Table of Contents

Previous topic

Initialization, Finalization, and Threads

Next topic

Object Implementation Support

This Page

Report a Bug
Show Source

Memory Management

Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: malloc(), calloc(), realloc() and free(). This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the string object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

See also: The PYTHONMALLOC environment variable can be used to configure the memory allocators used by Python.

The PYTHONMALLOCSTATS environment variable can be used to print statistics of the pymalloc memory allocator every time a new pymalloc object arena is created, and on shutdown.

Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the GIL does not need to be held.

The default raw memory allocator uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

New in version 3.4.

void* **PyMem_RawMalloc**(size_t *n*)

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails.

Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_RawCalloc**(size_t *nelem*, size_t *elsize*)

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

New in version 3.5.

void* **PyMem_RawRealloc**(void **p*, size_t *n*)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is *NULL*, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`.

If the request fails, `PyMem_RawRealloc()` returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyMem_RawFree**(void **p*)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is *NULL*, no operation is performed.

Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The default memory allocator uses the pymalloc memory allocator.

Warning: The GIL must be held when using these functions.

Changed in version 3.6: The default allocator is now pymalloc instead of system `malloc()`.

void* **PyMem_Malloc**(size_t *n*)

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails.

Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_Calloc**(size_t *nelem*, size_t *elsize*)

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type void* to the allocated memory, or *NULL* if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

New in version 3.5.

void* **PyMem_Realloc**(void **p*, size_t *n*)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is *NULL*, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to PyMem_Malloc(), PyMem_Realloc() or PyMem_Calloc().

If the request fails, PyMem_Realloc() returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void **p*)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to PyMem_Malloc(), PyMem_Realloc() or PyMem_Calloc(). Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is *NULL*, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

TYPE* **PyMem_New**(TYPE, size_t *n*)

Same as PyMem_Malloc(), but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to TYPE*. The memory will not have been initialized in any way.

TYPE* **PyMem_Resize**(void **p*, TYPE, size_t *n*)

Same as PyMem_Realloc(), but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to TYPE*. On return, *p* will be a pointer to the new memory area, or *NULL* in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void **p*)

Same as PyMem_Free().

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The default object allocator uses the pymalloc memory allocator.

Warning: The GIL must be held when using these functions.

void* **PyObject_Malloc**(size_t *n*)

Allocates *n* bytes and returns a pointer of type void* to the allocated memory, or *NULL* if the request fails.

Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyObject_Calloc**(size_t *nelem*, size_t *elsize*)

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type void* to the allocated memory, or *NULL* if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

New in version 3.5.

void* **PyObject_Realloc**(void **p*, size_t *n*)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is *NULL*, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`.

If the request fails, `PyObject_Realloc()` returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free**(void **p*)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is *NULL*, no operation is performed.

Default Memory Allocators

Default memory allocators:

Configuration	Name	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	"pymalloc"	malloc	pymalloc	pymalloc
Debug build	"pymalloc_debug"	malloc + debug	pymalloc + debug	pymalloc + debug
Release build, without pymalloc	"malloc"	malloc	malloc	malloc
Debug build, without pymalloc	"malloc_debug"	malloc + debug	malloc + debug	malloc + debug

Legend:

- Name: value for PYTHONMALLOC environment variable
- malloc: system allocators from the standard C library, C functions: malloc(), calloc(), realloc() and free()
- pymalloc: pymalloc memory allocator
- "+ debug": with debug hooks installed by PyMem_SetupDebugHooks()

Customize Memory Allocators

New in version 3.4.

PyMemAllocatorEx

Structure used to describe a memory block allocator. The structure has four fields:

Field	Meaning
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

Changed in version 3.5: The `PyMemAllocator` structure was renamed to PyMemAllocatorEx and a new `calloc` field was added.

