# Memory management in Python

Last updated on July 08, 2019, in Python

This article describes memory management in Python 3.6. If you are interested in GC details, you can read my article about Garbage collection in Python.

Everything in Python is an object. Some objects can hold other objects, such as lists, tuples, dicts, classes, etc. Because of dynamic Python's nature, such approach requires a lot of small memory allocations. To speed-up memory operations and reduce fragmentation Python uses a special manager on top of the general-purpose allocator, called PyMalloc.

We can depict the whole system as a set of hierarchical layers:

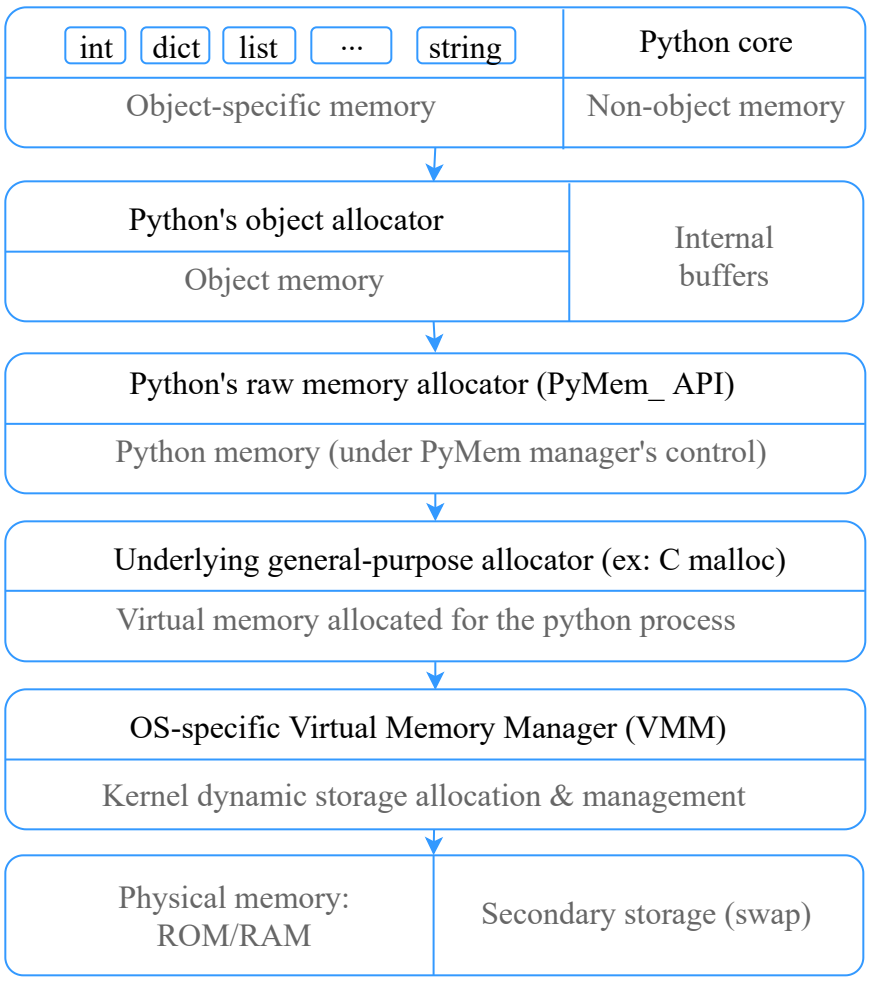



Illustration of hierarchical model (source).

## Small object allocation

To reduce overhead for small objects (less than 512 bytes) Python sub-allocates big blocks of memory. Larger objects are routed to standard C allocator. Small object allocator uses three levels of abstraction — `arena`, `pool`, and `block`.

Let's start with the smallest structure — block.

