **PyDict_Copy**(*PyObject* *p)
Return value : New reference. Return a new dictionary that contains the same key-value pairs as *p*.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)
Insert *value* into the dictionary *p* with a key of *key*. *key* must be *hashable* ; if it isn't, `TypeError` will be raised. Return `0` on success or `-1` on failure.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)
Insert *value* into the dictionary *p* using *key* as a key. *key* should be a `const char*`. The key object is created using `PyUnicode_FromString(key)`. Return `0` on success or `-1` on failure.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)
Remove the entry in dictionary *p* with key *key*. *key* must be hashable ; if it isn't, `TypeError` is raised. Return `0` on success or `-1` on failure.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)
Remove the entry in dictionary *p* which has a key specified by the string *key*. Return `0` on success or `-1` on failure.

*PyObject** **PyDict_GetItem**(*PyObject* *p, *PyObject* *key)
Return value : Borrowed reference. Return the object from dictionary *p* which has a key *key*. Return *NULL* if the key *key* is not present, but *without* setting an exception.

Note that exceptions which occur while calling `__hash__()` and `__eq__()` methods will get suppressed. To get error reporting use *PyDict_GetItemWithError()* instead.

*PyObject** **PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)
Return value : Borrowed reference. Variant of *PyDict_GetItem()* that does not suppress exceptions. Return *NULL* **with** an exception set if an exception occurred. Return *NULL* **without** an exception set if the key wasn't present.

*PyObject** **PyDict_GetItemString** (*PyObject* *p, const char *key)

Return value : Borrowed reference. This is the same as `PyDict_GetItem()`, but *key* is specified as a `const char*`, rather than a `PyObject*`.

Note that exceptions which occur while calling `__hash__()` and `__eq__()` methods and creating a temporary string object will get suppressed. To get error reporting use `PyDict_GetItemWithError()` instead.

*PyObject** **PyDict_SetDefault** (*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Return value : Borrowed reference. This is the same as the Python-level `dict.setdefault()`. If present, it returns the value corresponding to *key* from the dictionary *p*. If the key is not in the dict, it is inserted with value *defaultobj* and *defaultobj* is returned. This function evaluates the hash function of *key* only once, instead of evaluating it independently for the lookup and the insertion.

Nouveau dans la version 3.4.

*PyObject** **PyDict_Items** (*PyObject* *p)

Return value : New reference. Return a `PyListObject` containing all the items from the dictionary.

*PyObject** **PyDict_Keys** (*PyObject* *p)

Return value : New reference. Return a `PyListObject` containing all the keys from the dictionary.

*PyObject** **PyDict_Values** (*PyObject* *p)

Return value : New reference. Return a `PyListObject` containing all the values from the dictionary *p*.

Py_ssize_t **PyDict_Size** (*PyObject* *p)

Return the number of items in the dictionary. This is equivalent to `len(p)` on a dictionary.

int **PyDict_Next** (*PyObject* *p, Py_ssize_t *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration ; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be *NULL*. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

Par exemple :

```c
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example :