PyMemAllocatorDomain

Enum used to identify an allocator domain. Domains:

PYMEM_DOMAIN_RAW

Functions:

- PyMem_RawMalloc()
- PyMem_RawRealloc()
- PyMem_RawCalloc()
- PyMem_RawFree()

PYMEM_DOMAIN_MEM

Functions:

- PyMem_Malloc(),
- PyMem_Realloc()
- PyMem_Calloc()
- PyMem_Free()

PYMEM_DOMAIN_OBJ

Functions:

- PyObject_Malloc()
- PyObject_Realloc()
- PyObject_Calloc()
- PyObject_Free()

void **PyMem_GetAllocator**(PyMemAllocatorDomain *domain*, PyMemAllocatorEx **allocator*)

Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(PyMemAllocatorDomain *domain*, PyMemAllocatorEx **allocator*)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-NULL pointer when requesting zero bytes.

For the PYMEM_DOMAIN_RAW domain, the allocator must be thread-safe: the GIL is not held when the allocator is called.

If the new allocator is not a hook (does not call the previous allocator), the PyMem_SetupDebugHooks() function must be called to reinstall the debug hooks on top on the new allocator.

void **PyMem_SetupDebugHooks**(void)

Setup hooks to detect bugs in the Python memory allocator functions.

Newly allocated memory is filled with the byte 0xCD (CLEANBYTE), freed memory is filled with the byte 0xDD (DEADBYTE). Memory blocks are surrounded by “forbidden bytes” (FORBIDDENBYTE: byte 0xFD).

Runtime checks:

- Detect API violations, ex: PyObject_Free() called on a buffer allocated by PyMem_Malloc()
- Detect write before the start of the buffer (buffer underflow)
- Detect write after the end of the buffer (buffer overflow)
- Check that the GIL is held when allocator functions of PYMEM_DOMAIN_OBJ (ex: PyObject_Malloc()) and PYMEM_DOMAIN_MEM (ex: PyMem_Malloc()) domains are called

On error, the debug hooks use the tracemalloc module to get the traceback where a memory block was allocated. The traceback is only displayed if tracemalloc is tracing Python memory allocations and the memory block was traced.

These hooks are installed by default if Python is compiled in debug mode. The PYTHONMALLOC environment variable can be used to install debug hooks on a Python compiled in release mode.

Changed in version 3.6: This function now also works on Python compiled in release mode. On error, the debug hooks now use tracemalloc to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of PYMEM_DOMAIN_OBJ and PYMEM_DOMAIN_MEM domains are called.

Changed in version 3.7.3: Byte patterns 0xCB (CLEANBYTE), 0xDB (DEADBYTE) and 0xFB (FORBIDDENBYTE) have been replaced with 0xCD, 0xDD and 0xFD to use the same values than Windows CRT debug malloc() and free().

The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called “arenas” with a fixed size of 256 KiB. It falls back to PyMem_RawMalloc() and PyMem_RawRealloc() for allocations larger than 512 bytes.

pymalloc is the default allocator of the PYMEM_DOMAIN_MEM (ex: PyMem_Malloc()) and PYMEM_DOMAIN_OBJ (ex: PyObject_Malloc()) domains.

The arena allocator uses the following functions:

- VirtualAlloc() and VirtualFree() on Windows,
- mmap() and munmap() if available,
- malloc() and free() otherwise.

Customize pymalloc Arena Allocator

New in version 3.4.

PyObjectArenaAllocator

Structure used to describe an arena allocator. The structure has three fields:

Field	Meaning
void *ctx	user context passed as first argument
void* alloc(void *ctx, size_t size)	allocate an arena of size bytes
void free(void *ctx, size_t size, void *ptr)	free an arena

PyObject_GetArenaAllocator(PyObjectArenaAllocator **allocator*)

Get the arena allocator.

PyObject_SetArenaAllocator(PyObjectArenaAllocator **allocator*)

Set the arena allocator.

tracemalloc C API

New in version 3.7.

Examples

Here is the example from section Overview, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with PyObject_New(), PyObject_NewVar() and PyObject_Del().

These will be explained in the next chapter on defining and implementing new object types in C.