# Isolating Extension Modules

***Release 3.12.11***

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

**September 29, 2025**

**Python Software Foundation**
**Email: docs@python.org**

## Contents

**1 Who should read this** **2**

**2 Background** **2**
2.1 Enter Per-Module State . . . 2
2.2 Isolated Module Objects . . . 2
2.3 Surprising Edge Cases . . . 3

**3 Making Modules Safe with Multiple Interpreters** **3**
3.1 Managing Global State . . . 3
3.2 Managing Per-Module State . . . 3
3.3 Opt-Out: Limiting to One Module Object per Process . . . 4
3.4 Module State Access from Functions . . . 4

**4 Heap Types** **4**
4.1 Changing Static Types to Heap Types . . . 5
4.2 Defining Heap Types . . . 5
4.3 Garbage-Collection Protocol . . . 5
4.4 Module State Access from Classes . . . 7
4.5 Module State Access from Regular Methods . . . 7
4.6 Module State Access from Slot Methods, Getters and Setters . . . 8
4.7 Lifetime of the Module State . . . 9

**5 Open Issues** **9**
5.1 Per-Class Scope . . . 9
5.2 Lossless Conversion to Heap Types . . . 9

**Abstract**

Traditionally, state belonging to Python extension modules was kept in C `static` variables, which have process-wide scope. This document describes problems of such per-process state and shows a safer way: per-module state.

The document also describes how to switch to per-module state where possible. This transition involves allocating space for that state, potentially switching from static types to heap types, and—perhaps most importantly—accessing per-module state from code.

# 1 Who should read this

This guide is written for maintainers of C-API extensions who would like to make that extension safer to use in applications where Python itself is used as a library.

# 2 Background

An *interpreter* is the context in which Python code runs. It contains configuration (e.g. the import path) and runtime state (e.g. the set of imported modules).

Python supports running multiple interpreters in one process. There are two cases to think about—users may run interpreters:

- in sequence, with several `Py_InitializeEx()`/`Py_FinalizeEx()` cycles, and
- in parallel, managing "sub-interpreters" using `Py_NewInterpreter()`/`Py_EndInterpreter()`.

Both cases (and combinations of them) would be most useful when embedding Python within a library. Libraries generally shouldn't make assumptions about the application that uses them, which include assuming a process-wide "main Python interpreter".

Historically, Python extension modules don't handle this use case well. Many extension modules (and even some stdlib modules) use *per-process* global state, because C `static` variables are extremely easy to use. Thus, data that should be specific to an interpreter ends up being shared between interpreters. Unless the extension developer is careful, it is very easy to introduce edge cases that lead to crashes when a module is loaded in more than one interpreter in the same process.

Unfortunately, *per-interpreter* state is not easy to achieve. Extension authors tend to not keep multiple interpreters in mind when developing, and it is currently cumbersome to test the behavior.

## 2.1 Enter Per-Module State

Instead of focusing on per-interpreter state, Python's C API is evolving to better support the more granular *per-module* state. This means that C-level data should be attached to a *module object*. Each interpreter creates its own module object, keeping the data separate. For testing the isolation, multiple module objects corresponding to a single extension can even be loaded in a single interpreter.

Per-module state provides an easy way to think about lifetime and resource ownership: the extension module will initialize when a module object is created, and clean up when it's freed. In this regard, a module is just like any other `PyObject*`; there are no "on interpreter shutdown" hooks to think—or forget—about.

Note that there are use cases for different kinds of "globals": per-process, per-interpreter, per-thread or per-task state. With per-module state as the default, these are still possible, but you should treat them as exceptional cases: if you need them, you should give them additional care and testing. (Note that this guide does not cover them.)

## 2.2 Isolated Module Objects

The key point to keep in mind when developing an extension module is that several module objects can be created from a single shared library. For example:

```
>>> import sys
>>> import binascii
>>> old_binascii = binascii
>>> del sys.modules['binascii']
>>> import binascii  # create a new module object
>>> old_binascii == binascii
False
```

As a rule of thumb, the two modules should be completely independent. All objects and state specific to the module should be encapsulated within the module object, not shared with other module objects, and cleaned up when the module object is deallocated. Since this just is a rule of thumb, exceptions are possible (see *Managing Global State*), but they will need more thought and attention to edge cases.

While some modules could do with less stringent restrictions, isolated modules make it easier to set clear expectations and guidelines that work across a variety of use cases.