```c
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return -1;
    }
    Py_DECREF(o);
}
```

int **PyDict_Merge** (*PyObject *a*, *PyObject *b*, int *override*)
Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

int **PyDict_Update** (*PyObject *a*, *PyObject *b*)
This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no "keys" attribute. Return `0` on success or `-1` if an exception was raised.

int **PyDict_MergeFromSeq2** (*PyObject *a*, *PyObject *seq2*, int *override*)
Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value) :

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int **PyDict_ClearFreeList** ()
Clear the free list. Return the total number of freed items.

Nouveau dans la version 3.3.

8.4.2 Set Objects

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using the either the abstract object protocol (including `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()`, and `PyObject_GetIter()`) or the abstract number protocol (including `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()`, and `PyNumber_InPlaceXor()`).

PySetObject
This subtype of `PyObject` is used to hold the internal data for both `set` and `frozenset` objects. It is like a `PyDictObject` in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject **PySet_Type**
This is an instance of `PyTypeObject` representing the Python `set` type.

PyTypeObject **PyFrozenSet_Type**
This is an instance of `PyTypeObject` representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int **PySet_Check** (*PyObject* *p)
Return true if *p* is a `set` object or an instance of a subtype.

int **PyFrozenSet_Check** (*PyObject* *p)
Return true if *p* is a `frozenset` object or an instance of a subtype.

int **PyAnySet_Check** (*PyObject* *p)
Return true if *p* is a `set` object, a `frozenset` object, or an instance of a subtype.

int **PyAnySet_CheckExact** (*PyObject* *p)
Return true if *p* is a `set` object or a `frozenset` object but not an instance of a subtype.

int **PyFrozenSet_CheckExact** (*PyObject* *p)
Return true if *p* is a `frozenset` object but not an instance of a subtype.

*PyObject** **PySet_New** (*PyObject* *iterable)
Return value : New reference. Return a new `set` containing objects returned by the *iterable*. The *iterable* may be *NULL* to create a new empty set. Return the new set on success or *NULL* on failure. Raise `TypeError` if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

*PyObject** **PyFrozenSet_New** (*PyObject* *iterable)
Return value : New reference. Return a new `frozenset` containing objects returned by the *iterable*. The *iterable* may be *NULL* to create a new empty frozenset. Return the new set on success or *NULL* on failure. Raise `TypeError` if *iterable* is not actually iterable.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

Py_ssize_t **PySet_Size** (*PyObject* *anyset)
Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `PyExc_SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

Py_ssize_t **PySet_GET_SIZE** (*PyObject* *anyset)
Macro form of `PySet_Size()` without error checking.

int **PySet_Contains** (*PyObject* *anyset, *PyObject* *key)
Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the *key* is unhashable. Raise `PyExc_SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

int **PySet_Add** (*PyObject* *set, *PyObject* *key)
Add *key* to a `set` instance. Also works with `frozenset` instances (like `PyTuple_SetItem()` it can be used to fill-in the values of brand new frozensets before they are exposed to other code). Return `0` on success or `-1` on failure. Raise a `TypeError` if the *key* is unhashable. Raise a `MemoryError` if there is no room to grow. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

int **PySet_Discard** (*PyObject* *set, *PyObject* *key)
Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise `KeyError` for missing keys. Raise a `TypeError` if the *key* is unhashable. Unlike the Python `discard()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise `PyExc_SystemError` if *set* is not an instance of `set` or its subtype.

*PyObject** **PySet_Pop** (*PyObject* *set)
Return value : New reference. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return *NULL* on failure. Raise `KeyError` if the set is empty. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

int **PySet_Clear** (*PyObject* **set*)
Empty an existing set of all elements.

int **PySet_ClearFreeList** ()
Clear the free list. Return the total number of freed items.
Nouveau dans la version 3.3.

8.5 Objets fonctions

8.5.1 Objets fonctions

There are a few functions specific to Python functions.

PyFunctionObject
The C structure used for functions.

PyTypeObject **PyFunction_Type**
This is an instance of `PyTypeObject` and represents the Python function type. It is exposed to Python programmers as `types.FunctionType`.

int **PyFunction_Check** (*PyObject* **o*)
Return true if *o* is a function object (has type `PyFunction_Type`). The parameter must not be *NULL*.

*PyObject** **PyFunction_New** (*PyObject* **code*, *PyObject* **globals*)
Return value : New reference. Return a new function object associated with the code object *code*. *globals* must be a dictionary with the global variables accessible to the function.
The function's docstring and name are retrieved from the code object. *__module__* is retrieved from *globals*. The argument defaults, annotations and closure are set to *NULL*. *__qualname__* is set to the same value as the function's name.

*PyObject** **PyFunction_NewWithQualName** (*PyObject* **code*, *PyObject* **globals*, *PyObject* **qualname*)
Return value : New reference. As `PyFunction_New()`, but also allows setting the function object's `__qualname__` attribute. *qualname* should be a unicode object or NULL ; if NULL, the `__qualname__` attribute is set to the same value as its `__name__` attribute.
Nouveau dans la version 3.3.

*PyObject** **PyFunction_GetCode** (*PyObject* **op*)
Return value : Borrowed reference. Return the code object associated with the function object *op*.

*PyObject** **PyFunction_GetGlobals** (*PyObject* **op*)
Return value : Borrowed reference. Return the globals dictionary associated with the function object *op*.

*PyObject** **PyFunction_GetModule** (*PyObject* **op*)
Return value : Borrowed reference. Return the *__module__* attribute of the function object *op*. This is normally a string containing the module name, but can be set to any other object by Python code.

*PyObject** **PyFunction_GetDefaults** (*PyObject* **op*)
Return value : Borrowed reference. Return the argument default values of the function object *op*. This can be a tuple of arguments or *NULL*.

int **PyFunction_SetDefaults** (*PyObject* **op*, *PyObject* **defaults*)
Set the argument default values for the function object *op*. *defaults* must be *Py_None* or a tuple.
Raises `SystemError` and returns `-1` on failure.

*PyObject** **PyFunction_GetClosure** (*PyObject* **op*)
Return value : Borrowed reference. Return the closure associated with the function object *op*. This can be *NULL* or a tuple of cell objects.

int **PyFunction_SetClosure** (*PyObject *op*, *PyObject *closure*)
Set the closure associated with the function object *op*. *closure* must be *Py_None* or a tuple of cell objects.

Raises `SystemError` and returns `-1` on failure.

PyObject ***PyFunction_GetAnnotations** (*PyObject *op*)
Return value : Borrowed reference. Return the annotations of the function object *op*. This can be a mutable dictionary or *NULL*.

int **PyFunction_SetAnnotations** (*PyObject *op*, *PyObject *annotations*)
Set the annotations for the function object *op*. *annotations* must be a dictionary or *Py_None*.

Raises `SystemError` and returns `-1` on failure.

8.5.2 Instance Method Objects

An instance method is a wrapper for a `PyCFunction` and the new way to bind a `PyCFunction` to a class object. It replaces the former call `PyMethod_New(func, NULL, class)`.

PyTypeObject **PyInstanceMethod_Type**
This instance of `PyTypeObject` represents the Python instance method type. It is not exposed to Python programs.

int **PyInstanceMethod_Check** (*PyObject *o*)
Return true if *o* is an instance method object (has type `PyInstanceMethod_Type`). The parameter must not be *NULL*.

*PyObject** **PyInstanceMethod_New** (*PyObject *func*)
Return value : New reference. Return a new instance method object, with *func* being any callable object *func* is the function that will be called when the instance method is called.

*PyObject** **PyInstanceMethod_Function** (*PyObject *im*)
Return value : Borrowed reference. Return the function object associated with the instance method *im*.

*PyObject** **PyInstanceMethod_GET_FUNCTION** (*PyObject *im*)
Return value : Borrowed reference. Macro version of `PyInstanceMethod_Function()` which avoids error checking.

8.5.3 Objets méthode

Methods are bound function objects. Methods are always bound to an instance of a user-defined class. Unbound methods (methods bound to a class object) are no longer available.

PyTypeObject **PyMethod_Type**
This instance of `PyTypeObject` represents the Python method type. This is exposed to Python programs as `types.MethodType`.

int **PyMethod_Check** (*PyObject *o*)
Return true if *o* is a method object (has type `PyMethod_Type`). The parameter must not be *NULL*.

*PyObject** **PyMethod_New** (*PyObject *func*, *PyObject *self*)
Return value : New reference. Return a new method object, with *func* being any callable object and *self* the instance the method should be bound. *func* is the function that will be called when the method is called. *self* must not be *NULL*.

*PyObject** **PyMethod_Function** (*PyObject *meth*)
Return value : Borrowed reference. Return the function object associated with the method *meth*.

*PyObject** **PyMethod_GET_FUNCTION** (*PyObject *meth*)
Return value : Borrowed reference. Macro version of `PyMethod_Function()` which avoids error checking.

*PyObject** **PyMethod_Self** (*PyObject *meth*)
Return value : Borrowed reference. Return the instance associated with the method *meth*.

*PyObject** **PyMethod_GET_SELF** (*PyObject *meth*)
Return value : Borrowed reference. Macro version of `PyMethod_Self()` which avoids error checking.

int **PyMethod_ClearFreeList** ()
Clear the free list. Return the total number of freed items.

8.5.4 Objets Cellules

Les objets "Cellules" (*cell* en anglais) sont utilisés pour implémenter des variables référencées dans de multiples environnements. Pour chacune de ces variables, un objet cellule est créé pour stocker sa valeur ; les variables locales de chaque pile d'exécution qui référence cette valeur contiennent une référence sur les cellules des autres environnements qui utilisent aussi cette variable. Quand la valeur est accédée, la valeur de la cellule est utilisée, au lei de celle de l'objet cellule proprement dit. Ce dé-référencement de l'objet cellule requiert l'intervention du *bytecode* généré ; il n'est pas automatiquement dé-référencé quand il est accédé. Il est plausible que les objets cellules ne soit utilisés ailleurs.

PyCellObject
Structure C utilisée pour les objets cellules.

PyTypeObject **PyCell_Type**
Type objet correspondant aux objets cellules.

int **PyCell_Check** (ob)
Renvoie `True` si *ob* est un objet cellule ; *ob* ne doit pas être à *NULL*.

*PyObject** **PyCell_New** (*PyObject *ob*)
Return value : New reference. Crée et retourne un nouvel objet cellule contenant la valeur *ob*. Le paramètre peut être mis à *NULL*.

*PyObject** **PyCell_Get** (*PyObject *cell*)
Return value : New reference. Renvoie le contenu de la cellule *cell*.

*PyObject** **PyCell_GET** (*PyObject *cell*)
Return value : Borrowed reference. Renvoie le contenu de la cellule *cell*, mais sans vérifier si *cell* est non *NULL* et sans vérifier si c'est un objet cellule.

int **PyCell_Set** (*PyObject *cell*, *PyObject *value*)
Définit le contenu de l'objet cellule à *value*. Cela libère la référence à toute valeur de la cellule. *value* peut être fixé à *NULL*. *cell* ne doit pas être *NULL* ; si ce n'est pas un objet cellule, `-1` est renvoyé. Si c'est un objet cellule, renvoie `0`.

void **PyCell_SET** (*PyObject *cell*, *PyObject *value*)
Définit la valeur de l'objet cellule à *value*. Pas de comptage de références n'est ajusté et il n'y' a pas de contrôle effectué pour vérifier la sûreté ; *cell* doit être à non *NULL* et doit être un objet cellule.

8.5.5 Objets code

Les objets *Code* sont spécifiques à l'implémentation bas niveau de CPython. Chacun d'eux représente une partie de code exécutable, qui n'a pas encore été lié dans une fonction.

PyCodeObject
La structure C utilisée pour décrire les objets *Code*. Les attributs de cette structure sont sujets à changer à tout moment.

PyTypeObject **PyCode_Type**
C'est une instance de `PyTypeObject` représentant le type Python `code`.

int **PyCode_Check** (*PyObject* **co*)
Renvoie vrai si *co* est un objet `code`.

int **PyCode_GetNumFree** (*PyCodeObject* **co*)
Renvoie le nombre de variables libres dans *co*.

*PyCodeObject** **PyCode_New** (int *argcount*, int *kwonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* **code*, *PyObject* **consts*, *PyObject* **names*, *PyObject* **varnames*, *PyObject* **freevars*, *PyObject* **cellvars*, *PyObject* **filename*, *PyObject* **name*, int *firstlineno*, *PyObject* **lnotab*)
Return value : New reference. Renvoie un nouvel objet *code*. Si vous avez besoin d'un objet code factice pour créer une *frame*, utilisez plutôt `PyCode_NewEmpty()`. Appeler `PyCode_New()` peut vous lier directement à une version spécifique de Python, le *bytecode* étant sujet à modifications.

*PyCodeObject** **PyCode_NewEmpty** (const char **filename*, const char **funcname*, int *firstlineno*)
Return value : New reference. Renvoie un nouvel objet *code* avec le nom de fichier, le nom de fonction, et le numéro de première ligne donnés. Il n'est pas permis d'utiliser `exec()` ou `eval()` sur l'objet renvoyé.

8.6 Autres objets

8.6.1 Objets fichiers

Ces API sont une émulation minimale de l'API C Python 2 pour les objets fichier natifs, qui reposaient sur la gestion des entrées-sorties avec tampon (`FILE*`) de la bibliothèque standard C. En Python 3, les fichiers et les flux utilisent le nouveau module `io`, qui définit plusieurs couches au dessus des entrées/sorties bas niveau sans tampon du système d'exploitation. Les fonctions définies ci-dessous, écrites en C, encapsulent ces nouvelles APi pour les rendre plus faciles d'utilisation et sont majoritairement pensées pour signaler des erreurs internes dans l'interpréteur ; il est conseillé au code tiers d'utiliser les API de `io` à la place.

PyFile_FromFd (int *fd*, const char **name*, const char **mode*, int *buffering*, const char **encoding*, const char **errors*, const char **newline*, int *closefd*)
Return value : New reference. Crée un objet fichier Python à partir du descripteur d'un fichier déjà ouvert *fd*. Les arguments *name*, *encoding*, *errors* et *newline* peuvent être *NULL* pour utiliser ceux par défaut ; *buffering* peut être *-1* pour utiliser celui par défaut ; *name* est ignoré et gardé pour la rétro-compatibilité. Renvoie *NULL* en cas d'échec. Pour une description plus détaillée des arguments, merci de vous référer à la documentation de fonction `io.open()`.

Avertissement : Étant donné que les flux de données Python possèdent leur propre couche de tampon, les mélanger avec des descripteurs de fichiers du niveau du système d'exploitation peut produire des erreurs (comme par exemple un ordre des données inattendu).

Modifié dans la version 3.2 : ignore l'attribut *name*

int **PyObject_AsFileDescriptor** (*PyObject* **p*)
Renvoie un descripteur de fichier associé avec *p* comme un `int`. Si l'objet est un entier, sa valeur est renvoyée. Sinon, la méthode `fileno()` de l'objet est appelée si elle existe ; la méthode doit renvoyer un entier, qui est renvoyé en tant que valeur du descripteur. Affecte une exception et renvoie `-1` lors d'un échec.

*PyObject** **PyFile_GetLine** (*PyObject* **p*, int *n*)
Return value : New reference. Cette fonction, équivalente à `p.readline([n])`, lit une ligne de l'objet *p*. *p* peut être un objet fichier ou n'importe quel objet qui possède une méthode `readline()`. Si *n* vaut `0`, une seule ligne est lue, indépendamment de la taille de la ligne. Si *n* est plus grand que `0`, un maximum de *n* octets seront lus en provenance du fichier ; une ligne partielle peut être renvoyée. Dans les deux cas, une chaîne de caractères vide est renvoyée si la fin du fichier est atteinte immédiatement. Cependant, si *n* est plus petit que `0`, une ligne est lue indépendamment de sa taille, mais `EOFError` est levée si la fin du fichier est atteinte immédiatement.

int **PyFile_WriteObject** (*PyObject* *obj, *PyObject* *p, int *flags*)

Écrit l'objet *obj* dans l'objet fichier *p*. La seule option gérée pour *flags* est `Py_PRINT_RAW` ; si défini, l'attribut `str()` de l'objet est écrit à la place de l'attribut `repr()`. Retourne `0` en cas de succès ou `-1` en cas échec ; l'exception appropriée sera levée.

int **PyFile_WriteString** (const char *s, *PyObject* *p)

Écrit la chaîne de caractères *s* dans l'objet fichier *p*. Retourne `0` en cas de succès ou `-1` en cas d'échec ; l'exception appropriée sera mise en place.

8.6.2 Module Objects

PyTypeObject **PyModule_Type**

This instance of `PyTypeObject` represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int **PyModule_Check** (*PyObject* *p)

Return true if *p* is a module object, or a subtype of a module object.

int **PyModule_CheckExact** (*PyObject* *p)

Return true if *p* is a module object, but not a subtype of `PyModule_Type`.

*PyObject** **PyModule_NewObject** (*PyObject* *name)

Return value : New reference. Return a new module object with the `__name__` attribute set to *name*. The module's `__name__`, `__doc__`, `__package__`, and `__loader__` attributes are filled in (all but `__name__` are set to `None`) ; the caller is responsible for providing a `__file__` attribute.

Nouveau dans la version 3.3.

Modifié dans la version 3.4 : `__package__` and `__loader__` are set to `None`.

*PyObject** **PyModule_New** (const char *name)

Return value : New reference. Similar to `PyModule_NewObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

*PyObject** **PyModule_GetDict** (*PyObject* *module)

Return value : Borrowed reference. Return the dictionary object that implements *module*'s namespace ; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and *NULL* is returned.

It is recommended extensions use other `PyModule_*()` and `PyObject_*()` functions rather than directly manipulate a module's `__dict__`.

*PyObject** **PyModule_GetNameObject** (*PyObject* *module)

Return value : New reference. Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and *NULL* is returned.

Nouveau dans la version 3.3.

const char* **PyModule_GetName** (*PyObject* *module)

Similar to `PyModule_GetNameObject()` but return the name encoded to `'utf-8'`.

void* **PyModule_GetState** (*PyObject* *module)

Return the "state" of the module, that is, a pointer to the block of memory allocated at module creation time, or *NULL*. See `PyModuleDef.m_size`.

*PyModuleDef** **PyModule_GetDef** (*PyObject* *module)

Return a pointer to the `PyModuleDef` struct from which the module was created, or *NULL* if the module wasn't created from a definition.

*PyObject** **PyModule_GetFilenameObject** (*PyObject* *module)

Return value : New reference. Return the name of the file from which *module* was loaded using *module*'s `__file__` attribute. If this is not defined, or if it is not a unicode string, raise `SystemError` and return *NULL* ; otherwise return a reference to a Unicode object.

Nouveau dans la version 3.2.

const char* **PyModule_GetFilename** (*PyObject* **module*)

Similar to *PyModule_GetFilenameObject()* but return the filename encoded to 'utf-8'.

Obsolète depuis la version 3.2 : *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

Initializing C modules

Modules objects are usually created from extension modules (shared libraries which export an initialization function), or compiled-in modules (where the initialization function is added using *PyImport_AppendInittab()*). See building or extending-with-embedding for details.

The initialization function can either pass a module definition instance to *PyModule_Create()*, and return the resulting module object, or request "multi-phase initialization" by returning the definition struct itself.

PyModuleDef

The module definition struct, which holds all information needed to create a module object. There is usually only one statically initialized variable of this type for each module.

PyModuleDef_Base **m_base**

Always initialize this member to `PyModuleDef_HEAD_INIT`.

const char ***m_name**

Name for the new module.

const char ***m_doc**

Docstring for the module ; usually a docstring variable created with `PyDoc_STRVAR()` is used.

Py_ssize_t **m_size**

Module state may be kept in a per-module memory area that can be retrieved with *PyModule_GetState()*, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the `m_free` function has been called, if present.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state. Non-negative `m_size` is required for multi-phase initialization.

See **PEP 3121** for more details.

*PyMethodDef** **m_methods**

A pointer to a table of module-level functions, described by *PyMethodDef* values. Can be *NULL* if no functions are present.

*PyModuleDef_Slot** **m_slots**

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using single-phase initialization, *m_slots* must be *NULL*.

Modifié dans la version 3.5 : Prior to version 3.5, this member was always set to *NULL*, and was defined as :

inquiry **m_reload**

traverseproc **m_traverse**

A traversal function to call during GC traversal of the module object, or *NULL* if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return *NULL*), and before the *Py_mod_exec* function is executed.

inquiry **m_clear**

A clear function to call during GC clearing of the module object, or *NULL* if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return *NULL*), and before the *Py_mod_exec* function is executed.

freefunc **m_free**

A function to call during deallocation of the module object, or *NULL* if not needed. This function may be called before module state is allocated (`PyModule_GetState()` may return *NULL*), and before the `Py_mod_exec` function is executed.

Single-phase initialization

The module initialization function may create and return the module object directly. This is referred to as "single-phase initialization", and uses one of the following two module creation functions :

*PyObject** **PyModule_Create** (*PyModuleDef* **def*)

Return value : New reference. Create a new module object, given the definition in *def*. This behaves like `PyModule_Create2()` with *module_api_version* set to `PYTHON_API_VERSION`.

*PyObject** **PyModule_Create2** (*PyModuleDef* **def*, int *module_api_version*)

Return value : New reference. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Note : Most uses of this function should be using `PyModule_Create()` instead ; only use this if you are sure you need it.

Before it is returned from in the initialization function, the resulting module object is typically populated using functions like `PyModule_AddObject()`.

Multi-phase initialization

An alternate way to specify extensions is to request "multi-phase initialization". Extension modules created this way behave more like Python modules : the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons : if the *sys.modules* entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection – as with Python modules. By default, multiple modules created from the same definition should be independent : changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using `PyModule_GetState()`), or its contents (such as the module's `__dict__` or individual classes created with `PyType_FromSpec()`).

All modules created using multi-phase initialization are expected to support *sub-interpreters*. Making sure multiple modules are independent is typically enough to achieve this.

To request multi-phase initialization, the initialization function (PyInit_modulename) returns a `PyModuleDef` instance with non-empty `m_slots`. Before it is returned, the `PyModuleDef` instance must be initialized with the following function :

*PyObject** **PyModuleDef_Init** (*PyModuleDef* **def*)

Return value : Borrowed reference. Ensures a module definition is a properly initialized Python object that correctly reports its type and reference count.

Returns *def* cast to `PyObject*`, or *NULL* if an error occurred.

Nouveau dans la version 3.5.

The *m_slots* member of the module definition must point to an array of `PyModuleDef_Slot` structures :

PyModuleDef_Slot

int **slot**
A slot ID, chosen from the available values explained below.

void* **value**
Value of the slot, whose meaning depends on the slot ID.

Nouveau dans la version 3.5.

The *m_slots* array must be terminated by a slot with id 0.

The available slot types are :

Py_mod_create
Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature :

*PyObject** **create_module**(*PyObject* **spec*, *PyModuleDef* **def*)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return *NULL*.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using `PyModule_New()`. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of `PyModule_Type`. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-*NULL* `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Py_mod_exec
Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module : typically, this function adds classes and constants to the module. The signature of the function is :

int **exec_module**(*PyObject** *module*)

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the *m_slots* array.

See **PEP 489** for more details on multi-phase initialization.

Low-level module creation functions

The following functions are called under the hood when using multi-phase initialization. They can be used directly, for example when creating module objects dynamically. Note that both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

PyObject * **PyModule_FromDefAndSpec**(*PyModuleDef* **def*, *PyObject* **spec*)
Return value : New reference. Create a new module object, given the definition in *module* and the ModuleSpec *spec*. This behaves like `PyModule_FromDefAndSpec2()` with *module_api_version* set to `PYTHON_API_VERSION`.

Nouveau dans la version 3.5.

PyObject * **PyModule_FromDefAndSpec2**(*PyModuleDef* **def*, *PyObject* **spec*, int *module_api_version*)
Return value : New reference. Create a new module object, given the definition in *module* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Note : Most uses of this function should be using `PyModule_FromDefAndSpec()` instead ; only use this if you are sure you need it.

Nouveau dans la version 3.5.

int **PyModule_ExecDef** (*PyObject *module*, *PyModuleDef *def*)
Process any execution slots (`Py_mod_exec`) given in *def*.

Nouveau dans la version 3.5.

int **PyModule_SetDocString** (*PyObject *module*, const char **docstring*)
Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Nouveau dans la version 3.5.

int **PyModule_AddFunctions** (*PyObject *module*, *PyMethodDef *functions*)
Add the functions from the *NULL* terminated *functions* array to *module*. Refer to the `PyMethodDef` documentation for details on individual entries (due to the lack of a shared module namespace, module level "functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Nouveau dans la version 3.5.

Support functions

The module initialization function (if using single phase initialization) or a function called from a module execution slot (if using multi-phase initialization), can use the following functions to help initialize the module state :

int **PyModule_AddObject** (*PyObject *module*, const char **name*, *PyObject *value*)
Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function. This steals a reference to *value*. Return `-1` on error, `0` on success.

int **PyModule_AddIntConstant** (*PyObject *module*, const char **name*, long *value*)
Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` on error, `0` on success.

int **PyModule_AddStringConstant** (*PyObject *module*, const char **name*, const char **value*)
Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be *NULL*-terminated. Return `-1` on error, `0` on success.

int **PyModule_AddIntMacro** (*PyObject *module*, macro)
Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` on error, `0` on success.

int **PyModule_AddStringMacro** (*PyObject *module*, macro)
Add a string constant to *module*.

Module lookup

Single-phase initialization creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

*PyObject** **PyState_FindModule** (*PyModuleDef *def*)
Return value : Borrowed reference. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with `PyState_AddModule()` beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns *NULL*.

int **PyState_AddModule** (*PyObject *module*, *PyModuleDef *def*)
Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via `PyState_FindModule()`.

Only effective on modules created using single-phase initialization.

Nouveau dans la version 3.3.

int **PyState_RemoveModule** (*PyModuleDef *def*)
Removes the module object created from *def* from the interpreter state.

Nouveau dans la version 3.3.

8.6.3 Itérateurs

Python fournit deux itérateurs d'usage générique. Le premier est un itérateur de séquence, il fonctionne avec n'importe quelle séquence implémentant la méthode `__getitem__()`. Le second fonctionne avec un objet appelable et une valeur sentinelle, l'appelable permet d'obtenir chaque élément de la séquence, et l'itération se termine lorsque la sentinelle est atteinte.

PyTypeObject **PySeqIter_Type**
Type des itérateurs renvoyés par les fonctions `PySeqIter_New()` et la forme à un argument de la fonction native `iter()` pour les séquences natives.

int **PySeqIter_Check** (op)
Renvoie vrai si *op* est de type `PySeqIter_Type`.

*PyObject** **PySeqIter_New** (*PyObject *seq*)
Return value : New reference. Renvoie un itérateur sur la séquence *seq*. L'itération prend fin lorsque la séquence lève `IndexError` lors d'une tentative d'accès.

PyTypeObject **PyCallIter_Type**
Type de l'itérateur renvoyé par les fonctions `PyCallIter_New()` et `iter()` à deux arguments.

int **PyCallIter_Check** (op)
Renvoie vrai si *op* est de type `PyCallIter_Type`.

*PyObject** **PyCallIter_New** (*PyObject *callable*, *PyObject *sentinel*)
Return value : New reference. Renvoie un nouvel itérateur. Le premier paramètre, *callable*, peut être n'importe quel objet Python appelable sans aucun paramètre ; chaque appel doit renvoyer l'élément suivant de l'itération. Lorsque *callable* renvoie une valeur égale à *sentinel*, l'itération prend fin.

8.6.4 Les descripteurs

Les "Descripteurs" sont des objets décrivant des attributs pour un objet. Ils se trouvent dans le dictionnaire du type de l'objet.

PyTypeObject **PyProperty_Type**
L'objet *type* des descripteurs natifs.

*PyObject** **PyDescr_NewGetSet** (*PyTypeObject *type*, struct *PyGetSetDef *getset*)
Return value : New reference.

*PyObject** **PyDescr_NewMember** (*PyTypeObject* **type*, struct *PyMemberDef* **meth*)
Return value : New reference.

*PyObject** **PyDescr_NewMethod** (*PyTypeObject* **type*, struct *PyMethodDef* **meth*)
Return value : New reference.

*PyObject** **PyDescr_NewWrapper** (*PyTypeObject* **type*, struct wrapperbase **wrapper*, void **wrapped*)
Return value : New reference.

*PyObject** **PyDescr_NewClassMethod** (*PyTypeObject* **type*, *PyMethodDef* **method*)
Return value : New reference.

int **PyDescr_IsData** (*PyObject* **descr*)
Renvoie vrai si le descripteur *descr* décrit un attribut de donnée, ou faux s'il décrit une méthode. *descr* doit être un objet descripteur. Il n'y a pas de vérification d'erreur.

*PyObject** **PyWrapper_New** (*PyObject* *, *PyObject* *)
Return value : New reference.

8.6.5 Slice Objects

PyTypeObject **PySlice_Type**
The type object for slice objects. This is the same as `slice` in the Python layer.

int **PySlice_Check** (*PyObject* **ob*)
Return true if *ob* is a slice object ; *ob* must not be *NULL*.

*PyObject** **PySlice_New** (*PyObject* **start*, *PyObject* **stop*, *PyObject* **step*)
Return value : New reference. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be *NULL*, in which case the `None` will be used for the corresponding attribute. Return *NULL* if the new object could not be allocated.

int **PySlice_GetIndices** (*PyObject* **slice*, Py_ssize_t *length*, Py_ssize_t **start*, Py_ssize_t **stop*, Py_ssize_t **step*)
Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function.

Modifié dans la version 3.2 : The parameter type for the *slice* parameter was `PySliceObject*` before.

int **PySlice_GetIndicesEx** (*PyObject* **slice*, Py_ssize_t *length*, Py_ssize_t **start*, Py_ssize_t **stop*, Py_ssize_t **step*, Py_ssize_t **slicelength*)
Usable replacement for `PySlice_GetIndices()`. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns `0` on success and `-1` on error with exception set.

Note : This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of `PySlice_Unpack()` and `PySlice_AdjustIndices()` where

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {
    // return error
}
```

is replaced by

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

Modifié dans la version 3.2 : The parameter type for the *slice* parameter was `PySliceObject*` before.

