

Python: учёт ссылок, сборка мусора и профилирование памяти

Содержание

- [1. Различие между `is` и `==` в Python](#)
- [2. Модуль `psutil`: CPU и память](#)
- [3. Ограничения ресурсов \(`resource`\)](#)
- [4. Модуль `tracemalloc` для профилирования памяти](#)
- [5. Подсчёт ссылок \(Reference Counting\)](#)
- [6. Сборщик мусора \(GC\)](#)
- [7. Слабые ссылки \(`weakref`\)](#)
- [8. Многопоточность и GC/RC](#)
- [9. Пороговые значения GC \(`gc.set_threshold`\)](#)
- [10. Утечки памяти: обнаружение неосвобождённых объектов](#)
- [11. Free-threading, Biased RC, Deferred RC](#)
- [12. Структура PyObject в CPython](#)
- [13. Новый сборщик мусора в Python 3.12–3.13](#)

1. Различие между `is` и `==` в Python

В Python оператор `is` проверяет **тождество** объектов, а оператор `==` проверяет **равенство** по значению. То есть `a is b` истинно только тогда, когда `a` и `b` — это один и тот же объект в памяти (один и тот же адрес) ¹. Функция `id(obj)` возвращает уникальный идентификатор объекта (в CPython это адрес в памяти) ¹. Оператор `==` по умолчанию вызывает метод `__eq__`, определяющий логику равенства значений. У встроенных типов (например, числа, строки) `__eq__` реализован для сравнения значений, а у пользовательских классов по умолчанию (`object.__eq__`) он просто проверяет тождество (как `is`) ².

Например, для небольших целых чисел (обычно $-5...256$ в CPython) Python использует кэширование (интернирование) целых. Поэтому для маленьких `int` две переменные могут ссылаться на один и тот же объект:

```
a = 256
b = 256
print(a == b, a is b) # True, True (значения равны, и объекты совпадают)
c = 257
d = 257
print(c == d, c is d) # True, False (значения равны, объекты разные)
```

Это обусловлено кэшем малых чисел в CPython ³. Аналогично, короткие строковые литералы могут быть интернированы (в том числе идентификаторы), что может сделать `a is b`

ИСТИННЫМ:

```
x = "hello"
y = "hello"
print(x == y, x is y) # True, True (строки равны, возможно интернированы)
s1 = "hello world"
s2 = "hello world"
print(s1 == s2, s1 is s2) # True, False (значения равны, объекты разные)
```

То есть `is` означает «это один и тот же объект», а `==` означает «значения равны (по `__eq__`)» ¹ ².

- `None` — это одиночный объект (singleton). Правильная проверка на `None` – через `is`:

```
if x is None:
    ...
```

- **Списки, множества и словари** сравниваются поэлементно. Два разных списка с одинаковым содержимым будут равны по `==`, но не тождественны:

```
L1 = [1, 2, 3]
L2 = [1, 2, 3]
print(L1 == L2, L1 is L2) # True, False
```

- **Переопределённый `__eq__`**: пользовательские классы могут определить метод `__eq__`. Тогда `==` будет сравнивать объекты по собственной логике, а `is` всё равно проверяет идентичность. Например:

```
class Vector:
    def __init__(self, x,y):
        self.x,self.y = x,y
    def __eq__(self, other):
        return isinstance(other, Vector) and (self.x, self.y)==(other.x,
other.y)

v1 = Vector(1,2)
v2 = Vector(1,2)
print(v1 == v2, v1 is v2) # True, False
```

По умолчанию (если `__eq__` не задан), сравнение пользовательских объектов просто проверяет `is` ².

Итого: `a is b` проверяет идентичность объекта, `a == b` – равенство по значению (через `__eq__`). Это особенно важно учитывать при сравнении неизменяемых объектов: они могут быть равны, но различны по адресу (как большие числа, строки с пробелами и т.д.).

2. Модуль `psutil`: CPU и память

Модуль `psutil` позволяет собирать информацию о загрузке системы и процессах.

