Object life cycle

What object is and more

Every object has:

- identity: is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same identity value. This is the address of the object in memory.
- type
- value

Objects are created at runtime, and can be modified further after creation.

Object types:

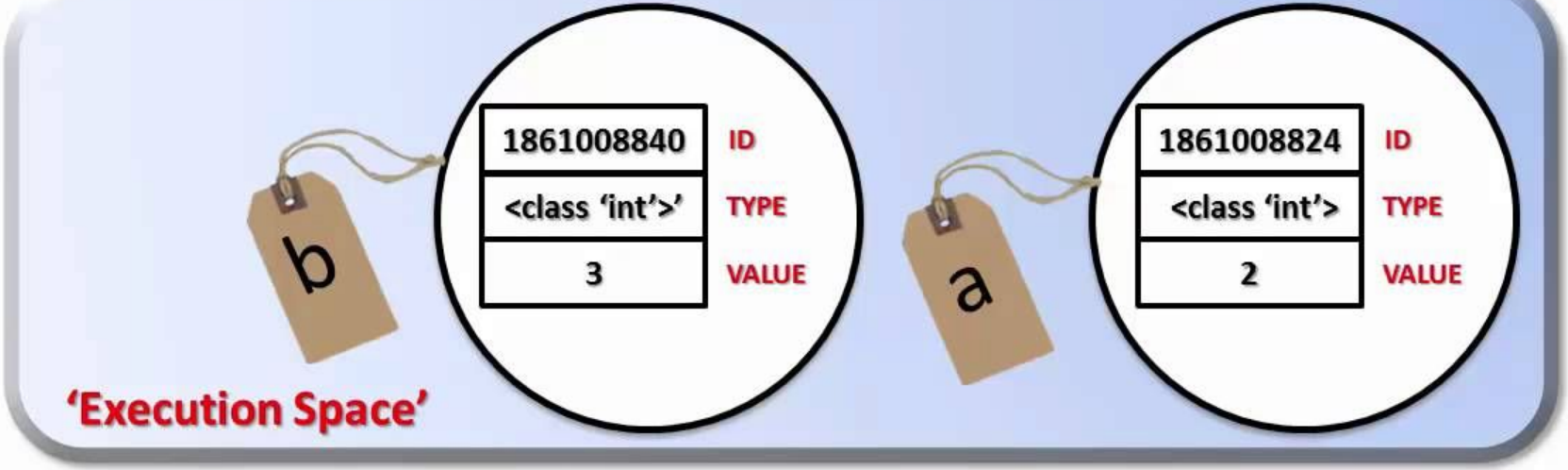
There are 2 types of objects:

- immutable: int, long, float, complex, str/unicode bytes, tuple, frozenset
- mutable: list, dict, set, byte-array

```
File  Edit  Format  Run  Options  Windo
a = 2
b = 3
print(id(a))
print(type(a))
print(a)
print(id(b))
print(type(b))
print(b)
```

```
>>> ====================
>>>
1861008824
<class 'int'>
2
1861008840
<class 'int'>
3
>>>
```

integer Class



Object life stages

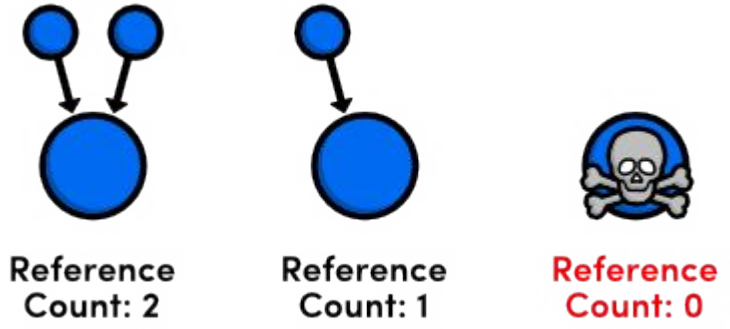
1. **Definition + Initialization**
2. **Access and Manipulation**
3. **Destruction**

Алгоритми GC

Стандартний інтерпритатор пайтона використовує одразу два алгоритма - підрахунок референсів і generational GC, яким можна маніпулювати стандартним модулем gc.

Алгоритм підрахунку референсів дуже простий і ефективний, але він не вміє визначати циклічні референси, тому і використовується додатковий gc, який слідкує за об’єктами і намагається вичислити циклічні референси.