## Block

Block is a chunk of memory of a certain size. Each block can keep only one Python object of a fixed size. The size of the block can vary from 8 to 512 bytes and be a multiple of eight (i.e., 8-byte alignment). For convenience, such blocks are grouped in 64 size classes.

| Request in bytes | Size of allocated block | size class idx |
|---|---|---|
| 1-8 | 8 | 0 |
| 9-16 | 16 | 1 |
| 17-24 | 24 | 2 |
| 25-32 | 32 | 3 |
| 33-40 | 40 | 4 |
| 41-48 | 48 | 5 |
| ... | ... | ... |
| 505-512 | 512 | 63 |

## Pool

A collection of blocks of the same size is called a pool. Normally, the size of the pool is equal to the size of a memory page, i.e., 4Kb. Limiting pool to the fixed size of blocks helps with fragmentation. If an object gets destroyed, the memory manager can fill this space with a new object of the same size.

Each pool has a special header structure, which is defined as follows:

```
/* Pool for small blocks. */
struct pool_header {
    union { block *_padding;
            uint count; } ref;          /* number of allocated blocks    */
    block *freeblock;                   /* pool's free list head         */
```

```
    struct pool_header *nextpool;       /* next pool of this size class  */
    struct pool_header *prevpool;       /* previous pool       ""        */
    uint arenaindex;                    /* index into arenas of base adr */
    uint szidx;                         /* block size class index        */
    uint nextoffset;                    /* bytes to virgin block         */
    uint maxnextoffset;                 /* largest valid nextoffset      */
};
```

Pools of the same sized blocks are linked together using doubly linked list (the `nextpool` and `prevpool` fields). The `szidx` field keeps the size class index, whereas `ref.count` keeps the number of used blocks. The `arenaindex` stores the number of an arena in which Pool was created.

The `freeblock` field is described as follows:

```
Blocks within pools are again carved out as needed.  pool->freeblock points to
the start of a singly-linked list of free blocks within the pool.  When a
block is freed, it's inserted at the front of its pool's freeblock list.  Note
that the available blocks in a pool are *not* linked all together when a pool
is initialized.  Instead only "the first two" (lowest addresses) blocks are
set up, returning the first such block, and setting pool->freeblock to a
one-block list holding the second such block.  This is consistent with that
pymalloc strives at all levels (arena, pool, and block) never to touch a piece
of memory until it's actually needed.

So long as a pool is in the used state, we're certain there *is* a block
available for allocating, and pool->freeblock is not NULL.  If pool->freeblock
points to the end of the free list before we've carved the entire pool into
blocks, that means we simply haven't yet gotten to one of the higher-address
blocks.  The offset from the pool_header to the start of "the next" virgin
block is stored in the pool_header nextoffset member, and the largest value
of nextoffset that makes sense is stored in the maxnextoffset member when a
pool is initialized.  All the blocks in a pool have been passed out at least
once when and only when nextoffset > maxnextoffset.
```