- `psutil.cpu_percent(interval=1, percpu=False)` возвращает процент загрузки CPU за указанный интервал (в секундах). Например, `psutil.cpu_percent(interval=1)` блокирует выполнение на 1 секунду и измеряет нагрузку, возвращая число в процентах. Если `interval=None`, метод вернёт процент с момента последнего вызова (неблокирующий режим). Если `percpu=True`, возвращается список загрузок по каждому CPU ⁴ ⁵.
- `psutil.virtual_memory()` возвращает статистику использования физической памяти в виде именованного кортежа с полями (в байтах):
 - `total` – общий объём физической памяти (RAM) ⁶.
 - `available` – объём памяти, который может быть выделен мгновенно без использования swap (различается по платформе) ⁶.
 - `percent` – процент использования ($(total - available) / total * 100$) ⁶.
 - `used` – занято памяти (разница `total - available`, но считается по-разному на разных ОС) ⁷.
 - `free` – объём абсолютно свободной (заново обнулённой) памяти; не всегда соответствует реальному доступному объёму (см. `available`) ⁷.
 - `active`, `inactive` (UNIX) – объём недавно используемой и неактивной памяти соответственно ⁸.
 - `buffers`, `cached` (Linux/BSD) – кэш файловых метаданных и кэш (например, страниц файлов) ⁹.
 - `shared` – память, доступная совместно нескольким процессам ¹⁰.
 - `slab` – память, выделенная под внутренние структуры ядра (Linux) ¹⁰.(Например, на Linux вывод может выглядеть так:

```
mem = psutil.virtual_memory()
print(mem)
# svmem(total=16305916928, available=6259399680, percent=61.6,
#       used=10046558768,
#       free=2847756544, active=6592863744, inactive=2074094080,
#       buffers=1650113536, cached=3333606912, shared=264068864,
#       slab=272010112)
```

)

Эти метрики помогают понять использование RAM. Поля `available` и `percent` обычно даются в приоритете для определения свободной памяти на разных ОС ¹¹.

- `psutil.Process().memory_full_info()` возвращает детальную информацию об использовании памяти конкретным процессом. Полученный объект содержит поля (на Linux, macOS, Windows):
 - `rss` (Resident Set Size) – физический размер памяти, который процесс занимает прямо сейчас.
 - `vms` – виртуальный адресный размер процесса (включая всё адресное пространство).
 - `shared` – память, разделяемая с другими процессами.
 - `text`, `data`, `lib`, `dirty` – специализированные поля (на Linux) для текстового сегмента, сегмента данных, библиотек и «грязных» страниц ¹².

Модуль `memory_full_info()` добавляет к `memory_info()` ещё:

- `uss` (**Unique Set Size**) – объём памяти, уникальный для процесса, т.е. сколько реально «освободится», если процесс завершится ¹³.
- `pss` (**Proportional Set Size**) – пропорциональный объём общей памяти, который равномерно делится между процессами, её разделяющими ¹³.
- `swap` – объём памяти, выгруженной на диск (swap) ¹⁴.

Например:

```
import psutil
proc = psutil.Process()
info = proc.memory_full_info()
print(info)
# pfullmem(rss=15491072, vms=84025344, shared=5206016, text=2555904,
#          lib=0, data=33261568, dirty=0, uss=12456704, pss=13061440,
#          swap=0)
```

Среди полей наиболее информативны `rss` и `uss`. `rss` показывает фактическую физическую память, используемую процессом, а `uss` – сколько памяти реально занято именно этим процессом (без учёта разделяемых страниц) ¹⁵ ¹⁶. Эти метрики помогают анализировать, насколько процесс нагружает память и какой объём освободится после его завершения.

3. Ограничения ресурсов (`resource`)

Модуль `resource` позволяет устанавливать и получать лимиты на различные ресурсы процесса (например, потребление памяти, число процессов и т.д.). Основные функции – `resource.getrlimit(resource)`, `resource.setrlimit(resource, (soft, hard))`. Лимиты задаются в виде пары (текущий_уровень, максимальный).