Modifié dans la version 3.6.1 : If `Py_LIMITED_API` is not set or set to the value between `0x03050400` and `0x03060000` (not including) or `0x03060100` or higher `PySlice_GetIndicesEx()` is implemented as a macro using `PySlice_Unpack()` and `PySlice_AdjustIndices()`. Arguments *start*, *stop* and *step* are evaluated more than once.

Obsolète depuis la version 3.6.1 : If `Py_LIMITED_API` is set to the value less than `0x03050400` or between `0x03060000` and `0x03060100` (not including) `PySlice_GetIndicesEx()` is a deprecated function.

int **PySlice_Unpack** (*PyObject* **slice*, Py_ssize_t **start*, Py_ssize_t **stop*, Py_ssize_t **step*)
Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than `PY_SSIZE_T_MAX` to `PY_SSIZE_T_MAX`, silently boost the start and stop values less than `PY_SSIZE_T_MIN` to `PY_SSIZE_T_MIN`, and silently boost the step values less than `-PY_SSIZE_T_MAX` to `-PY_SSIZE_T_MAX`.

Return `-1` on error, `0` on success.

Nouveau dans la version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices** (Py_ssize_t *length*, Py_ssize_t **start*, Py_ssize_t **stop*, Py_ssize_t *step*)
Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn't call Python code.

Nouveau dans la version 3.6.1.

8.6.6 Ellipsis Object

PyObject ***Py_Ellipsis**
The Python `Ellipsis` object. This object has no methods. It needs to be treated just like any other object with respect to reference counts. Like `Py_None` it is a singleton object.

8.6.7 Objets de type MemoryView

A `memoryview` object exposes the C level *buffer interface* as a Python object which can then be passed around like any other object.

PyObject ***PyMemoryView_FromObject** (*PyObject* **obj*)
Return value : New reference. Create a memoryview object from an object that provides the buffer interface. If *obj* supports writable buffer exports, the memoryview object will be read/write, otherwise it may be either read-only or read/write at the discretion of the exporter.

PyObject ***PyMemoryView_FromMemory** (char **mem*, Py_ssize_t *size*, int *flags*)
Return value : New reference. Create a memoryview object using *mem* as the underlying buffer. *flags* can be one of `PyBUF_READ` or `PyBUF_WRITE`.

Nouveau dans la version 3.3.

PyObject ***PyMemoryView_FromBuffer** (*Py_buffer* **view*)
Return value : New reference. Create a memoryview object wrapping the given buffer structure *view*. For simple byte buffers, `PyMemoryView_FromMemory()` is the preferred function.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* **obj*, int *buffertype*, char *order*)
Return value : New reference. Create a memoryview object to a *contiguous* chunk of memory (in either 'C' or 'F'ortran *order*) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise, a copy is made and the memoryview points to a new bytes object.

int **PyMemoryView_Check**(*PyObject* **obj*)
Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of `memoryview`.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* **mview*)
Return a pointer to the memoryview's private copy of the exporter's buffer. *mview* **must** be a memoryview instance ; this macro doesn't check its type, you must do it yourself or you will risk crashes.

Py_buffer ***PyMemoryView_GET_BASE**(*PyObject* **mview*)
Return either a pointer to the exporting object that the memoryview is based on or *NULL* if the memoryview has been created by one of the functions `PyMemoryView_FromMemory()` or `PyMemoryView_FromBuffer()`. *mview* **must** be a memoryview instance.

8.6.8 Objets à références faibles

Python gère les *références faibles* comme des objets de première classe. Il existe deux types d'objets spécifiques qui implémentent directement les références faibles. Le premier est un objet de référence simple, et le second agit autant que possible comme un mandataire vers l'objet original.

int **PyWeakref_Check**(ob)
Renvoie vrai si *ob* est soit une référence, soit un objet proxy.

int **PyWeakref_CheckRef**(ob)
Retourne vrai si *ob* est un objet référence.

int **PyWeakref_CheckProxy**(ob)
Retourne vrai si *ob* est un objet proxy

*PyObject** **PyWeakref_NewRef**(*PyObject* **ob*, *PyObject* **callback*)
Return value : New reference. Retourne un objet de référence faible pour l'objet *ob*. Elle renvoie toujours une nouvelle référence, mais cela ne signifie pas qu'un nouvel objet est créé ; un objet référence existant peut être renvoyé. Le second paramètre, *callback*, peut être un objet appelable qui reçoit une notification lorsque *ob* est collecté par le ramasse-miette (*garbage collected* en anglais) ; il doit accepter un paramètre unique, qui est l'objet référence faible lui-même. *callback* peut aussi être positionné à `None` ou à *NULL*. Si *ob* n'est pas un objet faiblement référençable, ou si *callback* n'est pas appelable, `None`` ou *NULL*, ceci retourne *NULL* et lève une `TypeError`.

*PyObject** **PyWeakref_NewProxy**(*PyObject* **ob*, *PyObject* **callback*)
Return value : New reference. Retourne un objet mandataire à référence faible pour l'objet *ob*. Ceci renvoie toujours une nouvelle référence, mais ne garantit pas la création d'un nouvel objet ; un objet proxy existant peut être retourné. Le second paramètre, *callback*, peut être un objet appelable qui reçoit une notification lorsque *ob* est collecté ; il doit accepter un seul paramètre, qui sera l'objet de référence faible lui-même. *callback* peut aussi être `None` ou *NULLL*. Si *ob* n'est pas un objet faiblement référençable, ou si *callback* n'est pas appelable, `None`` ou *NULL*, ceci renvoie *NULL* et lève une `TypeError`.

*PyObject** **PyWeakref_GetObject**(*PyObject* **ref*)
Return value : Borrowed reference. Retourne l'objet référencé à partir d'une référence faible, *ref*. Si le référence n'existe plus, alors l'objet renvoie `Py_None`.

Note : Cette fonction renvoie une **référence empruntée** à l'objet référencé. Cela signifie que vous devez toujours appeler `Py_INCREF()` sur l'objet sauf si vous savez qu'il ne peut pas être détruit tant que vous l'utilisez encore.

*PyObject** **PyWeakref_GET_OBJECT**(*PyObject *ref*)
Return value : Borrowed reference. Similaire à `PyWeakref_GetObject()`, mais implémenté comme une macro qui ne vérifie pas les erreurs.

8.6.9 Capsules

Reportez-vous à using-capsules pour plus d'informations sur l'utilisation de ces objets.

Nouveau dans la version 3.1.

PyCapsule
This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

PyCapsule_Destructor
The type of a destructor callback for a capsule. Defined as :

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See `PyCapsule_New()` for the semantics of PyCapsule_Destructor callbacks.

int **PyCapsule_CheckExact**(*PyObject *p*)
Return true if its argument is a `PyCapsule`.

*PyObject** **PyCapsule_New**(void **pointer*, const char **name*, *PyCapsule_Destructor destructor*)
Return value : New reference. Create a `PyCapsule` encapsulating the *pointer*. The *pointer* argument may not be *NULL*.

On failure, set an exception and return *NULL*.

The *name* string may either be *NULL* or a pointer to a valid C string. If non-*NULL*, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not *NULL*, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using `PyCapsule_Import()`.

void* **PyCapsule_GetPointer**(*PyObject *capsule*, const char **name*)
Retrieve the *pointer* stored in the capsule. On failure, set an exception and return *NULL*.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is *NULL*, the *name* passed in must also be *NULL*. Python uses the C function `strcmp()` to compare capsule names.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject *capsule*)
Return the current destructor stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* destructor. This makes a *NULL* return code somewhat ambiguous ; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void* **PyCapsule_GetContext**(*PyObject *capsule*)
Return the current context stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* context. This makes a *NULL* return code somewhat ambiguous ; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

const char* **PyCapsule_GetName**(*PyObject *capsule*)
Return the current name stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* name. This makes a *NULL* return code somewhat ambiguous ; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void* **PyCapsule_Import** (const char **name*, int *no_block*)
Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly. If *no_block* is true, import the module without blocking (using `PyImport_ImportModuleNoBlock()`). If *no_block* is false, import the module conventionally (using `PyImport_ImportModule()`).

Return the capsule's internal *pointer* on success. On failure, set an exception and return *NULL*.

int **PyCapsule_IsValid** (*PyObject* **capsule*, const char **name*)
Determines whether or not *capsule* is a valid capsule. A valid capsule is non-*NULL*, passes `PyCapsule_CheckExact()`, has a non-*NULL* pointer stored in it, and its internal name matches the *name* parameter. (See `PyCapsule_GetPointer()` for information on how capsule names are compared.)

In other words, if `PyCapsule_IsValid()` returns a true value, calls to any of the accessors (any function starting with `PyCapsule_Get()`) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return `0` otherwise. This function will not fail.

int **PyCapsule_SetContext** (*PyObject* **capsule*, void **context*)
Set the context pointer inside *capsule* to *context*.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetDestructor** (*PyObject* **capsule*, *PyCapsule_Destructor* *destructor*)
Set the destructor inside *capsule* to *destructor*.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetName** (*PyObject* **capsule*, const char **name*)
Set the name inside *capsule* to *name*. If non-*NULL*, the name must outlive the capsule. If the previous *name* stored in the capsule was not *NULL*, no attempt is made to free it.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetPointer** (*PyObject* **capsule*, void **pointer*)
Set the void pointer inside *capsule* to *pointer*. The pointer may not be *NULL*.

Return `0` on success. Return nonzero and set an exception on failure.

8.6.10 Objets générateur

Python utilise des objets générateurs pour implémenter les itérations de générateurs. Ils sont normalement crées en itérant sur une fonction donnant des valeurs via `yield`, au lieu d'appeler explicitement `PyGen_New()` ou `PyGen_NewWithQualName()`.

PyGenObject
La structure C utilisée pour les objets générateurs.

PyTypeObject **PyGen_Type**
Le type objet correspondant aux objets générateurs.

int **PyGen_Check** (*PyObject* **ob*)
Renvoie `True` si *ob* est un objet générateur. *ob* ne doit pas être *NULL*.

int **PyGen_CheckExact** (*PyObject* **ob*)
Renvoie `True` si le type de *ob* est *PyGen_Type*. *ob* ne doit pas être *NULL*.

*PyObject** **PyGen_New** (*PyFrameObject* **frame*)
Return value : New reference. Crée et renvoie un nouvel objet générateur basé sur l'objet *frame*. Une référence à *frame* est volée par cette fonction. L'argument ne doit pas être *NULL*.

*PyObject** **PyGen_NewWithQualName** (*PyFrameObject* **frame*, *PyObject* **name*, *PyObject* **qualname*)
Return value : New reference. Crée et renvoie un nouvel objet générateur basé sur l'objet *frame*, avec `__name__` et

`__qualname__` valant *name* et *qualname*. Une référence à *frame* est volée par cette fonction. L'argument *frame* ne doit pas être *NULL*.

8.6.11 Objets coroutines

Nouveau dans la version 3.5.

Les objets coroutines sont les objets renvoyés par les fonctions déclarées avec le mot clef `async`.

PyCoroObject
La structure C utilisée pour les objets coroutine.

PyTypeObject **PyCoro_Type**
L'objet type correspondant aux objets coroutines.

int **PyCoro_CheckExact**(*PyObject *ob*)
Renvoie vrai si *ob* est de type *PyCoro_Type*. *ob* ne doit pas être *NULL*.

*PyObject** **PyCoro_New**(*PyFrameObject *frame*, *PyObject *name*, *PyObject *qualname*)
Return value : New reference. Crée et renvoie un nouvel objet coroutine basé sur l'objet *frame*, avec `__name__` et `__qualname__` respectivement assignés de *name* et *qualname*. Une référence à *frame* est empruntée par cette fonction. L'argument *frame* ne doit pas être *NULL*.

8.6.12 Context Variables Objects

Note : Modifié dans la version 3.7.1 : In Python 3.7.1 the signatures of all context variables C APIs were **changed** to use `PyObject` pointers instead of `PyContext`, `PyContextVar`, and `PyContextToken`, e.g. :

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

See bpo-34762 for more details.

Nouveau dans la version 3.7.

This section details the public C API for the `contextvars` module.

PyContext
The C structure used to represent a `contextvars.Context` object.

PyContextVar
The C structure used to represent a `contextvars.ContextVar` object.

PyContextToken
The C structure used to represent a `contextvars.Token` object.

PyTypeObject **PyContext_Type**
The type object representing the *context* type.

PyTypeObject **PyContextVar_Type**
The type object representing the *context variable* type.

PyTypeObject **PyContextToken_Type**
The type object representing the *context variable token* type.

Macros pour vérifier les types :

int **PyContext_CheckExact** (*PyObject* *o)
Return true if *o* is of type `PyContext_Type`. *o* must not be *NULL*. This function always succeeds.

int **PyContextVar_CheckExact** (*PyObject* *o)
Return true if *o* is of type `PyContextVar_Type`. *o* must not be *NULL*. This function always succeeds.

int **PyContextToken_CheckExact** (*PyObject* *o)
Return true if *o* is of type `PyContextToken_Type`. *o* must not be *NULL*. This function always succeeds.

Context object management functions :

PyObject ***PyContext_New** (void)
Return value : New reference. Create a new empty context object. Returns `NULL` if an error has occurred.

PyObject ***PyContext_Copy** (*PyObject* *ctx)
Return value : New reference. Create a shallow copy of the passed *ctx* context object. Returns `NULL` if an error has occurred.

PyObject ***PyContext_CopyCurrent** (void)
Return value : New reference. Create a shallow copy of the current thread context. Returns `NULL` if an error has occurred.

int **PyContext_Enter** (*PyObject* *ctx)
Set *ctx* as the current context for the current thread. Returns `0` on success, and `-1` on error.

int **PyContext_Exit** (*PyObject* *ctx)
Deactivate the *ctx* context and restore the previous context as the current context for the current thread. Returns `0` on success, and `-1` on error.

int **PyContext_ClearFreeList** ()
Clear the context variable free list. Return the total number of freed items. This function always succeeds.

Context variable functions :

PyObject ***PyContextVar_New** (const char *name, *PyObject* *def)
Return value : New reference. Create a new `ContextVar` object. The *name* parameter is used for introspection and debug purposes. The *def* parameter may optionally specify the default value for the context variable. If an error has occurred, this function returns `NULL`.

int **PyContextVar_Get** (*PyObject* *var, *PyObject* *default_value, *PyObject* **value)
Get the value of a context variable. Returns `-1` if an error has occurred during lookup, and `0` if no error occurred, whether or not a value was found.

If the context variable was found, *value* will be a pointer to it. If the context variable was *not* found, *value* will point to :

— *default_value*, if not `NULL` ;
— the default value of *var*, if not `NULL` ;
— `NULL`

If the value was found, the function will create a new reference to it.

PyObject ***PyContextVar_Set** (*PyObject* *var, *PyObject* *value)
Return value : New reference. Set the value of *var* to *value* in the current context. Returns a pointer to a `PyObject` object, or `NULL` if an error has occurred.

int **PyContextVar_Reset** (*PyObject* *var, *PyObject* *token)
Reset the state of the *var* context variable to that it was in before `PyContextVar_Set()` that returned the *token* was called. This function returns `0` on success and `-1` on error.

