Using the C API: Assorted topics

- Errors and Exceptions
- Embedding an extension
- Calling Python Functions from C
- Extracting Parameters in Extension Functions
- Keyword Parameters for Extension Functions
- Building Arbitrary Values
- Reference Counts
  - Reference Counting in Python
  - Ownership Rules
  - Thin Ice
  - NULL Pointers
- Writing Extensions in C++
- Providing a C API for an Extension Module

The `tutorial <first-extension-module>` walked you through creating a C API extension module, but left many areas unexplained. This document looks at several concepts that you'll need to learn in order to write more complex extensions.

Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually `-1` or a `NULL` pointer). Exception information is stored in three members of the interpreter's thread state. These are `NULL` if there is no exception. Otherwise they are the C equivalents of the members of the Python tuple returned by `sys.exc_info`. These are the exception type, exception instance, and a traceback object. It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the "associated value" of the exception.

Another useful function is `PyErr_SetFromErrno`, which only takes an exception argument and constructs the associated value by inspection of the global variable `errno`. The most general function is `PyErr_SetObject`, which takes two object arguments, the exception and its associated value. You don't need to `Py_INCREF` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with `PyErr_Occurred`. This returns the current exception object, or `NULL` if no exception has occurred. You normally don't need to call `PyErr_Occurred` to see whether an error occurred in a function call, since you should be able to tell from the return value.

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

(There are situations where a module can actually give a more detailed error message by calling another `PyErr_*` function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling `PyErr_Clear`. The only time C code should call `PyErr_Clear` is if it doesn't want to pass the error on to the interpreter but wants to handle it completely by itself (possibly by trying something else, or pretending nothing went wrong).

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

Also note that, with the important exception of `PyArg_ParseTuple` and friends, functions that return an integer status usually return a positive value or zero for success and `-1` for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF` or `Py_DECREF` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely --- don't use `PyExc_TypeError` to mean that a file couldn't be opened (that should probably be `PyExc_OSError`). If something's wrong with the argument list, the `PyArg_ParseTuple` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. The simplest way to do this is to declare a static global object variable at the beginning of the file:

```
static PyObject *SpamError = NULL;
```

and initialize it by calling `PyErr_NewException` in the module's `Py_mod_exec` function (`!spam_module_exec`):

```
SpamError = PyErr_NewException("spam.error", NULL, NULL);
```

Since `!SpamError` is a global variable, it will be overwritten every time the module is reinitialized, when the `Py_mod_exec` function is called.

For now, let's avoid the issue: we will block repeated initialization by raising an `ImportError`:

```
static PyObject *SpamError = NULL;

static int
spam_module_exec(PyObject *m)
{
    if (SpamError != NULL) {
        PyErr_SetString(PyExc_ImportError,
                        "cannot initialize spam module more than once");
        return -1;
    }
    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    if (PyModule_AddObjectRef(m, "SpamError", SpamError) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot spam_module_slots[] = {
```

```
    {Py_mod_exec, spam_module_exec},
    {0, NULL}
};

static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    .m_size = 0,  // non-negative
    .m_slots = spam_module_slots,
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

Note that the Python name for the exception object is `!spam.error`. The `PyErr_NewException` function may create a class with the base class being `Exception` (unless another class is passed in instead of `NULL`), described in `bltin-exceptions`.

Note also that the `!SpamError` variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing `!SpamError` to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

For now, the `Py_DECREF` call to remove this reference is missing. Even when the Python interpreter shuts down, the global `!SpamError` variable will not be garbage-collected. It will "leak". We did, however, ensure that this will happen at most once per process.

We discuss the use of `PyMODINIT_FUNC` as a function return type later in this sample.

The `!spam.error` exception can be raised in your extension module using a call to `PyErr_SetString` as shown below:

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

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

Embedding an extension

If you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. On Unix, place your file (`spammodule.c` for example) in the `Modules/` directory of an unpacked source distribution, add a line to the file `Modules/Setup.local` describing your file:

```
spam spammodule.o
```

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

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

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

Calling Python Functions from C

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

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (If you're interested in how to call the Python parser with a particular string as input, see `veryhigh`.)

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

```
static PyObject *my_callback = NULL;

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

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

This function must be registered with the interpreter using the `METH_VARARGS` flag in `PyMethodDef.ml_flags`. The `PyArg_ParseTuple` function and its arguments are documented in section `parsetuple`.

The macros `Py_XINCREF` and `Py_XDECREF` increment/decrement the reference count of an object and are safe in the presence of `NULL` pointers (but note that *temp* will not be `NULL` in this context). More info on them in section `refcounts`.

single: PyObject_CallObject (C function)

Later, when it is time to call the function, you call the C function `PyObject_CallObject`. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in `NULL`, or an empty tuple; to call

it with one argument, pass a singleton tuple. `Py_BuildValue` returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

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

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

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

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

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

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyObject_CallObject`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue`. For example, if you want to pass an integral event code, you might use the following code:

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

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check!

