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 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.

New in version 3.3.

Changed in 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 *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.

New in version 3.3.

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 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.

New in version 3.2.

char* **`PyModule_GetFilename`**(PyObject **module*)

Similar to `PyModule_GetFilenameObject()` but return the filename encoded to 'utf-8'.

Deprecated since 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 C and C++ Extensions or Extending Embedded Python 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`.

char* **m_name**

Name for the new module.

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*.

Changed in 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*)

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*)

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*)

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.

New in 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.

New in 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*)

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`.

New in version 3.5.

PyObject * **`PyModule_FromDefAndSpec2`**(PyModuleDef **def*, PyObject **spec*, int *module_api_version*)

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.

New in version 3.5.

int **`PyModule_ExecDef`**(PyObject **module*, PyModuleDef **def*)

Process any execution slots (`Py_mod_exec`) given in *def*.

New in 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`.

New in 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`.

New in 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*)

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.

New in version 3.3.

int **PyState_RemoveModule**(PyModuleDef **def*)

Removes the module object created from *def* from the interpreter state.

New in version 3.3.