Reference count



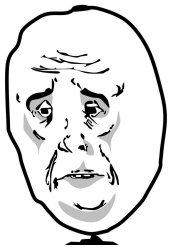
Коли збільшується кількість посилань на об’єкт

- оператор присвоювання
- передача аргументів
- об’єкт вставляється в ліст
- конструція вигляду foo = bar (фу референсить той же об’єкт що і бар)

Як тільки референс каунтер для об’єкта досягає нуля інтерпритатор запускає процес знищення об’єкта. Якщо об’єкт мав в собі референси на інші об’єкти, ці референси теж знищуються.

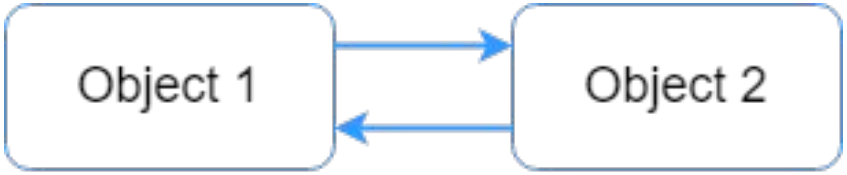
Змінні створені всередині **локального скоупу** знищуються як тільки пайтон виходить

з відповідного блоку кода



Змінні які створені в **глобальному скоупі** живуть як правило стільки ж як і сам python процес. Кількість референсів на об’єкти які живуть в глобальному скоупі ніколи не падає до нуля.

Циклічні залежності



Циклічні залежності можуть виникати тільки в контейнерних об’єктах. GC не слідкує за простими і незмінними типами (за викл. туплів).


```
a = {}
b = {}
a['b'] = b # a contains ref to b
b['a'] = a # b contains ref to a
del a
del b
```

Коли спрацьовує циклічний GC

На відміну від алгоритму підрахунку посилань, циклічний GC не працює в режимі реального часу і запускається періодично. Кожен запуск створює мікропаузи в роботі коду.

Циклічний збирач сміття розділяє всі об'єкти на 3 покоління (генерації).

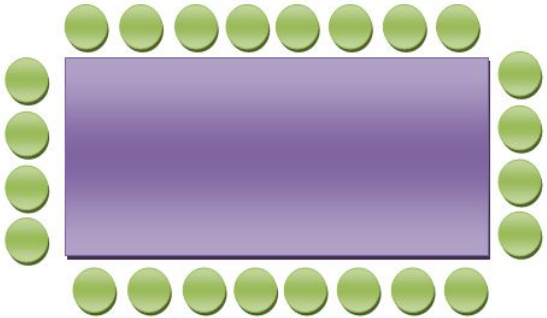
У кожному поколінні є спеціальний лічильник і поріг спрацьовування, при досягненні яких спрацьовує процес збірки сміття. Кожен лічильник зберігає кількість аллокацій мінус кількість деаллокацій в даній генерації. Як тільки в Python створюється будь-якої контейнерний об'єкт, він перевіряє ці лічильники. Якщо умови спрацьовують, то починається процес збірки сміття.

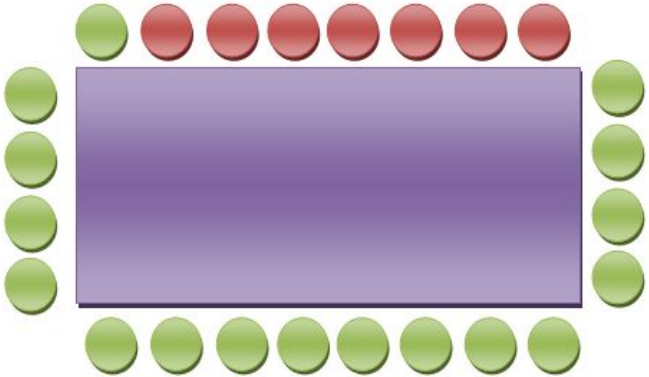
Якщо відразу кілька або більше поколінь подолали поріг, то вибирається найбільш старше покоління. Це зроблено через те, що старі покоління також сканують всі попередні. Щоб скоротити число пауз збірки сміття для довгоживучих об'єктів, найстарша генерація має додатковий набір умов.

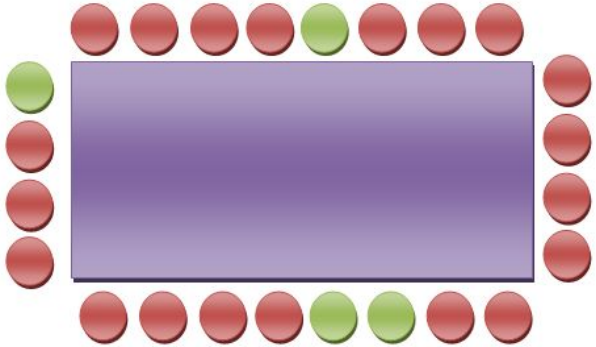
Стандартні пороги спрацьовування для поколінь встановлені на 700, 10 і 10 відповідно, але ви завжди можете їх змінити за допомогою функцій gc.get_threshold і gc.set_threshold.