Например, типовые ресурсы:

- `RLIMIT_STACK` – максимальный размер стека (в байтах) процесса ¹⁷ (во многопоточных программах влияет только на главный поток).
- `RLIMIT_DATA` – максимальный размер сегмента данных (heap) в байтах ¹⁸.
- `RLIMIT_RSS` – «рекомендуемый» максимальный размер резидентного набора (RAM) ¹⁹.
- `RLIMIT_AS` – максимальный общий адресный объём процесса (виртуальная память) ²⁰.
- `RLIMIT_NPROC` – максимальное число процессов/потоков, которые может создать процесс (для текущего пользователя) ²¹.
- `RLIMIT_MEMLOCK` – максимальный объём памяти, который процесс может «заблокировать» в RAM (предотвратить выгрузку в swap) ²².
- `RLIMIT_CPU` – максимальное процессорное время (в секундах). При превышении посылается сигнал `SIGXCPU` ²³.
- И другие (например, `RLIMIT_NOFILE` – число открытых файлов).

```
import resource

# Проверка текущих лимитов:
soft, hard = resource.getrlimit(resource.RLIMIT_STACK)
print("Stack limit:", soft, "bytes")
```

```
# Попытка установить небольшой лимит CPU (1 секунда).
resource.setrlimit(resource.RLIMIT_CPU, (1, 1))
```

При превышении лимита возникают ошибки или сигналы:

- Если `setrlimit` вызван некорректно (неверный ресурс, `soft > hard` и т.п.), Python выбросит `ValueError` ²⁴. Если процесс пытается увеличить «жёсткий» лимит выше системного, тоже `ValueError` ²⁴. Нормальным пользователям недоступно повышение «жёстких» лимитов сверх системных.

- При превышении реальных лимитов ОС действует по виду ресурса. Например, для `RLIMIT_CPU` процесс получит `SIGXCPU` (по умолчанию приводит к завершению) ²³. Для ограничений памяти ОС может завершить процесс (например, Out-Of-Memory Kill). Для `RLIMIT_NPROC` создание новой нити/процесса завершится ошибкой (`OSError/Errno`).

- Типичный пример:

```
import subprocess, resource
# Установим лимит на 1 процесс (себя) – даже дочерний spawn будет запрещен.
resource.setrlimit(resource.RLIMIT_NPROC, (1, 1))
# Попытка создать новый процесс (например, os.fork() или subprocess) может
# завершиться ошибкой:
subprocess.Popen(["echo", "hello"]) # приведёт к ошибке из-за RLIMIT_NPROC
```

Таким образом, с помощью `resource.setrlimit` можно принудительно ограничивать потребление памяти, количество потоков, CPU-время и т.д. Это полезно для безопасного запуска неблагонадёжного кода или ограничения ресурсов задач в операционной системе.

4. Модуль `tracemalloc` для профилирования памяти

Модуль `tracemalloc` позволяет отслеживать и анализировать распределение памяти Python-процессом. Его основные возможности ²⁵:

- `tracemalloc.start(nframe=1)` – начинает трассировку распределений памяти. Опция `nframe` задаёт число кадров стека (по умолчанию 1), которые будут сохраняться в записи об аллокации (для группировок по стеку). Обычно начинают трассировку как можно раньше (например, перед импортом других модулей) ²⁵ ²⁶.
- `tracemalloc.get_traced_memory()` – возвращает кортеж `(current, peak)` с текущим объёмом памяти (в байтах), занимаемым отслеживаемыми блоками, и максимумом с момента старта трассировки ²⁷.
- `tracemalloc.take_snapshot()` – берёт «снимок» текущего состояния всех распределённых блоков памяти (создаёт `Snapshot`-объект) ²⁸.
- `tracemalloc.stop()` – останавливает трассировку и очищает все собранные данные. Обычно после этого `get_traced_memory` вернёт `(0,0)`.
- **Анализ снимков:**
- `snapshot.statistics(key_type, cumulative=False)` возвращает список статистик `Statistic`, сгруппированных по `key_type` (`'filename'`, `'lineno'` или `'traceback'`) ²⁹. Например, `snapshot.statistics('lineno')` покажет, в каких строках какого файла было выделено больше всего памяти, с указанием количества и суммарного размера блоков. Если `cumulative=True`, суммируются все кадры стека, а не только текущий ³⁰.