8.6.13 Objets DateTime

De nombreux objets *date* et *time* sont exposés par le module `DateTime`. Avant d'utiliser une de ces fonctions, le fichier d'en-tête `datetime.h` doit être inclus dans vos sources (veuillez noter qu'il n'est pas inclus par le fichier `Python.h`) et la macro `PyDateTime_IMPORT` doit-être invoquée, généralement lors de la fonction d'initialisation du module. La macro crée un pointeur vers une structure C et place celui-ci dans une variable statique, `PyDateTimeAPI`, qui est utilisée par les macros suivantes.

Macro pour accéder au singleton UTC :

*PyObject** **PyDateTime_TimeZone_UTC**
: Renvoie le singleton du fuseau horaire UTC, qui est le même objet que `datetime.timezone.utc`.
: Nouveau dans la version 3.7.

Macros pour vérifier les types :

int **PyDate_Check** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_DateType` ou un sous-type de `PyDateTime_DateType`. *ob* ne doit pas être *NULL*.

int **PyDate_CheckExact** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_DateType`. *ob* ne doit pas être *NULL*.

int **PyDateTime_Check** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_DateTimeType` ou un sous-type de `PyDateTime_DateTimeType`. *ob* ne doit pas être *NULL*.

int **PyDateTime_CheckExact** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_DateTimeType`. *ob* ne doit pas être *NULL*.

int **PyTime_Check** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_TimeType` ou un sous-type de `PyDateTime_TimeType`. *ob* ne doit pas être *NULL*.

int **PyTime_CheckExact** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_TimeType`. *ob* ne doit pas être *NULL*.

int **PyDelta_Check** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_DeltaType` ou un sous-type de `PyDateTime_DeltaType`. *ob* ne doit pas être *NULL*.

int **PyDelta_CheckExact** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_DeltaType`. *ob* ne doit pas être *NULL*.

int **PyTZInfo_Check** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_TZInfoType` ou un sous-type de `PyDateTime_TZInfoType`. *ob* ne doit pas être *NULL*.

int **PyTZInfo_CheckExact** (*PyObject *ob*)
: Renvoie vrai si *ob* est de type `PyDateTime_TZInfoType`. *ob* ne doit pas être *NULL*.

Macros pour créer des objets :

*PyObject** **PyDate_FromDate** (int *year*, int *month*, int *day*)
: *Return value : New reference.* Renvoie un objet `datetime.date` avec l'année, le mois et le jour spécifiés.

*PyObject** **PyDateTime_FromDateAndTime** (int *year*, int *month*, int *day*, int *hour*, int *minute*, int *second*, int *usecond*)
: *Return value : New reference.* Renvoie un objet `datetime.datetime` avec l'année, le mois, le jour, l'heure, la minute, la seconde et la microseconde spécifiés.

*PyObject** **PyDateTime_FromDateAndTimeAndFold**(int *year*, int *month*, int *day*, int *hour*, int *minute*, int *second*, int *usecond*, int *fold*)

Return value : New reference. Renvoie un objet `datetime.datetime` avec l'année, le mois, le jour, l'heure, la minute, la seconde, la microseconde et le pli (*fold* en anglais) spécifiés.

Nouveau dans la version 3.6.

*PyObject** **PyTime_FromTime**(int *hour*, int *minute*, int *second*, int *usecond*)

Return value : New reference. Renvoie un objet `datetime.time` avec l'heure, la minute, la seconde et la microseconde spécifiées.

*PyObject** **PyTime_FromTimeAndFold**(int *hour*, int *minute*, int *second*, int *usecond*, int *fold*)

Return value : New reference. Renvoie un objet `datetime.time` avec l'heure, la minute, la seconde, la microseconde et le pli (*fold* en anglais) spécifiés.

Nouveau dans la version 3.6.

*PyObject** **PyDelta_FromDSU**(int *days*, int *seconds*, int *useconds*)

Return value : New reference. Renvoie un objet `datetime.timedelta` représentant le nombre passé en paramètre de jours, de secondes et de microsecondes. Le résultat est normalisé pour que le nombre de microsecondes et de secondes tombe dans la plage documentée pour les objets `datetime.timedelta`.

*PyObject** **PyTimeZone_FromOffset**(PyDateTime_DeltaType* *offset*)

Return value : New reference. Renvoie un objet `datetime.timezone` avec un décalage anonyme fixe représenté par l'argument *offset*.

Nouveau dans la version 3.7.

*PyObject** **PyTimeZone_FromOffsetAndName**(PyDateTime_DeltaType* *offset*, PyUnicode* *name*)

Return value : New reference. Renvoie un objet `datetime.timezone` avec un décalage fixe représenté par l'argument *offset* et avec le nom de fuseau horaire *name*.

Nouveau dans la version 3.7.

Macros pour extraire les champs des objets *date*. L'argument doit être une instance de `PyDateTime_Date`, ou une sous-classe (telle que `PyDateTime_DateTime`). L'argument ne doit pas être *NULL*, et le type n'est pas vérifié :

int **PyDateTime_GET_YEAR**(PyDateTime_Date **o*)

Renvoie l'année, sous forme d'entier positif.

int **PyDateTime_GET_MONTH**(PyDateTime_Date **o*)

Renvoie le mois, sous forme d'entier allant de 1 à 12.

int **PyDateTime_GET_DAY**(PyDateTime_Date **o*)

Renvoie le jour, sous forme d'entier allant de 1 à 31.

Macros pour extraire les champs des objets *datetime*. L'argument doit être une instance de `PyDateTime_DateTime` ou une sous-classe de celle-ci. L'argument ne doit pas être *NULL*, et le type n'est pas vérifié :

int **PyDateTime_DATE_GET_HOUR**(PyDateTime_DateTime **o*)

Renvoie l'heure, sous forme d'entier allant de 0 à 23.

int **PyDateTime_DATE_GET_MINUTE**(PyDateTime_DateTime **o*)

Renvoie la minute, sous forme d'entier allant de 0 à 59.

int **PyDateTime_DATE_GET_SECOND**(PyDateTime_DateTime **o*)

Renvoie la seconde, sous forme d'entier allant de 0 à 59.

int **PyDateTime_DATE_GET_MICROSECOND**(PyDateTime_DateTime **o*)

Renvoie la microseconde, sous forme d'entier allant de 0 à 999999.

Macros pour extraire les champs des objets *time*. L'argument doit être une instance de `PyDateTime_Time` ou une sous-classe de celle-ci. L'argument ne doit pas être *NULL*, et le type n'est pas vérifié :

int **PyDateTime_TIME_GET_HOUR**(PyDateTime_Time **o*)

Renvoie l'heure, sous forme d'entier allant de 0 à 23.

int **PyDateTime_TIME_GET_MINUTE**(PyDateTime_Time **o*)
Renvoie la minute, sous forme d'entier allant de 0 à 59.

int **PyDateTime_TIME_GET_SECOND**(PyDateTime_Time **o*)
Renvoie la seconde, sous forme d'entier allant de 0 à 59.

int **PyDateTime_TIME_GET_MICROSECOND**(PyDateTime_Time **o*)
Renvoie la microseconde, sous forme d'entier allant de 0 à 999999.

Macros pour extraire les champs des objets *time delta*. L'argument doit être une instance de `PyDateTime_Delta` ou une sous-classe de celle-ci. L'argument ne doit pas être *NULL*, et le type n'est pas vérifié :

int **PyDateTime_DELTA_GET_DAYS**(PyDateTime_Delta **o*)
Renvoie le nombre de jours, sous forme d'entier allant de -999999999 à 999999999.

Nouveau dans la version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(PyDateTime_Delta **o*)
Renvoie le nombre de secondes sous forme d'entier allant de 0 à 86399.

Nouveau dans la version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(PyDateTime_Delta **o*)
Renvoie le nombre de microsecondes, sous forme d'entier allant de 0 à 999999.

Nouveau dans la version 3.3.

Macros de confort pour les modules implémentant l'API DB :

*PyObject** **PyDateTime_FromTimestamp**(*PyObject* **args*)
Return value : New reference. Crée et renvoie un nouvel objet `datetime.datetime` à partir d'un n-uplet qui peut être passé à `datetime.datetime.fromtimestamp()`.

*PyObject** **PyDate_FromTimestamp**(*PyObject* **args*)
Return value : New reference. Crée et renvoie un nouvel objet `datetime.date` à partir d'un n-uplet qui peut être passé à `datetime.date.fromtimestamp()`.

CHAPITRE 9

Initialization, Finalization, and Threads

9.1 Before Python Initialization

In an application embedding Python, the *Py_Initialize()* function must be called before using any other Python/C API functions ; with the exception of a few functions and the *global configuration variables*.

The following functions can be safely called before Python is initialized :

- Configuration functions :
  - *PyImport_AppendInittab()*
  - *PyImport_ExtendInittab()*
  - PyInitFrozenExtensions()
  - *PyMem_SetAllocator()*
  - *PyMem_SetupDebugHooks()*
  - *PyObject_SetArenaAllocator()*
  - *Py_SetPath()*
  - *Py_SetProgramName()*
  - *Py_SetPythonHome()*
  - *Py_SetStandardStreamEncoding()*
  - *PySys_AddWarnOption()*
  - *PySys_AddXOption()*
  - *PySys_ResetWarnOptions()*
- Informative functions :
  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
- Utilities :
  - *Py_DecodeLocale()*
- Memory allocators :

— *PyMem_RawMalloc()*
— *PyMem_RawRealloc()*
— *PyMem_RawCalloc()*
— *PyMem_RawFree()*

Note : The following functions **should not be called** before *Py_Initialize()* : *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()* and *PyEval_InitThreads()*.

9.2 Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets *Py_BytesWarningFlag* to 1 and `-bb` sets *Py_BytesWarningFlag* to 2.

Py_BytesWarningFlag
Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error if greater or equal to `2`.

Set by the `-b` option.

Py_DebugFlag
Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

Py_DontWriteBytecodeFlag
If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

Set by the `-B` option and the `PYTHONDONTWRITEBYTECODE` environment variable.

Py_FrozenFlag
Suppress error messages when calculating the module search path in *Py_GetPath()*.

Private flag used by `_freeze_importlib` and `frozenmain` programs.

Py_HashRandomizationFlag
Set to `1` if the `PYTHONHASHSEED` environment variable is set to a non-empty string.

If the flag is non-zero, read the `PYTHONHASHSEED` environment variable to initialize the secret hash seed.

Py_IgnoreEnvironmentFlag
Ignore toutes les variables d'environnement `PYTHON*` qui pourraient être définies. Par exemple, `PYTHONPATH` et `PYTHONHOME`.

Set by the `-E` and `-I` options.

Py_InspectFlag
When a script is passed as first argument or the `-c` option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Set by the `-i` option and the `PYTHONINSPECT` environment variable.

Py_InteractiveFlag
Set by the `-i` option.

Py_IsolatedFlag
Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.

Set by the `-I` option.

Nouveau dans la version 3.4.

Py_LegacyWindowsFSEncodingFlag

If the flag is non-zero, use the `mbcs` encoding instead of the UTF-8 encoding for the filesystem encoding.

Set to `1` if the `PYTHONLEGACYWINDOWSFSENCODING` environment variable is set to a non-empty string.

Voir la **PEP 529** pour plus d'informations.

Disponibilité : Windows.

Py_LegacyWindowsStdioFlag

If the flag is non-zero, use `io.FileIO` instead of `WindowsConsoleIO` for `sys` standard streams.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

See **PEP 528** for more details.

Disponibilité : Windows.

Py_NoSiteFlag

Désactive l'importation du module `site` et les modifications locales de `sys.path` qu'il implique. Désactive aussi ces manipulations si `site` est importé explicitement plus tard (appelez `site.main()` si vous voulez les déclencher).

Set by the `-S` option.

Py_NoUserSiteDirectory

N'ajoute pas le répertoire utilisateur `site-packages` à `sys.path`.

Set by the `-s` and `-I` options, and the `PYTHONNOUSERSITE` environment variable.

Py_OptimizeFlag

Set by the `-O` option and the `PYTHONOPTIMIZE` environment variable.

Py_QuietFlag

N'affiche pas le copyright et la version, même en mode interactif.

Set by the `-q` option.

Nouveau dans la version 3.2.

Py_UnbufferedStdioFlag

Force the stdout and stderr streams to be unbuffered.

Set by the `-u` option and the `PYTHONUNBUFFERED` environment variable.

Py_VerboseFlag

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Set by the `-v` option and the `PYTHONVERBOSE` environment variable.

9.3 Initializing and finalizing the interpreter

void **Py_Initialize**()

Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see *Before Python Initialization* for the few exceptions.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use `PySys_SetArgvEx()` for that. This is a no-op when called for a second time (without calling `Py_FinalizeEx()` first). There is no return value; it is a fatal error if the initialization fails.

Note : On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

void **Py_InitializeEx**(int *initsigs*)

This function works like *Py_Initialize()* if *initsigs* is 1. If *initsigs* is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

int **Py_IsInitialized**()

Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After *Py_FinalizeEx()* is called, this returns false until *Py_Initialize()* is called again.

int **Py_FinalizeEx**()

Undo all initializations made by *Py_Initialize()* and subsequent use of Python/C API functions, and destroy all sub-interpreters (see *Py_NewInterpreter()* below) that were created and not yet destroyed since the last call to *Py_Initialize()*. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling *Py_Initialize()* again first). Normally the return value is 0. If there were errors during finalization (flushing buffered data), -1 is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats : The destruction of modules and objects in modules is done in random order ; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once ; this can happen if an application calls *Py_Initialize()* and *Py_FinalizeEx()* more than once.

Nouveau dans la version 3.6.

void **Py_Finalize**()

This is a backwards-compatible version of *Py_FinalizeEx()* that disregards the return value.

9.4 Process-wide parameters

int **Py_SetStandardStreamEncoding**(const char **encoding*, const char **errors*)

This function should be called before *Py_Initialize()*, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in `str.encode()`.

It overrides `PYTHONIOENCODING` values, and allows embedding code to control IO encoding when the environment variable does not work.

`encoding` and/or `errors` may be NULL to use `PYTHONIOENCODING` and/or default values (depending on other settings).

Note that `sys.stderr` always uses the "backslashreplace" error handler, regardless of this (or any other) setting.

If *Py_FinalizeEx()* is called, this function will need to be called again in order to affect subsequent calls to *Py_Initialize()*.

Returns 0 if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

Nouveau dans la version 3.4.

void **Py_SetProgramName**(const wchar_t **name*)

This function should be called before *Py_Initialize()* is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the `main()` function of the program (converted to wide characters). This is used by *Py_GetPath()* and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

wchar* **Py_GetProgramName**()

Return the program name set with `Py_SetProgramName()`, or the default. The returned string points into static storage; the caller should not modify its value.

wchar_t* **Py_GetPrefix**()

Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.prefix`. It is only useful on Unix. See also the next function.

wchar_t* **Py_GetExecPrefix**()

Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the exec-prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level `Makefile` and the `--exec-prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.exec_prefix`. It is only useful on Unix.

Background : The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled !).

System administrators will know how to configure the **mount** or **automount** programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

wchar_t* **Py_GetProgramFullPath**()

Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by `Py_SetProgramName()` above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

wchar_t* **Py_GetPath**()

Return the default module search path; this is computed from the program name (set by `Py_SetProgramName()` above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and Mac OS X, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

void **Py_SetPath**(const wchar_t *)

Set the default module search path. If this function is called before `Py_Initialize()`, then `Py_GetPath()` won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is `':'` on Unix and Mac OS X, `';'` on Windows.

This also causes `sys.executable` to be set only to the raw program name (see `Py_SetProgramName()`) and for `sys.prefix` and `sys.exec_prefix` to be empty. It is up to the caller to modify these if required after calling `Py_Initialize()`.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

The path argument is copied internally, so the caller may free it after the call completes.

const char* **Py_GetVersion**()

Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version ; the first three characters are the major and minor version separated by a period. The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as `sys.version`.

const char* **Py_GetPlatform**()

Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number ; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On Mac OS X, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char* **Py_GetCopyright**()

Return the official copyright string for the current Python version, for example

`'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'`

The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char* **Py_GetCompiler**()

Return an indication of the compiler used to build the current Python version, in square brackets, for example :

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char* **Py_GetBuildInfo**()

Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void **PySys_SetArgvEx**(int *argc*, wchar_t ***argv*, int *updatepath*)

Set `sys.argv` based on *argc* and *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using *Py_FatalError()*.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies `sys.path` according to the following algorithm :

— If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.

— Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`"."`).

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

Note : It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See CVE-2008-5983.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called *PySys_SetArgv()*, for example using :

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Nouveau dans la version 3.1.3.

void **PySys_SetArgv** (int *argc*, wchar_t ***argv*)

This function works like `PySys_SetArgvEx()` with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

Modifié dans la version 3.4 : The *updatepath* value depends on `-I`.

void **Py_SetPythonHome** (const wchar_t **home*)

Set the default "home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

w_char* **Py_GetPythonHome** ()

Return the default "home", that is, the value set by a previous call to `Py_SetPythonHome()`, or the value of the `PYTHONHOME` environment variable if it is set.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program : for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called `PyThreadState`. There's also one global variable pointing to the current `PyThreadState` : it can be retrieved using `PyThreadState_Get()`.

9.5.1 Releasing the GIL from extension code

Most extension code manipulating the *GIL* has the following simple structure :

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it :

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The *Py_BEGIN_ALLOW_THREADS* macro opens a new block and declares a hidden local variable; the *Py_END_ALLOW_THREADS* macro closes the block.

The block above expands to the following code :

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work : the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

Note : Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules release the GIL when compressing or hashing data.

9.5.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The *PyGILState_Ensure()* and *PyGILState_Release()* functions do all of the above automatically. The typical idiom for calling into Python from a C thread is :

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*()` functions assume there is only one global interpreter (created automatically by *Py_Initialize()*). Python supports the creation of additional interpreters (using *Py_NewInterpreter()*), but mixing multiple interpreters and the `PyGILState_*()` API is unsupported.

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. That also means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding

Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. `PyOS_AfterFork_Child()` tries to reset the necessary locks, but is not always able to.

9.5.3 High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter :

PyInterpreterState

This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

PyThreadState

This data structure represents the state of a single thread. The only public data member is `PyInterpreterState *interp`, which points to this thread's interpreter state.

void **PyEval_InitThreads**()

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating a second thread or engaging in any other thread operations such as `PyEval_ReleaseThread(tstate)`. It is not needed before calling `PyEval_SaveThread()` or `PyEval_RestoreThread()`.

This is a no-op when called for a second time.

Modifié dans la version 3.7 : This function is now called by `Py_Initialize()`, so you don't have to call it yourself anymore.

Modifié dans la version 3.2 : This function cannot be called before `Py_Initialize()` anymore.

int **PyEval_ThreadsInitialized**()

Returns a non-zero value if `PyEval_InitThreads()` has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

Modifié dans la version 3.7 : The *GIL* is now initialized by `Py_Initialize()`.

*PyThreadState** **PyEval_SaveThread**()

Release the global interpreter lock (if it has been created and thread support is enabled) and reset the thread state to *NULL*, returning the previous thread state (which is not *NULL*). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread**(*PyThreadState *tstate*)

Acquire the global interpreter lock (if it has been created and thread support is enabled) and set the thread state to *tstate*, which must not be *NULL*. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

Note : Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

*PyThreadState** **PyThreadState_Get**()

Return the current thread state. The global interpreter lock must be held. When the current thread state is *NULL*, this issues a fatal error (so that the caller needn't check for *NULL*).

*PyThreadState** **PyThreadState_Swap** (*PyThreadState* **tstate*)

Swap the current thread state with the thread state given by the argument *tstate*, which may be *NULL*. The global interpreter lock must be held and is not released.

void **PyEval_ReInitThreads** ()

This function is called from `PyOS_AfterFork_Child()` to ensure that newly created child processes don't hold locks referring to threads which are not running in the child process.

The following functions use thread-local storage, and are not compatible with sub-interpreters :

PyGILState_STATE **PyGILState_Ensure** ()

Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to `PyGILState_Release()`. In general, other thread-related APIs may be used between `PyGILState_Ensure()` and `PyGILState_Release()` calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros is acceptable.

The return value is an opaque "handle" to the thread state when `PyGILState_Ensure()` was called, and must be passed to `PyGILState_Release()` to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to `PyGILState_Ensure()` must save the handle for its call to `PyGILState_Release()`.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

Note : Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

void **PyGILState_Release** (PyGILState_STATE)

Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding `PyGILState_Ensure()` call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to `PyGILState_Ensure()` must be matched by a call to `PyGILState_Release()` on the same thread.

*PyThreadState** **PyGILState_GetThisThreadState** ()

Get the current thread state for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check** ()

Return `1` if the current thread is holding the GIL and `0` otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Nouveau dans la version 3.4.

The following macros are normally used without a trailing semicolon ; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace ; it must be matched with a following `Py_END_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS
This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace ; it must be matched with an earlier *Py_BEGIN_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_BLOCK_THREADS
This macro expands to `PyEval_RestoreThread(_save);` : it is equivalent to *Py_END_ALLOW_THREADS* without the closing brace.

Py_UNBLOCK_THREADS
This macro expands to `_save = PyEval_SaveThread();` : it is equivalent to *Py_BEGIN_ALLOW_THREADS* without the opening brace and variable declaration.

9.5.4 Low-level API

All of the following functions must be called after *Py_Initialize()*.

Modifié dans la version 3.7 : *Py_Initialize()* now initializes the *GIL*.

*PyInterpreterState** **PyInterpreterState_New**()
Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyInterpreterState_Clear**(*PyInterpreterState *interp*)
Reset all information in an interpreter state object. The global interpreter lock must be held.

void **PyInterpreterState_Delete**(*PyInterpreterState *interp*)
Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to *PyInterpreterState_Clear()*.

*PyThreadState** **PyThreadState_New**(*PyInterpreterState *interp*)
Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear**(*PyThreadState *tstate*)
Reset all information in a thread state object. The global interpreter lock must be held.

void **PyThreadState_Delete**(*PyThreadState *tstate*)
Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

PY_INT64_T **PyInterpreterState_GetID**(*PyInterpreterState *interp*)
Return the interpreter's unique ID. If there was any error in doing so then `-1` is returned and an error is set.

Nouveau dans la version 3.7.

*PyObject** **PyThreadState_GetDict**()
Return value : Borrowed reference. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns *NULL*, no exception has been raised and the caller should assume no current thread state is available.

int **PyThreadState_SetAsyncExc**(unsigned long *id*, *PyObject *exc*)
Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread ; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified ; this is normally one, but will be zero if the thread id isn't found. If *exc* is `NULL`, the pending exception (if any) for the thread is cleared. This raises no exceptions.

Modifié dans la version 3.7 : The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState *tstate*)

Acquire the global interpreter lock and set the current thread state to *tstate*, which should not be *NULL*. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState *tstate*)

Reset the current thread state to *NULL* and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be *NULL*, is only used to check that it represents the current thread state — if it isn't, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_AcquireLock**()

Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

Obsolète depuis la version 3.2 : This function does not update the current thread state. Please use `PyEval_RestoreThread()` or `PyEval_AcquireThread()` instead.

void **PyEval_ReleaseLock**()

Release the global interpreter lock. The lock must have been created earlier.

Obsolète depuis la version 3.2 : This function does not update the current thread state. Please use `PyEval_SaveThread()` or `PyEval_ReleaseThread()` instead.

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that. You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions :

*PyThreadState** **Py_NewInterpreter**()

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created ; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, *NULL* is returned ; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns ; however, unlike most other Python/C API functions, there needn't be a current thread state on entry.)

Extension modules are shared between (sub-)interpreters as follows : the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy ; the extension's `init` function is not called. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling `Py_FinalizeEx()` and `Py_Initialize()` ; in that case, the extension's `initmodule` function *is* called again.

void **Py_EndInterpreter**(*PyThreadState *tstate*)

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is *NULL*. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) `Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly ; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module's dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter ; this should be done with great care to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules.

Also note that combining this functionality with `PyGILState_*()` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching `PyGILState_Ensure()` and `PyGILState_Release()` calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

9.7 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **Py_AddPendingCall**(int (**func*)(void *), void **arg*)

Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met :

— on a *bytecode* boundary ;

— with the main thread holding the *global interpreter lock* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn't need a current thread state to run, and it doesn't need the global interpreter lock.

> **Avertissement :** This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the *PyGILState API*.

Nouveau dans la version 3.1.

9.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

int **(*Py_tracefunc)** (*PyObject *obj*, *PyFrameObject *frame*, int *what*, *PyObject *arg*)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what* :

Value of *what*	Meaning of *arg*
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or *NULL* if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.
`PyTrace_OPCODE`	Always `Py_None`.

int **PyTrace_CALL**

The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int **PyTrace_LINE**

The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**

The value for the *what* parameter to `Py_tracefunc` functions when a call is about to return.

int **PyTrace_C_CALL**

The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**

The value for the *what* parameter to `Py_tracefunc` functions when a C function has raised an exception.

int **PyTrace_C_RETURN**

The value for the *what* parameter to `Py_tracefunc` functions when a C function has returned.

int **PyTrace_OPCODE**

The value for the *what* parameter to `Py_tracefunc` functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default : it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile** (*Py_tracefunc func*, *PyObject *obj*)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or *NULL*. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except `PyTrace_LINE` `PyTrace_OPCODE` and `PyTrace_EXCEPTION`.

void **PyEval_SetTrace** (*Py_tracefunc func*, *PyObject *obj*)
Set the tracing function to *func*. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using `PyEval_SetTrace()` will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

9.9 Support avancé du débogueur

These functions are only intended to be used by advanced debugging tools.

*PyInterpreterState** **PyInterpreterState_Head** ()
Return the interpreter state object at the head of the list of all such objects.

*PyInterpreterState** **PyInterpreterState_Main** ()
Return the main interpreter state object.

*PyInterpreterState** **PyInterpreterState_Next** (*PyInterpreterState *interp*)
Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState * **PyInterpreterState_ThreadHead** (*PyInterpreterState *interp*)
Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

*PyThreadState** **PyThreadState_Next** (*PyThreadState *tstate*)
Return the next thread state object after *tstate* from the list of all such objects belonging to the same `PyInterpreterState` object.

9.10 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows : use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions ; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

Note : None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

9.10.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type `Py_tss_t` instead of `int` to represent thread keys.

Nouveau dans la version 3.7.

Voir aussi :

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

Py_tss_t

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When *Py_LIMITED_API* is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

Py_tss_NEEDS_INIT

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Dynamic Allocation

Dynamic allocation of the `Py_tss_t`, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

*Py_tss_t** **PyThread_tss_alloc**()

Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or *NULL* in the case of dynamic allocation failure.

void **PyThread_tss_free**(*Py_tss_t* **key*)

Free the given *key* allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is *NULL*.

Note : A freed key becomes a dangling pointer, you should reset the key to *NULL*.

Méthodes

The parameter *key* of these functions must not be *NULL*. Moreover, the behaviors of `PyThread_tss_set()` and `PyThread_tss_get()` are undefined if the given `Py_tss_t` has not been initialized by `PyThread_tss_create()`.

int **PyThread_tss_is_created**(*Py_tss_t* **key*)

Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

int **PyThread_tss_create**(*Py_tss_t* **key*)

Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key – calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete**(*Py_tss_t* **key*)

Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t* **key*, void **value*)

Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void* **PyThread_tss_get**(*Py_tss_t* **key*)

Return the `void*` value associated with a TSS key in the current thread. This returns *NULL* if no value is associated with the key in the current thread.

9.10.2 Thread Local Storage (TLS) API

Obsolète depuis la version 3.7 : This API is superseded by *Thread Specific Storage (TSS) API*.

Note : This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, `PyThread_create_key()` will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **PyThread_create_key**()

void **PyThread_delete_key**(int *key*)

int **PyThread_set_key_value**(int *key*, void **value*)

void* **PyThread_get_key_value**(int *key*)

void **PyThread_delete_key_value**(int *key*)

void **PyThread_ReInitTLS**()

CHAPITRE 10

Memory Management

10.1 Aperçu

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library : `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example :

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
```

(suite sur la page suivante)

(suite de la page précédente)

```
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

Voir aussi :

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

10.2 Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the *GIL* does not need to be held.

The *default raw memory allocator* uses the following functions : `malloc()`, `calloc()`, `realloc()` and `free()` ; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Nouveau dans la version 3.4.

void* **PyMem_RawMalloc**(size_t *n*)

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails.

Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_RawCalloc**(size_t *nelem*, size_t *elsize*)

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

Nouveau dans la version 3.5.

void* **PyMem_RawRealloc**(void **p*, size_t *n*)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is *NULL*, the call is equivalent to `PyMem_RawMalloc(n)` ; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`.

If the request fails, `PyMem_RawRealloc()` returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyMem_RawFree**(void *p*)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_RawMalloc()*, *PyMem_RawRealloc()* or *PyMem_RawCalloc()*. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is *NULL*, no operation is performed.

10.3 Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default memory allocator* uses the *pymalloc memory allocator*.

Avertissement : The *GIL* must be held when using these functions.

Modifié dans la version 3.6 : The default allocator is now pymalloc instead of system `malloc()`.

void* **PyMem_Malloc**(size_t *n*)

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails.

Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_Calloc**(size_t *nelem*, size_t *elsize*)

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

Nouveau dans la version 3.5.

void* **PyMem_Realloc**(void *p*, size_t *n*)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is *NULL*, the call is equivalent to `PyMem_Malloc(n)` ; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*.

If the request fails, *PyMem_Realloc()* returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void *p*)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is *NULL*, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

TYPE* **PyMem_New**(TYPE, size_t *n*)

Same as *PyMem_Malloc()*, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

TYPE* **PyMem_Resize**(void *p*, TYPE, size_t *n*)

Same as *PyMem_Realloc()*, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or *NULL* in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void *p)
Same as `PyMem_Free()`.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

— `PyMem_MALLOC(size)`
— `PyMem_NEW(type, size)`
— `PyMem_REALLOC(ptr, size)`
— `PyMem_RESIZE(ptr, type, size)`
— `PyMem_FREE(ptr)`
— `PyMem_DEL(ptr)`

10.4 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default object allocator* uses the *pymalloc memory allocator*.

Avertissement : The *GIL* must be held when using these functions.

void* **PyObject_Malloc**(size_t *n*)
Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails.

Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyObject_Calloc**(size_t *nelem*, size_t *elsize*)
Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

Nouveau dans la version 3.5.

void* **PyObject_Realloc**(void *p, size_t *n*)
Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is *NULL*, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`.

If the request fails, `PyObject_Realloc()` returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free**(void *p)
Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is *NULL*, no operation is performed.

10.5 Default Memory Allocators

Default memory allocators :

Configuration	Nom	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend :

- Name : value for `PYTHONMALLOC` environment variable
- `malloc` : system allocators from the standard C library, C functions : `malloc()`, `calloc()`, `realloc()` and `free()`
- `pymalloc` : *pymalloc memory allocator*
- "+ debug" : with debug hooks installed by `PyMem_SetupDebugHooks()`

10.6 Customize Memory Allocators

Nouveau dans la version 3.4.

`PyMemAllocatorEx`

Structure used to describe a memory block allocator. The structure has four fields :

Field	Signification
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

Modifié dans la version 3.5 : The `PyMemAllocator` structure was renamed to `PyMemAllocatorEx` and a new `calloc` field was added.

`PyMemAllocatorDomain`

Enum used to identify an allocator domain. Domains :

`PYMEM_DOMAIN_RAW`

Functions :

- `PyMem_RawMalloc()`
- `PyMem_RawRealloc()`
- `PyMem_RawCalloc()`
- `PyMem_RawFree()`

`PYMEM_DOMAIN_MEM`

Functions :

- `PyMem_Malloc()`,
- `PyMem_Realloc()`

— `PyMem_Calloc()`
— `PyMem_Free()`

PYMEM_DOMAIN_OBJ
Functions :
— `PyObject_Malloc()`
— `PyObject_Realloc()`
— `PyObject_Calloc()`
— `PyObject_Free()`

void **PyMem_GetAllocator**(*PyMemAllocatorDomain domain*, *PyMemAllocatorEx *allocator*)
Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain domain*, *PyMemAllocatorEx *allocator*)
Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-NULL pointer when requesting zero bytes.

