Type Objects

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 (Pyth
                                    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;
    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.

New in 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.

New in 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()`:

Constant	Comparison
`Py_LT`	`<`
`Py_LE`	`<=`
`Py_EQ`	`==`

Constant	Comparison
Py_NE	!=
Py_GT	>
Py_GE	>=

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).

> **Warning:** 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 *t
```

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_dictoffs
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 *
```

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.

New in version 3.4.

See also: "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.

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.

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.

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 inplace 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 inplace multiplication via the `nb_inplace_multiply` slot.

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.

Async Object Structures

New in 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*.