- `snapshot2.compare_to(snapshot1, 'lineno')` вычисляет разницу между двумя снимками, возвращая список `StatisticDiff`, упорядоченный по величине изменения расхода памяти ³¹. Это позволяет увидеть, какие объекты «утекают» (у которых увеличилось число блоков или объём) между двумя точками времени.

Пример использования для поиска утечки:

```
import tracemalloc

tracemalloc.start()
# ... выполняем код, в котором подозреваем утечку ...
snapshot1 = tracemalloc.take_snapshot()
# Допустим, запускаем подозрительный код в цикле
for _ in range(1000):
    lst = [i for i in range(1000)] # пример временных выделений
snapshot2 = tracemalloc.take_snapshot()
top_stats = snapshot2.compare_to(snapshot1, 'lineno')
for stat in top_stats[:5]:
    print(stat)
```

Выше мы сравниваем два снимка: до и после выполнения кода. Результат покажет по строкам кода, насколько выросло потребление памяти. Например, если какой-то список или структура не освобождается, это будет видно как положительный рост (`size_diff`) ³¹.

Таким образом, `tracemalloc` позволяет диагностировать утечки памяти: сравнивая снимки, видно, какие участки кода сгенерировали наибольший рост используемой памяти. Интерпретация результатов и поиск утечек часто строится на сравнении статистик `statistics()` и `compare_to()`, позволяющих локализовать проблемные участки (строки/файлы) в коде.

5. Подсчёт ссылок (Reference Counting)

CPython реализует сборку мусора с подсчётом ссылок. Каждый объект содержит число ссылок (`refcount`), увеличивающееся при создании новой ссылки на объект (`Py_INCREF`) и уменьшающееся при удалении ссылки (`Py_DECREF`). Когда счётчик ссылок объекта становится нулём, объект немедленно уничтожается (освобождается память и вызывается деструктор `__del__`, если он определён). Макросы C-API: `Py_INCREF(o)` – добавить ссылку, `Py_DECREF(o)` – удалить ³². Например:

```
PyObject *obj = ...;
Py_INCREF(obj); // указали, что ещё одно место ссылается на obj
Py_DECREF(obj); // убрали одну ссылку
```

Вызов `Py_DECREF` автоматически приведёт к `free` объекта, если после этого ссылка была последней (`refcount` стал 0).

`del` vs `__del__`:
Выражение `del x` в Python просто удаляет имя `x`, тем самым уменьшая счётчик ссылок на

объект. Оно не гарантирует немедленный вызов деструктора, если на объект ещё есть другие ссылки. Метод `__del__(self)` – это финализатор, который вызывается при разрушении объекта (когда `refcount` достигнет 0 и объект готовится к удалению). Однако важно понимать, что `del` не вызывает `__del__` явно – `__del__` будет вызван в процессе реального освобождения памяти. До Python 3.4 попадание объектов с `__del__` в мусор было неблагоприятным: такие объекты в циклах попадали в список `gc.garbage` (недоступные объекты, требующие ручной сборки). С появлением PEP 442 (в Python 3.4) объекты с `__del__` стали финализироваться безопасно и **не попадают в `gc.garbage`** по умолчанию ³³.