For the `PYMEM_DOMAIN_RAW` domain, the allocator must be thread-safe : the *GIL* is not held when the allocator is called.

If the new allocator is not a hook (does not call the previous allocator), the `PyMem_SetupDebugHooks()` function must be called to reinstall the debug hooks on top on the new allocator.

void **PyMem_SetupDebugHooks**(void)
Setup hooks to detect bugs in the Python memory allocator functions.

Newly allocated memory is filled with the byte `0xCD` (`CLEANBYTE`), freed memory is filled with the byte `0xDD` (`DEADBYTE`). Memory blocks are surrounded by ”forbidden bytes” (`FORBIDDENBYTE` : byte `0xFD`).

Runtime checks :
— Detect API violations, ex : `PyObject_Free()` called on a buffer allocated by `PyMem_Malloc()`
— Detect write before the start of the buffer (buffer underflow)
— Detect write after the end of the buffer (buffer overflow)
— Check that the *GIL* is held when allocator functions of `PYMEM_DOMAIN_OBJ` (ex : `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex : `PyMem_Malloc()`) domains are called

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

These hooks are *installed by default* if Python is compiled in debug mode. The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode.

Modifié dans la version 3.6 : This function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of `PYMEM_DOMAIN_OBJ` and `PYMEM_DOMAIN_MEM` domains are called.

Modifié dans la version 3.7.3 : Byte patterns `0xCB` (`CLEANBYTE`), `0xDB` (`DEADBYTE`) and `0xFB` (`FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

10.7 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called ”arenas” with a fixed size of 256 KiB. It falls back to `PyMem_RawMalloc()` and `PyMem_RawRealloc()` for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the `PYMEM_DOMAIN_MEM` (ex : `PyMem_Malloc()`) and `PYMEM_DOMAIN_OBJ` (ex : `PyObject_Malloc()`) domains.

The arena allocator uses the following functions :
— `VirtualAlloc()` and `VirtualFree()` on Windows,

— `mmap()` and `munmap()` if available,
— `malloc()` and `free()` otherwise.

10.7.1 Customize pymalloc Arena Allocator

Nouveau dans la version 3.4.

PyObjectArenaAllocator
Structure used to describe an arena allocator. The structure has three fields :

Field	Signification
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, size_t size, void *ptr)`	free an arena

PyObject_GetArenaAllocator(*PyObjectArenaAllocator *allocator*)
Get the arena allocator.

PyObject_SetArenaAllocator(*PyObjectArenaAllocator *allocator*)
Set the arena allocator.

10.8 tracemalloc C API

Nouveau dans la version 3.7.

10.9 Exemples

Here is the example from section *Aperçu*, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set :

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set :

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New()`, `PyObject_NewVar()` and `PyObject_Del()`.

These will be explained in the next chapter on defining and implementing new object types in C.

CHAPITRE 11

Implémentation d'objets

Ce chapitre décrit les fonctions, types, et macros utilisées pour définir de nouveaux types d'objets.

11.1 Allouer des objets dans le tas

*PyObject** **_PyObject_New** (*PyTypeObject *type*)

Return value : New reference.

*PyVarObject** **_PyObject_NewVar** (*PyTypeObject *type*, Py_ssize_t *size*)

Return value : New reference.

*PyObject** **PyObject_Init** (*PyObject *op*, *PyTypeObject *type*)

Return value : Borrowed reference. Permet d'initialiser un objet *op* nouvellement alloué ainsi que son type et sa référence initiale. Renvoie l'objet initialisé. La présence de *type* indique que l'objet doit être traité par le détecteur d'ordures cycliques, il est de ce fait ajouté à l'ensemble du détecteur d'objets observés. Les autres champs de l'objet ne sont pas affectés.

*PyVarObject** **PyObject_InitVar** (*PyVarObject *op*, *PyTypeObject *type*, Py_ssize_t *size*)

Return value : Borrowed reference. Effectue les mêmes opérations que `PyObject_Init()` fait, et initialise également l'information de la longueur pour un objet de taille variable.

TYPE* **PyObject_New** (TYPE, *PyTypeObject *type*)

Return value : New reference. Alloue un nouvel objet Python en utilisant le type de structure C *TYPE* et l'objet Python *type*. Les champs non définis par l'en-tête de l'objet Python ne sont pas initialisés ; le compteur de la référence objet sera égal à un. La taille de l'allocation mémoire est déterminée par le champ `tp_basicsize` de l'objet type.

TYPE* **PyObject_NewVar** (TYPE, *PyTypeObject *type*, Py_ssize_t *size*)

Return value : New reference. Alloue un nouvel objet Python en utilisant le type de structure C *TYPE* et l'objet Python de type *type*. Les champs non définis par l'en-tête de l'objet Python ne sont pas initialisés. La mémoire allouée est suffisante pour la structure *TYPE* plus *size* champs de la taille donnée par le champ de *type* `tp_itemsize`. Ceci est utile pour l'implémentation d'objets comme les tuples, qui sont capables de déterminer leur taille à la construction. Allouer les champs en même temps que l'objet diminue le nombre d'allocations, améliorant ainsi les performances.

void **PyObject_Del**(void *op)
Libère la mémoire allouée à un objet utilisant *PyObject_New()* ou *PyObject_NewVar()*. Ceci est normalement appelé par le gestionnaire *tp_dealloc* spécifié dans le type d'objet. Les champs de l'objet ne doivent plus être accédés après cet appel puisque cet emplacement mémoire ne correspond plus à un objet Python valide.

PyObject **_Py_NoneStruct**
Objet qui est visible en tant que `None` dans Python. Ne devrait être accessible uniquement en utilisant la macro *Py_None*, qui évalue cet objet à un pointeur.

Voir aussi :

PyModule_Create() Allouer et créer des modules d'extension.

11.2 Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the *PyObject* and *PyVarObject* types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

PyObject
All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**. Access to the members must be done by using the macros *Py_REFCNT* and *Py_TYPE*.

PyVarObject
This is an extension of *PyObject* that adds the `ob_size` field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be done by using the macros *Py_REFCNT*, *Py_TYPE*, and *Py_SIZE*.

PyObject_HEAD
This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to :

```
PyObject ob_base;
```

See documentation of *PyObject* above.

PyObject_VAR_HEAD
This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to :

```
PyVarObject ob_base;
```

See documentation of *PyVarObject* above.

Py_TYPE(o)
This macro is used to access the `ob_type` member of a Python object. It expands to :

```
(((PyObject*)(o))->ob_type)
```

Py_REFCNT(o)
This macro is used to access the `ob_refcnt` member of a Python object. It expands to :

```
(((PyObject*)(o))->ob_refcnt)
```

Py_SIZE(o)
This macro is used to access the ob_size member of a Python object. It expands to :

```
(((PyVarObject*)(o))->ob_size)
```

PyObject_HEAD_INIT(type)
This is a macro which expands to initialization values for a new *PyObject* type. This macro expands to :

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)
This is a macro which expands to initialization values for a new *PyVarObject* type, including the ob_size field. This macro expands to :

```
_PyObject_EXTRA_INIT
1, type, size,
```

PyCFunction
Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject** parameters and return one such value. If the return value is *NULL*, an exception shall have been set. If not *NULL*, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

PyCFunctionWithKeywords
Type of the functions used to implement Python callables in C with signature METH_VARARGS | METH_KEYWORDS.

_PyCFunctionFast
Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*.

_PyCFunctionFastWithKeywords
Type of the functions used to implement Python callables in C with signature METH_FASTCALL | METH_KEYWORDS.

PyMethodDef
Structure used to describe a method of an extension type. This structure has four fields :

Field	Type C	Signification
ml_name	const char *	name of the method
ml_meth	PyCFunction	pointer to the C implementation
ml_flags	*int*	flag bits indicating how the call should be constructed
ml_doc	const char *	points to the contents of the docstring

The ml_meth is a C function pointer. The functions may be of different types, but they always return *PyObject**. If the function is not of the *PyCFunction*, the compiler will require a cast in the method table. Even though *PyCFunction* defines the first parameter as *PyObject**, it is common that the method implementation uses the specific C type of the *self* object.

The ml_flags field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are four basic calling conventions for positional arguments and two of them can be combined with METH_KEYWORDS to support also keyword arguments. So there are a total of 6 calling conventions :

METH_VARARGS

This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods ; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_VARARGS | METH_KEYWORDS

Methods with these flags must be of type *PyCFunctionWithKeywords*. The function expects three parameters : *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly *NULL* if there are no keyword arguments. The parameters are typically processed using *PyArg_ParseTupleAndKeywords()*.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type *_PyCFunctionFast*. The first parameter is *self*, the second parameter is a C array of *PyObject** values indicating the arguments and the third parameter is the number of arguments (the length of the array).

This is not part of the *limited API*.

Nouveau dans la version 3.7.

METH_FASTCALL | METH_KEYWORDS

Extension of *METH_FASTCALL* supporting also keyword arguments, with methods of type *_PyCFunctionFastWithKeywords*. Keyword arguments are passed the same way as in the vectorcall protocol : there is an additional fourth *PyObject** parameter which is a tuple representing the names of the keyword arguments or possibly *NULL* if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

This is not part of the *limited API*.

Nouveau dans la version 3.7.

METH_NOARGS

Methods without parameters don't need to check whether arguments are given if they are listed with the *METH_NOARGS* flag. They need to be of type *PyCFunction*. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be *NULL*.

METH_O

Methods with a single object argument can be listed with the *METH_O* flag, instead of invoking *PyArg_ParseTuple()* with a `"O"` argument. They have the type *PyCFunction*, with the *self* parameter, and a *PyObject** parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

METH_STATIC

The method will be passed *NULL* as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyMemberDef

Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are :

Field	Type C	Signification
name	const char *	name of the member
type	*int*	the type of the member in the C struct
offset	Py_ssize_t	the offset in bytes that the member is located on the type's object struct
flags	*int*	flag bits indicating if the field should be read-only or writable
doc	const char *	points to the contents of the docstring

type can be one of many T_ macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

Macro name	Type C
T_SHORT	short
T_INT	*int*
T_LONG	long
T_FLOAT	*float*
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

T_OBJECT and T_OBJECT_EX differ in that T_OBJECT returns None if the member is *NULL* and T_OBJECT_EX raises an AttributeError. Try to use T_OBJECT_EX over T_OBJECT because T_OBJECT_EX handles use of the del statement on that attribute more correctly than T_OBJECT.

flags can be 0 for write and read access or READONLY for read-only access. Using T_STRING for type implies READONLY. T_STRING data is interpreted as UTF-8. Only T_OBJECT and T_OBJECT_EX members can be deleted. (They are set to *NULL*).

PyGetSetDef

Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

Field	Type C	Signification
name	const char *	attribute name
get	getter	C Function to get the attribute
set	setter	optional C function to set or delete the attribute, if omitted the attribute is readonly
doc	const char *	optional docstring
closure	void *	optional function pointer, providing additional data for getter and setter

The get function takes one *PyObject** parameter (the instance) and a function pointer (the associated closure):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

It should return a new reference on success or *NULL* with a set exception on failure.

`set` functions take two *PyObject** parameters (the instance and the value to be set) and a function pointer (the associated `closure`) :

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

In case the attribute should be deleted the second parameter is *NULL*. Should return `0` on success or `-1` with a set exception on failure.

11.3 Objets type

Perhaps one of the most important structures of the Python object system is the structure that defines a new type : the *PyTypeObject* structure. Type objects can be handled using any of the `PyObject_*()` or `PyType_*()` functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

Typedefs : unaryfunc, binaryfunc, ternaryfunc, inquiry, intargfunc, intintargfunc, intobjargproc, intintobjargproc, objobjargproc, destructor, freefunc, printfunc, getattrfunc, getattrofunc, setattrfunc, setattrofunc, reprfunc, hashfunc

The structure definition for *PyTypeObject* can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there :

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

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

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

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
```

(suite sur la page suivante)

(suite de la page précédente)

```c
    setattrofunc tp_setattro;

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

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

    const char *tp_doc; /* Documentation string */

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

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

    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

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

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

    destructor tp_finalize;

} PyTypeObject;
```

The type object structure extends the *PyVarObject* structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the `ob_size` field.

*PyObject** **PyObject._ob_next**
*PyObject** **PyObject._ob_prev**

These fields are only present when the macro `Py_TRACE_REFS` is defined. Their initialization to *NULL* is taken care of by the `PyObject_HEAD_INIT` macro. For statically allocated objects, these fields always remain *NULL*. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of *all* live objects on the heap. This could be used for various debugging purposes ; currently the only use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

These fields are not inherited by subtypes.

Py_ssize_t **PyObject.ob_refcnt**

This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for statically allocated type objects, the type's instances (objects whose `ob_type` points back to the type) do *not* count as references. But for dynamically allocated type objects, the instances *do* count as references.

This field is not inherited by subtypes.

*PyTypeObject** **PyObject.ob_type**

This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass *NULL* to the `PyObject_HEAD_INIT` macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this :

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. *`PyType_Ready()`* checks if `ob_type` is *NULL*, and if so, initializes it to the `ob_type` field of the base class. *`PyType_Ready()`* will not change this field if it is non-zero.

This field is inherited by subtypes.

Py_ssize_t **PyVarObject.ob_size**

For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

This field is not inherited by subtypes.

const char* **PyTypeObject.tp_name**

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name ; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in module `M` in subpackage `Q` in package `P` should have the *`tp_name`* initializer `"P.Q.M.T"`.

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key `'__module__'`.

For statically allocated type objects, the tp_name field should contain a dot. Everything before the last dot is made accessible as the `__module__` attribute, and everything after the last dot is made accessible as the `__name__` attribute.

If no dot is present, the entire *`tp_name`* field is made accessible as the `__name__` attribute, and the `__module__` attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field is not inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_basicsize**
Py_ssize_t **PyTypeObject.tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types : types with fixed-length instances have a zero *`tp_itemsize`* field, types with variable-length instances have a non-zero *`tp_itemsize`* field. For a type with fixed-length instances, all instances have the same size, given in *`tp_basicsize`*.

For a type with variable-length instances, the instances must have an ob_size field, and the instance size is *tp_basicsize* plus N times *tp_itemsize*, where N is the "length" of the object. The value of N is typically stored in the instance's ob_size field. There are exceptions : for example, ints use a negative ob_size to indicate a negative number, and N is abs(ob_size) there. Also, the presence of an ob_size field in the instance layout doesn't mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful ob_size field).

The basic size includes the fields in the instance declared by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD* (whichever is used to declare the instance struct) and this in turn includes the _ob_prev and _ob_next fields if they are present. This means that the only correct way to get an initializer for the *tp_basicsize* is to use the sizeof operator on the struct used to declare the instance layout. The basic size does not include the GC header size.

These fields are inherited separately by subtypes. If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

A note about alignment : if the variable items require a particular alignment, this should be taken care of by the value of *tp_basicsize*. Example : suppose a type implements an array of double. *tp_itemsize* is sizeof(double). It is the programmer's responsibility that *tp_basicsize* is a multiple of sizeof(double) (assuming this is the alignment requirement for double).

destructor **PyTypeObject.tp_dealloc**

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons None and Ellipsis).

The destructor function is called by the *Py_DECREF()* and *Py_XDECREF()* macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and finally (as its last action) call the type's *tp_free* function. If the type is not subtypable (doesn't have the *Py_TPFLAGS_BASETYPE* flag bit set), it is permissible to call the object deallocator directly instead of via *tp_free*. The object deallocator should be the one used to allocate the instance ; this is normally *PyObject_Del()* if the instance was allocated using *PyObject_New()* or PyObject_VarNew(), or *PyObject_GC_Del()* if the instance was allocated using *PyObject_GC_New()* or *PyObject_GC_NewVar()*.

This field is inherited by subtypes.

printfunc **PyTypeObject.tp_print**

Reserved slot, formerly used for print formatting in Python 2.x.

getattrfunc **PyTypeObject.tp_getattr**

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_getattro* function, but taking a C string instead of a Python string object to give the attribute name. The signature is

```
PyObject * tp_getattr(PyObject *o, char *attr_name);
```

This field is inherited by subtypes together with *tp_getattro* : a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both *NULL*.

setattrfunc **PyTypeObject.tp_setattr**

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_setattro* function, but taking a C string instead of a Python string object to give the attribute name. The signature is