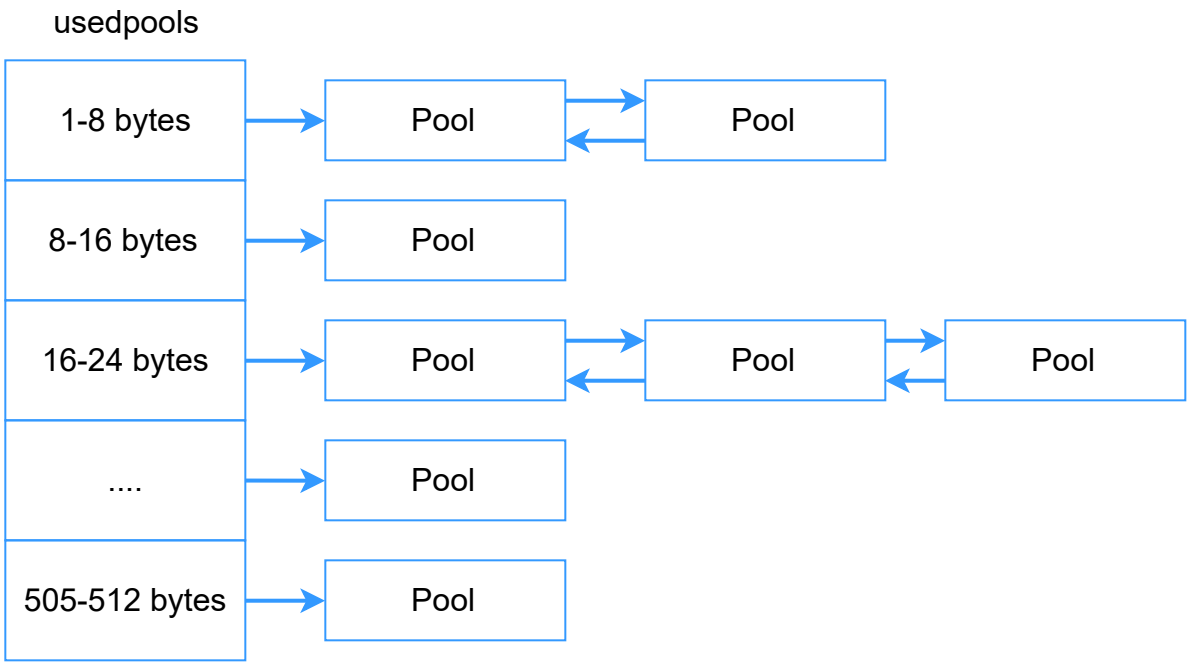
Therefore, If a block is empty instead of an object, it stores an address of the next empty block. This trick saves a lot of memory and computation.

Each pool has three states:

- used — partially used, neither empty nor full
- full — all the pool's blocks are currently allocated
- empty — all the pool's blocks are currently available for allocation

In order to efficiently manage pools Python uses an additional array called `usedpools`. It stores pointers to the pools grouped by class. As we already know, all pools of the same block size are

linked together. To iterate over them, we just need to know the start of the list. If there are no pools of such size, then a new pool will be created on the first memory request.




Note that pools and blocks are not allocating memory directly, instead they are using already allocated space from arenas.

## Arena

The arena is a chunk of 256kB memory allocated on the heap, which provides memory for 64 pools.

Arena

| | | |
|---|---|---|
| Pool (4kB) | Pool (4kB) | Pool (4kB) |
| Pool (4kB) | Pool (4kB) | Pool (4kB) |
| Pool (4kB) | Pool (4kB) | Pool (4kB) |
| Pool (4kB) | Pool (4kB) | Free Pool (4kB) |
| Free Pool (4kB) | Free Pool (4kB) | Free Pool (4kB) |
| ... | ... | ... |

The structure of arena object looks as follows:

```
struct arena_object {
    uintptr_t address;
    block* pool_address;
    uint nfreepools;
    uint ntotalpools;
    struct pool_header* freepools;
    struct arena_object* nextarena;
    struct arena_object* prevarena;
};
```

All arenas are linked using doubly linked list (the `nextarena` and `prevarena` fields), it helps to manage them. The `ntotalpools` and `nfreepools` are storing information about currently available pools.

The `freepools` field points to the linked list of available pools.

There is nothing complicated in the implementation of the arena. Think of it as a list of containers, which automatically allocates new memory for pools when needed.

## Memory deallocation

Python's small object manager rarely returns memory back to the Operating System.

An arena gets fully released If and only if all the pools in it are empty. For example, it can happen when you use a lot of temporary objects in a short period of time.

Speaking of long-running Python processes, they may hold a lot of unused memory because of this behavior.

### Allocation statistics

You can get allocations statistics by calling `sys._debugmallocstats()`.

## Diving deeper

If you are interested in more details about memory management, you can read the comments from the source code.

› Popular posts in Python category

Python, cpython internals, advanced python, memory

RSS

Want a monthly digest of these blog posts?

Email

Subscribe

No spam. No unnecessary emails.

Load disqus comments

Back to top