GC ітерує кожен об'єкт з обраних поколінь і тимчасово видаляє всі посилання від окремо взятого об'єкта (всі посилання на які цей об'єкт посилається). Після повного проходу, всі об'єкти, у яких лічильник посилань менше двох вважаються недоступними з пітона і можуть бути видалені.

wat is allocation?







Python memory managment of small objects

To speed-up memory operations and reduce fragmentation Python uses a special manager on top of the general-purpose allocator, called PyMalloc.

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.

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).

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 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

Arena

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

Arena

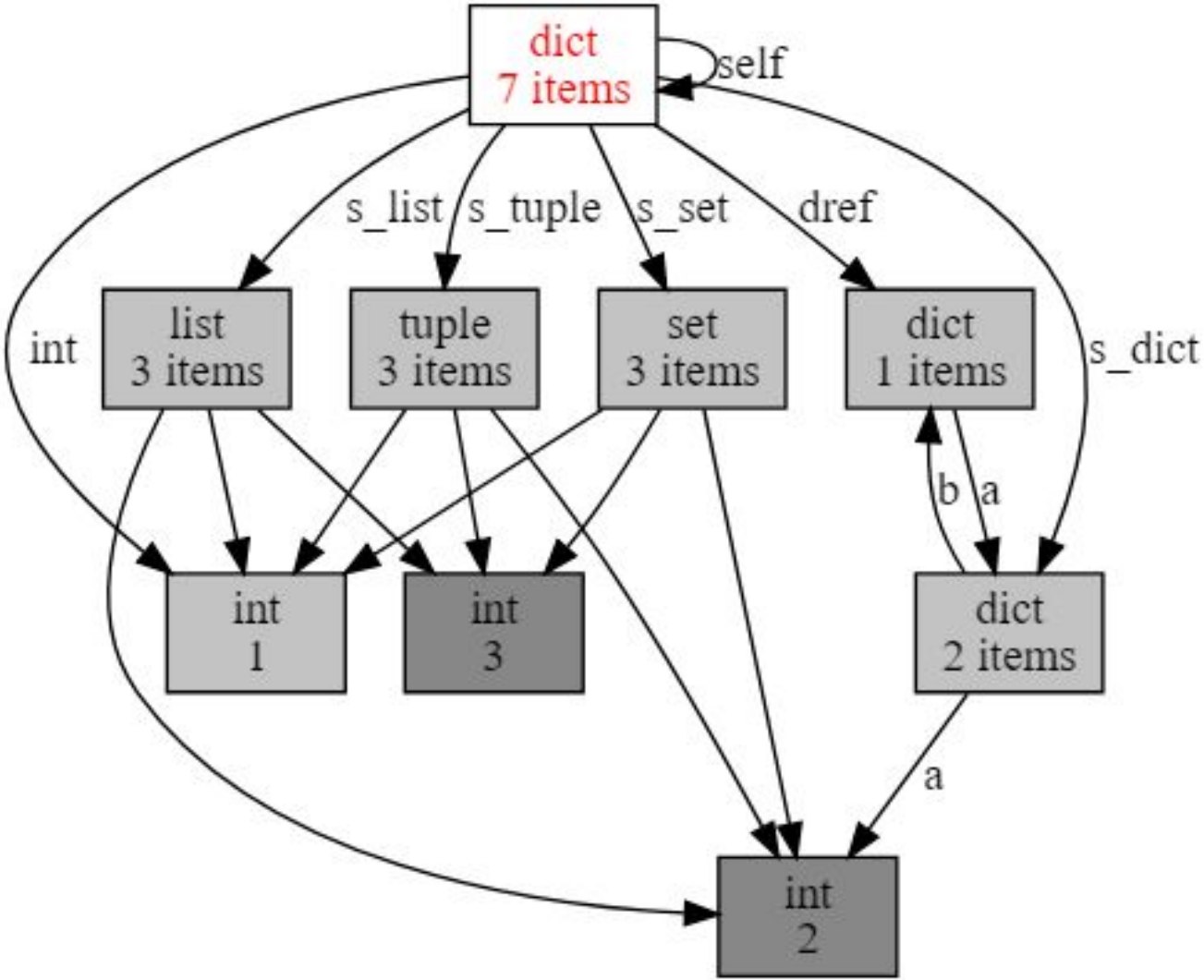
Pool (4kB)	Pool (4kB)	Pool (4kB)
Pool (4kB)	Pool (4kB)	Pool (4kB)
Pool (4kB)	Pool (4kB)	Pool (4kB)
Pool (4kB)	Pool (4kB)	Free Pool (4kB)
Free Pool (4kB)	Free Pool (4kB)	Free Pool (4kB)
...	...	...