```
PyObject * tp_setattr(PyObject *o, char *attr_name, PyObject *v);
```

The *v* argument is set to *NULL* to delete the attribute. This field is inherited by subtypes together with *tp_setattro* : a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both *NULL*.

*PyAsyncMethods** **tp_as_async**

Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

Nouveau dans la version 3.5 : Formerly known as `tp_compare` and `tp_reserved`.

reprfunc **PyTypeObject.tp_repr**

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for `PyObject_Repr()` ; it must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

This field is inherited by subtypes.

*PyNumberMethods** **tp_as_number**

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in *Number Object Structures*.

The `tp_as_number` field is not inherited, but the contained fields are inherited individually.

*PySequenceMethods** **tp_as_sequence**

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

The `tp_as_sequence` field is not inherited, but the contained fields are inherited individually.

*PyMappingMethods** **tp_as_mapping**

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

The `tp_as_mapping` field is not inherited, but the contained fields are inherited individually.

hashfunc **PyTypeObject.tp_hash**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for `PyObject_Hash()` ; it must return a value of the type Py_hash_t. The value `-1` should not be returned as a normal return value ; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

This field can be set explicitly to `PyObject_HashNotImplemented()` to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to `PyObject_HashNotImplemented()`.

When this field is not set, an attempt to take the hash of the object raises `TypeError`.

This field is inherited by subtypes together with `tp_richcompare` : a subtype inherits both of `tp_richcompare` and `tp_hash`, when the subtype's `tp_richcompare` and `tp_hash` are both *NULL*.

ternaryfunc **PyTypeObject.tp_call**

An optional pointer to a function that implements calling the object. This should be *NULL* if the object is not callable. The signature is the same as for `PyObject_Call()`.

This field is inherited by subtypes.

reprfunc **PyTypeObject.tp_str**

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls `PyObject_Str()` to do the actual work, and `PyObject_Str()` will call this handler.)

The signature is the same as for `PyObject_Str()` ; it must return a string or a Unicode object. This function should return a "friendly" string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

When this field is not set, `PyObject_Repr()` is called to return a string representation.

This field is inherited by subtypes.

getattrofunc **PyTypeObject.tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for `PyObject_GetAttr()`. It is usually convenient to set this field to `PyObject_GenericGetAttr()`, which implements the normal way of looking for object attributes.

This field is inherited by subtypes together with `tp_getattr` : a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype's `tp_getattr` and `tp_getattro` are both *NULL*.

setattrofunc **PyTypeObject.tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for `PyObject_SetAttr()`, but setting *v* to *NULL* to delete an attribute must be supported. It is usually convenient to set this field to `PyObject_GenericSetAttr()`, which implements the normal way of setting object attributes.

This field is inherited by subtypes together with `tp_setattr` : a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both *NULL*.

*PyBufferProcs** **PyTypeObject.tp_as_buffer**

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.

unsigned long **PyTypeObject.tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations ; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer`) that were historically not always present are valid ; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or *NULL* value instead.

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The `Py_TPFLAGS_HAVE_GC` flag bit is inherited together with the `tp_traverse` and `tp_clear` fields, i.e. if the `Py_TPFLAGS_HAVE_GC` flag bit is clear in the subtype and the `tp_traverse` and `tp_clear` fields in the subtype exist and have *NULL* values.

The following bit masks are currently defined ; these can be ORed together using the `|` operator to form the value of the `tp_flags` field. The macro `PyType_HasFeature()` takes a type and a flags value, *tp* and *f*, and checks whether `tp->tp_flags & f` is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap. In this case, the `ob_type` field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes ; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed).

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by `PyType_Ready()`.

Py_TPFLAGS_READYING

This bit is set while `PyType_Ready()` is in the process of initializing the type object.

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using `PyObject_GC_New()` and destroyed using `PyObject_GC_Del()`. More information in section *Supporting Cyclic Garbage Collection*. This bit also implies that the GC-related fields `tp_traverse` and `tp_clear` are present in the type object.

Py_TPFLAGS_DEFAULT
This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits : `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`, `Py_TPFLAGS_HAVE_VERSION_TAG`.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS
These flags are used by functions such as `PyLong_Check()` to quickly determine if a type is a subclass of a built-in type ; such specific checks are faster than a generic check, like `PyObject_IsInstance()`. Custom types that inherit from built-ins should have their `tp_flags` set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE
This bit is set when the `tp_finalize` slot is present in the type structure.

Nouveau dans la version 3.4.

const char* **PyTypeObject.tp_doc**
An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

This field is *not* inherited by subtypes.

traverseproc **PyTypeObject.tp_traverse**
An optional pointer to a traversal function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set. More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

The `tp_traverse` pointer is used by the garbage collector to detect reference cycles. A typical implementation of a `tp_traverse` function simply calls `Py_VISIT()` on each of the instance's members that are Python objects. For example, this is function `local_traverse()` from the `_thread` extension module :

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that `Py_VISIT()` is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be *NULL* or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

Note that `Py_VISIT()` requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names ; don't name them just anything.

This field is inherited by subtypes together with `tp_clear` and the `Py_TPFLAGS_HAVE_GC` flag bit : the flag bit, `tp_traverse`, and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

inquiry **PyTypeObject.tp_clear**
An optional pointer to a clear function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set.

The `tp_clear` member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all `tp_clear` functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a `tp_clear` function. For example, the tuple type does not implement a `tp_clear` function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the `tp_clear` functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing `tp_clear`.

Implementations of `tp_clear` should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to *NULL*, as in the following example :

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The `Py_CLEAR()` macro should be used, because clearing references is delicate : the reference to the contained object must not be decremented until after the pointer to the contained object is set to *NULL*. This is because decrementing the reference count may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be *NULL* at that time, so that *self* knows the contained object can no longer be used. The `Py_CLEAR()` macro performs the operations in a safe order.

Because the goal of `tp_clear` functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's `tp_dealloc` function to invoke `tp_clear`.

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

This field is inherited by subtypes together with `tp_traverse` and the `Py_TPFLAGS_HAVE_GC` flag bit : the flag bit, `tp_traverse`, and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

richcmpfunc **PyTypeObject.tp_richcompare**

An optional pointer to the rich comparison function, whose signature is `PyObject *tp_richcompare(PyObject *a, PyObject *b, int op)`. The first parameter is guaranteed to be an instance of the type that is defined by `PyTypeObject`.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

Note : If you want to implement a type for which only a limited set of comparisons makes sense (e.g. `==` and `!=`, but not `<` and friends), directly raise `TypeError` in the rich comparison function.

This field is inherited by subtypes together with `tp_hash` : a subtype inherits `tp_richcompare` and `tp_hash` when the subtype's `tp_richcompare` and `tp_hash` are both *NULL*.

The following constants are defined to be used as the third argument for `tp_richcompare` and for `PyObject_RichCompare()` :

Constante	Comparaison
`Py_LT`	`<`
`Py_LE`	`<=`
`Py_EQ`	`==`
`Py_NE`	`!=`
`Py_GT`	`>`
`Py_GE`	`>=`

The following macro is defined to ease writing rich comparison functions :

PyObject ***Py_RETURN_RICHCOMPARE**(VAL_A, VAL_B, int *op*)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for `PyObject_RichCompare()`.

The return value's reference count is properly incremented.

On error, sets an exception and returns NULL from the function.

Nouveau dans la version 3.7.

Py_ssize_t **PyTypeObject.tp_weaklistoffset**

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present) ; this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*()` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to *NULL*.

Do not confuse this field with `tp_weaklist` ; that is the list head for weak references to the type object itself.

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset ; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via `tp_weaklistoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's `tp_weaklistoffset`.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

getiterfunc **PyTypeObject.tp_iter**

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function).

This function has the same signature as `PyObject_GetIter()`.

This field is inherited by subtypes.

iternextfunc **PyTypeObject.tp_iternext**

An optional pointer to a function that returns the next item in an iterator. When the iterator is exhausted, it must return *NULL* ; a `StopIteration` exception may or may not be set. When another error occurs, it must return *NULL* too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the `tp_iter` function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as `PyIter_Next()`.

This field is inherited by subtypes.

struct *PyMethodDef** **PyTypeObject.tp_methods**

An optional pointer to a static *NULL*-terminated array of `PyMethodDef` structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a method descriptor.

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef** **PyTypeObject.tp_members**

An optional pointer to a static *NULL*-terminated array of `PyMemberDef` structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a member descriptor.

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef** **PyTypeObject.tp_getset**

An optional pointer to a static *NULL*-terminated array of `PyGetSetDef` structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a getset descriptor.

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

*PyTypeObject** **PyTypeObject.tp_base**

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported ; multiple inheritance require dynamically creating a type object by calling the metatype.

This field is not inherited by subtypes (obviously), but it defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

*PyObject** **PyTypeObject.tp_dict**

The type's dictionary is stored here by `PyType_Ready()`.

This field should normally be initialized to *NULL* before PyType_Ready is called ; it may also be initialized to a dictionary containing initial attributes for the type. Once `PyType_Ready()` has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

> **Avertissement :** It is not safe to use `PyDict_SetItem()` on or otherwise modify `tp_dict` with the dictionary C-API.

descrgetfunc **PyTypeObject.tp_descr_get**

An optional pointer to a "descriptor get" function.

The function signature is

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

This field is inherited by subtypes.

descrsetfunc **PyTypeObject.tp_descr_set**

An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to *NULL* to delete the value. This field is inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_dictoffset**

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary ; this offset is used by `PyObject_GenericGetAttr()`.

Do not confuse this field with `tp_dict` ; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the *tp_basicsize* field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, *tp_dictoffset* should be set to `-4` to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative *tp_dictoffset* as follows :

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where *tp_basicsize*, *tp_itemsize* and *tp_dictoffset* are taken from the type object, and `ob_size` is taken from the instance. The absolute value is taken because ints use the sign of `ob_size` to store the sign of the number. (There's never a need to do this calculation yourself ; it is done for you by `_PyObject_GetDictPtr()`.)

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset ; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via *tp_dictoffset*, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the *tp_dictoffset* is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its *tp_dictoffset* from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

initproc **PyTypeObject.tp_init**

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds)
```

The self argument is the instance to be initialized ; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The *tp_init* function, if not *NULL*, is called when an instance is created normally by calling its type, after the type's *tp_new* function has returned an instance of the type. If the *tp_new* function returns an instance of some other type that is not a subtype of the original type, no *tp_init* function is called ; if *tp_new* returns an instance of a subtype of the original type, the subtype's *tp_init* is called.

This field is inherited by subtypes.

allocfunc **PyTypeObject.tp_alloc**

An optional pointer to an instance allocation function.

The function signature is

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems)
```

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with `ob_refcnt` set to `1` and `ob_type` set to the type argument. If the type's *tp_itemsize* is non-zero, the object's `ob_size` field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)` ; otherwise, *nitems* is not used and the length of the block should be *tp_basicsize*.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by `tp_new`.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to `PyType_GenericAlloc()`, to force a standard heap allocation strategy. That is also the recommended value for statically defined types.

newfunc **PyTypeObject.tp_new**

An optional pointer to an instance creation function.

If this function is *NULL* for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

The function signature is

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
```

The subtype argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that subtype doesn't have to equal the type whose `tp_new` function is called; it may be a subtype of that type (but not an unrelated type).

The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.

This field is inherited by subtypes, except it is not inherited by static types whose `tp_base` is *NULL* or `&PyBaseObject_Type`.

destructor **PyTypeObject.tp_free**

An optional pointer to an instance deallocation function. Its signature is `freefunc`:

```
void tp_free(void *)
```

An initializer that is compatible with this signature is `PyObject_Free()`.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match `PyType_GenericAlloc()` and the value of the `Py_TPFLAGS_HAVE_GC` flag bit.

inquiry **PyTypeObject.tp_is_gc**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's `tp_flags` field, and check the `Py_TPFLAGS_HAVE_GC` flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is

```
int tp_is_gc(PyObject *self)
```

(The only example of this are types themselves. The metatype, `PyType_Type`, defines this function to distinguish between statically and dynamically allocated types.)

This field is inherited by subtypes.

*PyObject** **PyTypeObject.tp_bases**

Tuple of base types.

This is set for types created by a class statement. It should be *NULL* for statically defined types.

This field is not inherited.

*PyObject** **PyTypeObject.tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field is not inherited; it is calculated fresh by `PyType_Ready()`.

destructor **PyTypeObject.tp_finalize**

An optional pointer to an instance finalization function. Its signature is `destructor` :

```
void tp_finalize(PyObject *)
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

tp_finalize should not mutate the current exception status ; therefore, a recommended way to write a non-trivial finalizer is :

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

For this field to be taken into account (even through inheritance), you must also set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit.

This field is inherited by subtypes.

Nouveau dans la version 3.4.

Voir aussi :

"Safe object finalization" (**PEP 442**)

*PyObject** **PyTypeObject.tp_cache**

Unused. Not inherited. Internal use only.

*PyObject** **PyTypeObject.tp_subclasses**

List of weak references to subclasses. Not inherited. Internal use only.

*PyObject** **PyTypeObject.tp_weaklist**

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro `COUNT_ALLOCS` is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_allocs**

Number of allocations.

Py_ssize_t **PyTypeObject.tp_frees**

Number of frees.

Py_ssize_t **PyTypeObject.tp_maxalloc**

Maximum simultaneously allocated objects.

*PyTypeObject** **PyTypeObject.tp_next**

Pointer to the next type object with a non-zero *tp_allocs* field.

Also, note that, in a garbage collected Python, tp_dealloc may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which tp_dealloc is called will own the

Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called tp_dealloc will not violate any assumptions of the library.

11.4 Number Object Structures

PyNumberMethods

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the *Number Protocol* section.

Here is the structure definition :

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
     unaryfunc nb_int;
     void *nb_reserved;
     unaryfunc nb_float;

     binaryfunc nb_inplace_add;
     binaryfunc nb_inplace_subtract;
     binaryfunc nb_inplace_multiply;
     binaryfunc nb_inplace_remainder;
     ternaryfunc nb_inplace_power;
     binaryfunc nb_inplace_lshift;
     binaryfunc nb_inplace_rshift;
     binaryfunc nb_inplace_and;
     binaryfunc nb_inplace_xor;
     binaryfunc nb_inplace_or;

     binaryfunc nb_floor_divide;
     binaryfunc nb_true_divide;
     binaryfunc nb_inplace_floor_divide;
     binaryfunc nb_inplace_true_divide;

     unaryfunc nb_index;

     binaryfunc nb_matrix_multiply;
     binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

Note : Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

Note : The `nb_reserved` field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

11.5 Mapping Object Structures

`PyMappingMethods`
This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members :

lenfunc **`PyMappingMethods.mp_length`**
This function is used by *`PyMapping_Size()`* and *`PyObject_Size()`*, and has the same signature. This slot may be set to *NULL* if the object has no defined length.

binaryfunc **`PyMappingMethods.mp_subscript`**
This function is used by *`PyObject_GetItem()`* and *`PySequence_GetSlice()`*, and has the same signature as `PyObject_GetItem()`. This slot must be filled for the *`PyMapping_Check()`* function to return `1`, it can be *NULL* otherwise.

objobjargproc **`PyMappingMethods.mp_ass_subscript`**
This function is used by *`PyObject_SetItem()`*, *`PyObject_DelItem()`*, `PyObject_SetSlice()` and `PyObject_DelSlice()`. It has the same signature as `PyObject_SetItem()`, but *v* can also be set to *NULL* to delete an item. If this slot is *NULL*, the object does not support item assignment and deletion.

11.6 Sequence Object Structures

`PySequenceMethods`
This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc **`PySequenceMethods.sq_length`**
This function is used by *`PySequence_Size()`* and *`PyObject_Size()`*, and has the same signature. It is also used for handling negative indices via the *`sq_item`* and the *`sq_ass_item`* slots.

binaryfunc **`PySequenceMethods.sq_concat`**
This function is used by *`PySequence_Concat()`* and has the same signature. It is also used by the + operator, after trying the numeric addition via the `nb_add` slot.

ssizeargfunc **`PySequenceMethods.sq_repeat`**
This function is used by *`PySequence_Repeat()`* and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the `nb_multiply` slot.

ssizeargfunc **`PySequenceMethods.sq_item`**
This function is used by *`PySequence_GetItem()`* and has the same signature. It is also used by *`PyObject_GetItem()`*, after trying the subscription via the *`mp_subscript`* slot. This slot must be filled for the *`PySequence_Check()`* function to return `1`, it can be *NULL* otherwise.

Negative indexes are handled as follows : if the `sq_length` slot is filled, it is called and the sequence length is used to compute a positive index which is passed to `sq_item`. If `sq_length` is *NULL*, the index is passed as is to the function.

ssizeobjargproc **PySequenceMethods.sq_ass_item**

This function is used by *PySequence_SetItem()* and has the same signature. It is also used by *PyObject_SetItem()* and *PyObject_DelItem()*, after trying the item assignment and deletion via the *mp_ass_subscript* slot. This slot may be left to *NULL* if the object does not support item assignment and deletion.

objobjproc **PySequenceMethods.sq_contains**

This function may be used by *PySequence_Contains()* and has the same signature. This slot may be left to *NULL*, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

binaryfunc **PySequenceMethods.sq_inplace_concat**

This function is used by *PySequence_InPlaceConcat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to *NULL*, in this case `PySequence_InPlaceConcat()` will fall back to *PySequence_Concat()*. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the `nb_inplace_add` slot.

ssizeargfunc **PySequenceMethods.sq_inplace_repeat**

This function is used by *PySequence_InPlaceRepeat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to *NULL*, in this case `PySequence_InPlaceRepeat()` will fall back to *PySequence_Repeat()*. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the `nb_inplace_multiply` slot.

11.7 Buffer Object Structures

PyBufferProcs

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc **PyBufferProcs.bf_getbuffer**

The signature of this function is :

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps :

(1) Check if the request can be met. If not, raise `PyExc_BufferError`, set `view->obj` to *NULL* and return `-1`.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) Return `0`.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used :

— Re-export : Each member of the tree acts as the exporting object and sets `view->obj` to a new reference to itself.

— Redirect : The buffer request is redirected to the root object of the tree. Here, `view->obj` will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the *Py_buffer* structure belongs to the exporter and must remain valid until there are no consumers left. *format*, *shape*, *strides*, *suboffsets* and *internal* are read-only for the consumer.

PyBuffer_FillInfo() provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

PyObject_GetBuffer() is the interface for the consumer that wraps this function.

releasebufferproc **PyBufferProcs.bf_releasebuffer**

The signature of this function is :

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, *PyBufferProcs.bf_releasebuffer* may be *NULL*. Otherwise, a standard implementation of this function will take these optional steps :

(1) Decrement an internal counter for the number of exports.

(2) If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the *internal* field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement `view->obj`, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

PyBuffer_Release() is the interface for the consumer that wraps this function.

11.8 Async Object Structures

Nouveau dans la version 3.5.

PyAsyncMethods

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition :

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
} PyAsyncMethods;
```