### 2.3 Surprising Edge Cases

Note that isolated modules do create some surprising edge cases. Most notably, each module object will typically not share its classes and exceptions with other similar modules. Continuing from the *example above*, note that `old_binascii.Error` and `binascii.Error` are separate objects. In the following code, the exception is *not* caught:

```
>>> old_binascii.Error == binascii.Error
False
>>> try:
...     old_binascii.unhexlify(b'qwertyuiop')
... except binascii.Error:
...     print('boo')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
binascii.Error: Non-hexadecimal digit found
```

This is expected. Notice that pure-Python modules behave the same way: it is a part of how Python works.

The goal is to make extension modules safe at the C level, not to make hacks behave intuitively. Mutating `sys.modules` "manually" counts as a hack.

## 3 Making Modules Safe with Multiple Interpreters

### 3.1 Managing Global State

Sometimes, the state associated with a Python module is not specific to that module, but to the entire process (or something else "more global" than a module). For example:

- The `readline` module manages *the* terminal.
- A module running on a circuit board wants to control *the* on-board LED.

In these cases, the Python module should provide *access* to the global state, rather than *own* it. If possible, write the module so that multiple copies of it can access the state independently (along with other libraries, whether for Python or other languages). If that is not possible, consider explicit locking.

If it is necessary to use process-global state, the simplest way to avoid issues with multiple interpreters is to explicitly prevent a module from being loaded more than once per process—see *Opt-Out: Limiting to One Module Object per Process*.

### 3.2 Managing Per-Module State

To use per-module state, use multi-phase extension module initialization. This signals that your module supports multiple interpreters correctly.

Set `PyModuleDef.m_size` to a positive number to request that many bytes of storage local to the module. Usually, this will be set to the size of some module-specific `struct`, which can store all of the module's C-level state. In particular, it is where you should put pointers to classes (including exceptions, but excluding static types) and settings (e.g. `csv`'s `field_size_limit`) which the C code needs to function.

> **Note**
>
> Another option is to store state in the module's `__dict__`, but you must avoid crashing when users modify `__dict__` from Python code. This usually means error- and type-checking at the C level, which is easy to get wrong and hard to test sufficiently.

However, if module state is not needed in C code, storing it in `__dict__` only is a good idea.

If the module state includes `PyObject` pointers, the module object must hold references to those objects and implement the module-level hooks `m_traverse`, `m_clear` and `m_free`. These work like `tp_traverse`, `tp_clear` and `tp_free` of a class. Adding them will require some work and make the code longer; this is the price for modules which can be unloaded cleanly.

An example of a module with per-module state is currently available as xxlimited; example module initialization shown at the bottom of the file.

### 3.3 Opt-Out: Limiting to One Module Object per Process

A non-negative `PyModuleDef.m_size` signals that a module supports multiple interpreters correctly. If this is not yet the case for your module, you can explicitly make your module loadable only once per process. For example:

```
static int loaded = 0;

static int
exec_module(PyObject* module)
{
    if (loaded) {
        PyErr_SetString(PyExc_ImportError,
                        "cannot load module more than once per process");
        return -1;
    }
    loaded = 1;
    // ... rest of initialization
}
```

### 3.4 Module State Access from Functions

Accessing the state from module-level functions is straightforward. Functions get the module object as their first argument; for extracting the state, you can use `PyModule_GetState`:

```
static PyObject *
func(PyObject *module, PyObject *args)
{
    my_struct *state = (my_struct*)PyModule_GetState(module);
    if (state == NULL) {
        return NULL;
    }
    // ... rest of logic
}
```

**Note**

`PyModule_GetState` may return `NULL` without setting an exception if there is no module state, i.e. `PyModuleDef.m_size` was zero. In your own module, you're in control of `m_size`, so this is easy to prevent.

## 4 Heap Types

Traditionally, types defined in C code are *static*; that is, `static PyTypeObject` structures defined directly in code and initialized using `PyType_Ready()`.

Such types are necessarily shared across the process. Sharing them between module objects requires paying attention to any state they own or access. To limit the possible issues, static types are immutable at the Python level: for example, you can't set `str.myattribute = 123`.

**CPython implementation detail:** Sharing truly immutable objects between interpreters is fine, as long as they don't provide access to mutable objects. However, in CPython, every Python object has a mutable implementation detail: the reference count. Changes to the refcount are guarded by the GIL. Thus, code that shares any Python objects across interpreters implicitly depends on CPython's current, process-wide GIL.

Because they are immutable and process-global, static types cannot access "their" module state. If any method of such a type requires access to module state, the type must be converted to a *heap-allocated type*, or *heap type* for short. These correspond more closely to classes created by Python's `class` statement.

For new modules, using heap types by default is a good rule of thumb.