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

```
PyObject *arglist, *dict;
...
arglist = PyTuple_New(0);
if (arglist == NULL)
    return NULL;
dict = Py_BuildValue("{s:i}", "name", val);
if (dict == NULL) {
    Py_DECREF(arglist);
    return NULL;
}
result = PyObject_Call(my_callback, arglist, dict);
Py_DECREF(arglist);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

single: PyArg_ParseTuple (C function)

Extracting Parameters in Extension Functions

The `tutorial <first-extension-module>` uses a "`METH_O`" function, which is limited to a single Python argument. If you want more, you can use `METH_VARARGS` instead. With this flag, the C function will receive a `tuple` of arguments instead of a single object.

For unpacking the tuple, CPython provides the `PyArg_ParseTuple` function, declared as follows:

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

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

For example, to receive a single Python `str` object and turn it into a C buffer, you would use `"s"` as the format string:

```
const char *command;
if (!PyArg_ParseTuple(args, "s", &command)) {
    return NULL;
}
```

If an error is detected in the argument list, `!PyArg_ParseTuple` returns `NULL` (the error indicator for functions returning object pointers); your function may return `NULL`, relying on the exception set by `PyArg_ParseTuple`.

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

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

Some example calls:

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

```
int ok;
int i, j;
```

```
long k, l;
const char *s;
Py_ssize_t size;

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

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

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

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

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

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

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

Keyword Parameters for Extension Functions

If you also want your function to accept `keyword arguments <keyword argument>`, use the `METH_KEYWORDS` flag in combination with `METH_VARARGS`. (`!METH_KEYWORDS` can also be used with other flags; see its documentation for the allowed combinations.)

In this case, the C function should accept a third `PyObject *` parameter which will be a dictionary of keywords. Use `PyArg_ParseTupleAndKeywords` to parse the arguments to such a function.

single: PyArg_ParseTupleAndKeywords (C function)

The `PyArg_ParseTupleAndKeywords` function is declared as follows:

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

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

> Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the *kwlist* will cause `TypeError` to be raised.

single: Philbrick, Geoff

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

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

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

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

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

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

    Py_RETURN_NONE;
}

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

Building Arbitrary Values

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

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

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

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

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

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

Reference Counts

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

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

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

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

An alternative strategy is called `automatic garbage collection`. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence the use of "automatic" to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn't need to call `free` explicitly. (Another claimed advantage is an improvement in speed or memory usage --- this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc` and `free` are available --- which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

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

The cycle detector is able to detect garbage cycles and can reclaim them. The `gc` module exposes a way to run the detector (the `~gc.collect` function), as well as configuration interfaces and the

ability to disable the detector at runtime.

Reference Counting in Python

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

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

It is also possible to `borrow`[^1] a reference to an object. The borrower of a reference should not call `Py_DECREF`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely[^2].

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

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

Ownership Rules

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

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

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

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

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

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

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

Thin Ice

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

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

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

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

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

Let's follow the control flow into `PyList_SetItem`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `!__del__` method. If this class instance has a reference count of 1, disposing of it will call its `!__del__` method. Internally, `PyList_SetItem` calls `Py_DECREF` on the replaced item, which invokes replaced item's corresponding `~PyTypeObject.tp_dealloc` function. During deallocation, `~PyTypeObject.tp_dealloc` calls `~PyTypeObject.tp_finalize`, which is mapped to the `!__del__` method for class instances (see `442`). This entire sequence happens synchronously within the `PyList_SetItem` call.

Since it is written in Python, the `!__del__` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `!bug`? You bet! Assuming that the list passed into `!bug` is accessible to the `!__del__` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

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

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

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

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

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

```
void
bug(PyObject *list)
```

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

NULL Pointers

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

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

The macros `Py_INCREF` and `Py_DECREF` do not check for `NULL` pointers --- however, their variants `Py_XINCREF` and `Py_XDECREF` do.

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

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never `NULL` --- in fact it guarantees that it is always a tuple[^3].

It is a severe error to ever let a `NULL` pointer "escape" to the Python user.

Writing Extensions in C++

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

recent C++ compilers define this symbol).

Providing a C API for an Extension Module

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

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

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

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

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

Whichever method you choose, it's important to name your Capsules properly. The function `PyCapsule_New` takes a name parameter (`const char \*`); you're permitted to pass in a `NULL` name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of

runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

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

```
modulename.attributename
```

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

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

The exporting module is a modification of the `!spam` module from the `tutorial <first-extension-module>`. The function `!spam.system` does not call the C library function `system` directly, but a function `!PySpam_System`, which would of course do something more complicated in reality (such as adding "spam" to every command). This function `!PySpam_System` is also exported to other extension modules.

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

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

The function `!spam_system` is modified in a trivial way:

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

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

```
}
```

In the beginning of the module, right after the line :

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

two more lines must be added:

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

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

```
static int
spam_module_exec(PyObject *m)
{
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

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

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

    if (PyModule_Add(m, "_C_API", c_api_object) < 0) {
        return -1;
    }

    return 0;
}
```

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

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

```
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
```

```
extern "C" {
#endif

/* Header file for spammodule */

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

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



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

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

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

static void **PySpam_API;

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

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

#endif

#ifdef __cplusplus
```

```
}
#endif


#endif /* !defined(Py_SPAMMODULE_H) */
```

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

```
static int
client_module_exec(PyObject *m)
{
    if (import_spam() < 0) {
        return -1;
    }
    /* additional initialization can happen here */
    return 0;
}
```

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

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

Footnotes

[^1]: The metaphor of "borrowing" a reference is not completely correct: the owner still has a copy of the reference.

[^2]: Checking that the reference count is at least 1 **does not work** --- the reference count itself could be in freed memory and may thus be reused for another object!

[^3]: These guarantees don't hold when you use the "old" style calling convention --- this is still found in much existing code.

□□□ #2
□ □□□ □□□ 2026-10-31 16:57:04 UTC
□ □□□ □□□ 2026-10-31 16:57:11 UTC