unaryfunc **PyAsyncMethods.am_await**

The signature of this function is :

```
PyObject *am_await(PyObject *self)
```

The returned object must be an iterator, i.e. *PyIter_Check()* must return `1` for it.

This slot may be set to *NULL* if an object is not an *awaitable*.

unaryfunc **PyAsyncMethods.am_aiter**

The signature of this function is :

```
PyObject *am_aiter(PyObject *self)
```

Must return an *awaitable* object. See `__anext__()` for details.

This slot may be set to *NULL* if an object does not implement asynchronous iteration protocol.

unaryfunc **PyAsyncMethods.am_anext**

The signature of this function is :

```
PyObject *am_anext(PyObject *self)
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to *NULL*.

11.9 Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

`Py_TPFLAGS_HAVE_GC`
: Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules :

1. The memory for the object must be allocated using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

TYPE* **`PyObject_GC_New`**(TYPE, *PyTypeObject* **type*)
: Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **`PyObject_GC_NewVar`**(TYPE, *PyTypeObject* **type*, Py_ssize_t *size*)
: Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **`PyObject_GC_Resize`**(TYPE, *PyVarObject* **op*, Py_ssize_t *newsize*)
: Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or *NULL* on failure. *op* must not be tracked by the collector yet.

void **`PyObject_GC_Track`**(*PyObject* **op*)
: Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

void **`_PyObject_GC_TRACK`**(*PyObject* **op*)
: A macro version of `PyObject_GC_Track()`. It should not be used for extension modules.

  Obsolète depuis la version 3.6 : This macro is removed from Python 3.8.

Similarly, the deallocator for the object must conform to a similar pair of rules :

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.
2. The object's memory must be deallocated using `PyObject_GC_Del()`.

void **`PyObject_GC_Del`**(void **op*)
: Releases memory allocated to an object using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.

void **`PyObject_GC_UnTrack`**(void **op*)
: Remove the object *op* from the set of container objects tracked by the collector. Note that `PyObject_GC_Track()` can be called again on this object to add it back to the set of tracked objects. The deallocator (`tp_dealloc` handler) should call this for the object before any of the fields used by the `tp_traverse` handler become invalid.

void **`_PyObject_GC_UNTRACK`**(*PyObject* **op*)
: A macro version of `PyObject_GC_UnTrack()`. It should not be used for extension modules.

  Obsolète depuis la version 3.6 : This macro is removed from Python 3.8.

The `tp_traverse` handler accepts a function parameter of this type :

int **(*visitproc)**(*PyObject *object*, void **arg*)
Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection ; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type :

int **(*traverseproc)**(*PyObject *self*, *visitproc visit*, void **arg*)
Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a *NULL* object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg* :

void **Py_VISIT**(*PyObject *o*)
If *o* is not *NULL*, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like :

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The `tp_clear` handler must be of the `inquiry` type, or *NULL* if the object is immutable.

int **(*inquiry)**(*PyObject *self*)
Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

CHAPITRE 12

Version des API et ABI

PY_VERSION_HEX est le numéro de version de Python encodé en un seul entier.

Par exemple si le PY_VERSION_HEX` est défini à ``0x030401a2, la version d'information sous-jacente peut être trouvée en la traitant comme un nombre sous 32 bits de la manière suivante :

Bytes	Bits (ordre gros-boutiste)	Signification
1	1-8	PY_MAJOR_VERSION (le 3 dans 3.4.1a2)
2	9-16	PY_MINOR_VERSION (le 4 dans 3.4.1a2)
3	17-24	PY_MICRO_VERSION (le 1 dans 3.4.1a2)
4	25-28	PY_RELEASE_LEVEL (0xA pour alpha, 0xB pour bêta, 0xC pour une version candidate et 0xF pour final), dans ce cas c'est alpha.
	29-32	PY_RELEASE_SERIAL (le 2 au 3.4.1a2, zéro pour des versions finales)

Ainsi 3.4.1a2 est une hexane-version 0x030401a2.

Toutes les macros données sont définies dans Include/patchlevel.h.

ANNEXE A

Glossaire

>>> 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

objet octet-compatible Un objet gérant les *Protocole tampon* et pouvant exporter un tampon (*buffer* en anglais) C-*contiguous*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets `memoryview`. Les objets bytes-compatibles peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

***docstring* (chaîne de documentation)** Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elles 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 cancane, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes de base abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

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

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

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

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

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

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

objet fichier-compatible Synonyme de *objet fichier*.

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

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

Voir les **PEP 302**, **PEP 420** et **PEP 451** pour plus de détails.

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

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

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

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

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

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

Voir *variable annotation* et :pep :' 484', qui décrivent cette fonctionnalité.

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

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

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

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

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

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

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

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

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

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

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

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

GIL Voir *global interpreter lock*.

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

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

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

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

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

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

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

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

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

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

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*.

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

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

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

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

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

Vous trouverez davantage d'informations dans typeiter.

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

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

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

argument nommé Voir *argument*.

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

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

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

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

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

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

méthode magique Un synonyme informel de *special method*.

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

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

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

métaclasse 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 métaclasse 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 nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

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

MRO Voir *ordre de résolution des méthodes*.

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

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

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

Voir aussi *module*.

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

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

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

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

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

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

— *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, *foo* et *bar* dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

— *positional-only* : 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.

— *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (`*`) seule dans la liste des paramètres avant eux. Par exemple, *kw_only1* et *kw_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

— *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une `*`. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

— *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par `**`. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

Voir aussi *argument* dans le glossaire, la question sur la différence entre les arguments et les paramètres dans la FAQ, la classe `inspect.Parameter`, la section function et la **PEP 362**.

entrée de chemin Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

chercheur de chemins *chercheur* renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

point d'entrée pour la recherche dans *path* Appelable dans la liste `sys.path_hook` qui donne un *chercheur d'entrée dans path* s'il sait où trouver des modules pour une *entrée dans path* donnée.

chercheur basé sur les chemins L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

objet simili-chemin Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet `str` ou un objet `bytes` représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin `str` ou `bytes` du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type `str` ou `bytes` à la place. A été Introduit par la **PEP 519**.

PEP *Python Enhancement Proposal* (Proposition d'amélioration Python). Un PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

Les PEPs sont censés être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

Voir **PEP 1**.

portion Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la **PEP 420**.

argument positionnel Voir *argument*.

API provisoire Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changement cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

Même pour les API provisoires, les changement cassant la rétrocompatibilité sont considérées comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problème en conservant la rétrocompatibilité.

Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la **PEP 411** pour plus de détails.

paquet provisoire Voir *provisional API*.

Python 3000 Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

Pythonique Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

```
for i in range(len(food)):
    print(food[i])
```

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print(piece)
```

nom qualifié Nom, comprenant des points, montrant le "chemin" de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la **PEP 3155**. Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name - FQN* en anglais) signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

nombre de références Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Le module `sys` définit une fonction `getrefcount()` que les développeurs peuvent utiliser pour obtenir le nombre de références à un objet donné.

paquet classique *paquet* traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.

Voir aussi *paquet-espace de nommage*.

`__slots__` Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

séquence *itérable* qui offre un accès efficace à ses éléments par un indice sous forme de nombre entier via la méthode spéciale `__getitem__()` et qui définit une méthode `__len__()` donnant sa taille. Voici quelques séquences natives : `list`, `str`, `tuple`, et `bytes`. Notez que `dict` possède aussi une méthode `__getitem__()` et une méthode `__len__()`, mais il est considéré comme un *mapping* plutôt qu'une séquence, car ses accès se font par une clé arbitraire *immuable* plutôt qu'un nombre entier.

La classe abstraite de base `collections.abc.Sequence` définit une interface plus riche qui va au-delà des simples `__getitem__()` et `__len__()`, en ajoutant `count()`, `index()`, `__contains__()` et `__reversed__()`. Les types qui implémentent cette interface étendue peuvent s'enregistrer explicitement en utilisant `register()`.

distribution simple Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

tranche (*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets `slice` en interne.

méthode spéciale (*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans specialnames.

instruction Une instruction (*statement* en anglais) est un composant d'un "bloc" de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

struct sequence Un n-uplet (*tuple* en anglais) dont les éléments sont nommés. Les *struct sequences* exposent une interface similaire au *n-uplet nommé* car on peut accéder à leurs éléments par un nom d'attribut ou par un indice. Cependant, elles n'ont aucune des méthodes du *n-uplet nommé* : ni `collections.somenamedtuple._make()` ou `_asdict()`. Par exemple `sys.float_info` ou les valeurs données par `os.stat()` sont des *struct sequence*.

encodage de texte Codec (codeur-décodeur) qui convertit des chaînes de caractères Unicode en octets (classe *bytes*).

fichier texte *file object* capable de lire et d'écrire des objets `str`. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*text encoding* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte (`'r'` ou `'w'`), `sys.stdin`, `sys.stdout` et les instances de `io.StringIO`.

Voir aussi *binary file* pour un objet fichier capable de lire et d'écrire *bytes-like objects*.

chaîne entre triple guillemets Chaîne qui est délimitée par trois guillemets simples (`'`) ou trois guillemets doubles (`"`). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un `\`. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

type Le type d'un objet Python détermine quel genre d'objet c'est. Tous les objets ont un type. Le type d'un objet peut être obtenu via son attribut `__class__` ou via `type(obj)`.

alias de type Synonyme d'un type, créé en affectant le type à un identifiant.

Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Voir `typing` et **PEP 484**, qui décrivent cette fonctionnalité.

indication de type Le *annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Les indications de type sont facultatifs et ne sont pas indispensables à l'interpréteur Python, mais ils sont utiles aux outils d'analyse de type statique et aident les IDE à compléter et à réusiner (*code refactoring* en anglais) le code.

Les indicateurs de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultés en utilisant `typing.get_type_hints()`.

Voir `typing` et **PEP 484**, qui décrivent cette fonctionnalité.

retours à la ligne universels Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la **PEP 278** et la **PEP 3116**, ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

annotation de variable *annotation* d'une variable ou d'un attribut de classe.

Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative

```
class C:
    field: 'annotation'
```

Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int`

```
count: int = 0
```

La syntaxe d'annotation de la variable est expliquée dans la section annassign.

Reportez-vous à *function annotation*, à la :pep :' 484' et à la **PEP 526** qui décrivent cette fonctionnalité.

environnement virtuel Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

Voir aussi `venv`.

machine virtuelle Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *bytecode* produit par le compilateur de *bytecode*.

Le zen de Python Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant "`import this`" dans une invite Python interactive.

ANNEXE B

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 !

ANNEXE C

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 Viriginie, d'où il diffusa plusieurs versions du logiciel.

En mai 2000, Guido et l'équipe de développement centrale de Python sont partis vers BeOpen.com pour former l'équipe BeOpen PythonLabs. En octobre de la même année, l'équipe de PythonLabs est partie vers Digital Creations (désormais Zope Corporation ; voir 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 ultérieure	2.1.1	2001-maintenant	PSF	oui

Note : Compatible GPL ne signifie pas que nous distribuons Python sous licence GPL. Toutes les licences Python, excepté la licence GPL, vous permettent la distribution d'une version modifiée sans rendre open source ces changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non-compatibles GPL » ne le peuvent pas.

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

C.2 Conditions générales pour accéder à, ou utiliser, Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.4

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.7.4 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.7.4 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 3.7.4 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.7.4 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.7.4.

4. PSF is making Python 3.7.4 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.7.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.4
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.4, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

```
6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.7.4, 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
   third party.  As an exception, the "BeOpen Python" logos available at
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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
   1.6.1 that incorporate non-separable material that was previously distributed
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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 tiers

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

C.3.1 Mersenne twister

Le module `_random` inclut du code construit à partir d'un téléchargement depuis http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. Voici mot pour mot les commentaires du code original :

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).
```

(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 Interfaces de connexion asynchrones

Les modules `asynchat` et `asyncore` contiennent la note suivante :

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

(suite sur la page suivante)

(suite de la page précédente)

```
Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 Les fonctions UUencode et UUdecode

Le module `uu` contient la note suivante :

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
```

(suite sur la page suivante)

(suite de la page précédente)

```
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module `xmlrpc.client` contient la note suivante :

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY 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.3.8 test_epoll

Le module `test_epoll` contient la note suivante :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.
```

(suite sur la page suivante)

(suite de la page précédente)

C.3.9 Select kqueue

Le module `select` contient la note suivante pour l'interface *kqueue* :

C.3.10 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 *strtod* et *dtoa*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversion de *doubles* C vers et depuis les chaînes, est tiré d'un fichier du même nom par David M. Gay, actuellement disponible sur http://www.netlib.org/fp/. Le fichier original, tel que récupéré le 16 mars 2009, contient la licence suivante :

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et Mac OS X peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * 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.3.13 expat

Le module `pyexpat` est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

(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.3.15 zlib

Le module `zlib` est compilé en utilisant une copie du code source de *zlib* si la version de *zlib* trouvée sur le système est trop vieille pour être utilisée :

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet *cfuhash* :

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

(suite sur la page suivante)

(suite de la page précédente)

```
  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

Le module `_decimal` est construit en incluant une copie de la bibliothèque *libmpdec*, sauf si elle est compilée avec `--with-system-libmpdec` :

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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.
```

ANNEXE D

Copyright

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

Index

Non-alphabetical

A

B

C

R

S