## 4.1 Changing Static Types to Heap Types

Static types can be converted to heap types, but note that the heap type API was not designed for "lossless" conversion from static types—that is, creating a type that works exactly like a given static type. So, when rewriting the class definition in a new API, you are likely to unintentionally change a few details (e.g. pickleability or inherited slots). Always test the details that are important to you.

Watch out for the following two points in particular (but note that this is not a comprehensive list):

- Unlike static types, heap type objects are mutable by default. Use the `Py_TPFLAGS_IMMUTABLETYPE` flag to prevent mutability.
- Heap types inherit `tp_new` by default, so it may become possible to instantiate them from Python code. You can prevent this with the `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag.

## 4.2 Defining Heap Types

Heap types can be created by filling a `PyType_Spec` structure, a description or "blueprint" of a class, and calling `PyType_FromModuleAndSpec()` to construct a new class object.

> **Note**
>
> Other functions, like `PyType_FromSpec()`, can also create heap types, but `PyType_FromModuleAndSpec()` associates the module with the class, allowing access to the module state from methods.

The class should generally be stored in *both* the module state (for safe access from C) and the module's `__dict__` (for access from Python code).

## 4.3 Garbage-Collection Protocol

Instances of heap types hold a reference to their type. This ensures that the type isn't destroyed before all its instances are, but may result in reference cycles that need to be broken by the garbage collector.

To avoid memory leaks, instances of heap types must implement the garbage collection protocol. That is, heap types should:

- Have the `Py_TPFLAGS_HAVE_GC` flag.
- Define a traverse function using `Py_tp_traverse`, which visits the type (e.g. using `Py_VISIT(Py_TYPE(self))`).

Please refer to the documentation of `Py_TPFLAGS_HAVE_GC` and `tp_traverse` for additional considerations.

The API for defining heap types grew organically, leaving it somewhat awkward to use in its current state. The following sections will guide you through common issues.

### `tp_traverse` in Python 3.8 and lower

The requirement to visit the type from `tp_traverse` was added in Python 3.9. If you support Python 3.8 and lower, the traverse function must *not* visit the type, so it must be more complicated:

```
static int my_traverse(PyObject *self, visitproc visit, void *arg)
{
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
    return 0;
}
```

Unfortunately, `Py_Version` was only added in Python 3.11. As a replacement, use:

- `PY_VERSION_HEX`, if not using the stable ABI, or
- `sys.version_info` (via `PySys_GetObject()` and `PyArg_ParseTuple()`).

### Delegating `tp_traverse`

If your traverse function delegates to the `tp_traverse` of its base class (or another type), ensure that `Py_TYPE(self)` is visited only once. Note that only heap type are expected to visit the type in `tp_traverse`.

For example, if your traverse function includes:

```
base->tp_traverse(self, visit, arg)
```

…and `base` may be a static type, then it should also include:

```
if (base->tp_flags & Py_TPFLAGS_HEAPTYPE) {
    // a heap type's tp_traverse already visited Py_TYPE(self)
} else {
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
}
```

It is not necessary to handle the type's reference count in `tp_new` and `tp_clear`.

### Defining `tp_dealloc`

If your type has a custom `tp_dealloc` function, it needs to:

- call `PyObject_GC_UnTrack()` before any fields are invalidated, and
- decrement the reference count of the type.

To keep the type valid while `tp_free` is called, the type's refcount needs to be decremented *after* the instance is deallocated. For example:

```
static void my_dealloc(PyObject *self)
{
    PyObject_GC_UnTrack(self);
    ...
    PyTypeObject *type = Py_TYPE(self);
    type->tp_free(self);
    Py_DECREF(type);
}
```

The default `tp_dealloc` function does this, so if your type does *not* override `tp_dealloc` you don't need to add it.

#### Not overriding `tp_free`

The `tp_free` slot of a heap type must be set to `PyObject_GC_Del()`. This is the default; do not override it.

#### Avoiding `PyObject_New`

GC-tracked objects need to be allocated using GC-aware functions.

If you use use `PyObject_New()` or `PyObject_NewVar()`:

- Get and call type's `tp_alloc` slot, if possible. That is, replace `TYPE *o = PyObject_New(TYPE, typeobj)` with:

  ```
  TYPE *o = typeobj->tp_alloc(typeobj, 0);
  ```

  Replace `o = PyObject_NewVar(TYPE, typeobj, size)` with the same, but use size instead of the 0.