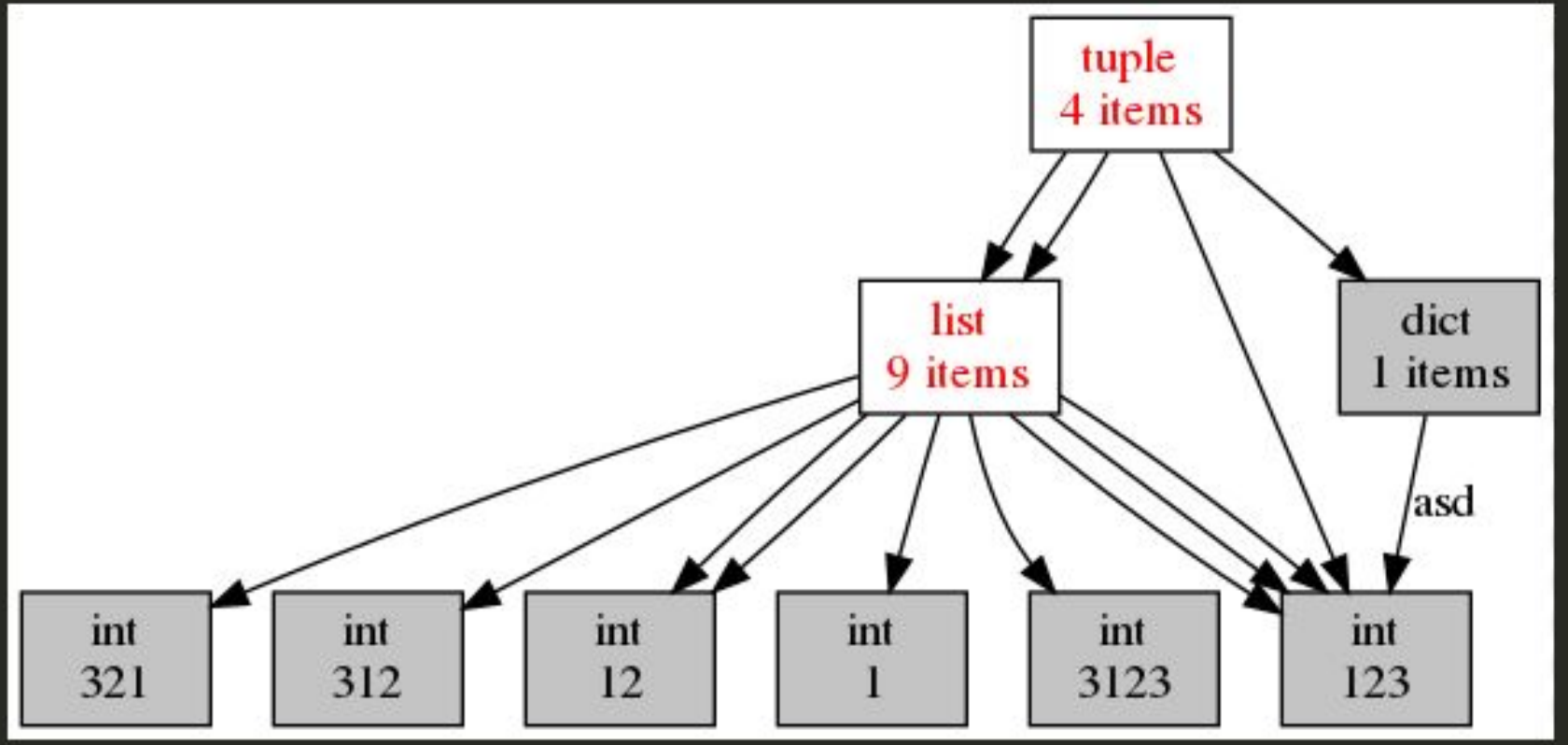
Python's small object manager rarely returns memory back to the Operating System.

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.

objgraph


dict
7 items
self
s_list
s_tuple
s_set
dref
int
list
3 items
tuple
3 items
set
3 items
dict
1 items
s_dict
b
a
int
1
int
3
dict
2 items
a
int
2


tuple
4 items
list
9 items
dict
1 items
asd
int
321
int
312
int
12
int
1
int
3123
int
123