- If the above is not possible (e.g. inside a custom `tp_alloc`), call `PyObject_GC_New()` or `PyObject_GC_NewVar()`:

  ```
  TYPE *o = PyObject_GC_New(TYPE, typeobj);

  TYPE *o = PyObject_GC_NewVar(TYPE, typeobj, size);
  ```

## 4.4 Module State Access from Classes

If you have a type object defined with `PyType_FromModuleAndSpec()`, you can call `PyType_GetModule()` to get the associated module, and then `PyModule_GetState()` to get the module's state.

To save a some tedious error-handling boilerplate code, you can combine these two steps with `PyType_GetModuleState()`, resulting in:

```
my_struct *state = (my_struct*)PyType_GetModuleState(type);
if (state == NULL) {
    return NULL;
}
```

## 4.5 Module State Access from Regular Methods

Accessing the module-level state from methods of a class is somewhat more complicated, but is possible thanks to API introduced in Python 3.9. To get the state, you need to first get the *defining class*, and then get the module state from it.

The largest roadblock is getting *the class a method was defined in*, or that method's "defining class" for short. The defining class can have a reference to the module it is part of.

Do not confuse the defining class with `Py_TYPE(self)`. If the method is called on a *subclass* of your type, `Py_TYPE(self)` will refer to that subclass, which may be defined in different module than yours.

**Note**

The following Python code can illustrate the concept. `Base.get_defining_class` returns `Base` even if `type(self) == Sub`:

```
class Base:
    def get_type_of_self(self):
        return type(self)

    def get_defining_class(self):
        return __class__

class Sub(Base):
    pass
```

For a method to get its "defining class", it must use the METH_METHOD | METH_FASTCALL | METH_KEYWORDS `calling convention` and the corresponding `PyCMethod` signature:

```
PyObject *PyCMethod(
    PyObject *self,               // object the method was called on
    PyTypeObject *defining_class, // defining class
    PyObject *const *args,        // C array of arguments
    Py_ssize_t nargs,             // length of "args"
    PyObject *kwnames)            // NULL, or dict of keyword arguments
```

Once you have the defining class, call `PyType_GetModuleState()` to get the state of its associated module.

For example:

```
static PyObject *
example_method(PyObject *self,
        PyTypeObject *defining_class,
        PyObject *const *args,
        Py_ssize_t nargs,
        PyObject *kwnames)
{
    my_struct *state = (my_struct*)PyType_GetModuleState(defining_class);
    if (state == NULL) {
        return NULL;
    }
    ... // rest of logic
}

PyDoc_STRVAR(example_method_doc, "...");

static PyMethodDef my_methods[] = {
    {"example_method",
      (PyCFunction)(void(*)(void))example_method,
      METH_METHOD|METH_FASTCALL|METH_KEYWORDS,
      example_method_doc}
    {NULL},
}
```

## 4.6 Module State Access from Slot Methods, Getters and Setters

**Note**

This is new in Python 3.11.

Slot methods—the fast C equivalents for special methods, such as `nb_add` for `__add__` or `tp_new` for initialization—have a very simple API that doesn't allow passing in the defining class, unlike with `PyCMethod`. The same goes for getters and setters defined with `PyGetSetDef`.

To access the module state in these cases, use the `PyType_GetModuleByDef()` function, and pass in the module definition. Once you have the module, call `PyModule_GetState()` to get the state:

```
PyObject *module = PyType_GetModuleByDef(Py_TYPE(self), &module_def);
my_struct *state = (my_struct*)PyModule_GetState(module);
if (state == NULL) {
```

(continues on next page)

(continued from previous page)

```
    return NULL;
}
```

`PyType_GetModuleByDef()` works by searching the method resolution order (i.e. all superclasses) for the first superclass that has a corresponding module.

> **Note**
>
> In very exotic cases (inheritance chains spanning multiple modules created from the same definition), `PyType_GetModuleByDef()` might not return the module of the true defining class. However, it will always return a module with the same definition, ensuring a compatible C memory layout.

### 4.7 Lifetime of the Module State

When a module object is garbage-collected, its module state is freed. For each pointer to (a part of) the module state, you must hold a reference to the module object.

Usually this is not an issue, because types created with `PyType_FromModuleAndSpec()`, and their instances, hold a reference to the module. However, you must be careful in reference counting when you reference module state from other places, such as callbacks for external libraries.

## 5 Open Issues

Several issues around per-module state and heap types are still open.

Discussions about improving the situation are best held on the capi-sig mailing list.

### 5.1 Per-Class Scope

It is currently (as of Python 3.11) not possible to attach state to individual *types* without relying on CPython implementation details (which may change in the future—perhaps, ironically, to allow a proper solution for per-class scope).

### 5.2 Lossless Conversion to Heap Types

The heap type API was not designed for "lossless" conversion from static types; that is, creating a type that works exactly like a given static type.