Алгоритм и многопоточность:

- В однопоточном CPython обновления `refcount` **гарантированно защищены GIL**. Благодаря GIL атомарность изменений счётчика ссылок обеспечена на уровне одновременной работы нитей – в стандартном режиме GIL позволяет обрабатывать операции `Py_INCREF` и `Py_DECREF` без гонок.
- Без GIL (в экспериментальном `free-threading`, см. ниже) простые инкремент/декремент референс-счетчика не защищены атомарно, поэтому вводятся другие механизмы (см. раздел о *Biased RC*).
- `del` просто уменьшает счётчик и при необходимости вызывает освобождение. `__del__` – метод, выполняющий дополнительную работу при уничтожении (закрытие файлов, логирование и т.д.). Объекты с `__del__` особенно чувствительны к циклам ссылок – до Python 3.4 такие циклы с `__del__` считались *неразрешимыми* и попадали в `gc.garbage`. С 3.4 финализатор вызывается в отдельной фазе (PEP 442), поэтому протокол освобождения стал более надёжным ³³.

6. Сборщик мусора (GC)

CPython сочетает подсчёт ссылок с **трассирующим сборщиком мусора (GC)** для обнаружения циклических ссылок. GC организован по поколенческой схеме:

- **Поколения:** объекты делятся на три поколения: 0 (молодое), 1 (среднее), 2 (старшее) ³⁴. Новые объекты попадают в поколение 0. Если объект переживает сборку в поколении 0, он продвигается в поколение 1; аналогично при выживании в 1 – переходит в 2. Объекты старшего поколения (2) остаются в нём до удаления ³⁴. Такая схема предполагает, что молодые объекты скорее уничтожаются рано, а старые – живут дольше.

Функции модуля `gc`:

- `gc.collect([generation])` – запускает сборку «от мусора» для указанного поколения (0, 1 или 2). Без аргумента выполняется полная сборка (то есть поколение 2) ³⁵. Возвращает число найденных и удалённых объектов, плюс число «недоступных» объектов.
- `gc.get_stats()` – возвращает статистику по поколениям в виде списка словарей: для каждого поколения доступны ключи `collections` (сколько раз проходили сборки), `collected` (число собранных объектов) и `uncollectable` (число объектов, не собранных из-за циклов) ³⁶. Это полезно для мониторинга частоты сборок и утечек.
- `gc.enable()` / `gc.disable()` – включают или выключают автоматическую сборку мусора. Если сборка отключена, циклы ссылок не будут автоматически очищаться (только ручной `gc.collect()`).

- `gc.get_objects()` – возвращает список всех отслеживаемых сборщиком объектов; если передать параметр `generation`, ограничивается указанным поколением ³⁷. Обычно используется для отладки.
- `gc.get_count()` – возвращает кортеж `(count0, count1, count2)` текущих счётчиков – число выделений минус число освобождений для каждого поколения с момента последней сборки ³⁸. Эти счётчики используются, чтобы определить, когда запускать следующий GC.
- `gc.get_debug()` – возвращает текущие флаги отладки GC (битовые константы `DEBUG_LEAK`, `DEBUG_SAVEALL` и др.).
- `gc.freeze()` и `gc.unfreeze()` – переводят все объекты в «замороженное» состояние (по сути, перемещают их в поколение вне списка объектов для сбора) ³⁹. Полезно при форке процесса: после `fork()` часто вызывают `gc.freeze()`, чтобы в дочернем процессе избежать двойного удаления некоторых объектов, а затем `gc.unfreeze()` после отпуска GIL.
- **Колбэки:** `gc.callbacks` – список функций, вызываемых до и после каждой сборки. Каждая функция получает аргументы `(phase, info)`, где `phase` – `'start'` или `'stop'`, а `info` – словарь с ключами `generation`, `collected`, `uncollectable`. Это позволяет логировать статистику сборок.
- `gc.garbage` – список объектов, которые сборщик нашёл недоступными, но не смог удалить (обычно из-за наличия `__del__` и флага `DEBUG_SAVEALL`). После Python 3.4 в норме он пуст, так как PEP 442 позволяет очищать такие объекты безопасно. Но при включении флага отладки (`gc.set_debug(gc.DEBUG_SAVEALL)`) все «нераскрываемые» объекты копируются сюда вместо освобождения ³³. Можно проверить его после `gc.collect()` для обнаружения «висячих» циклов (утечек).

Пример циклической утечки:

```
class Node:
    def __init__(self):
        self.ref = None
    def __del__(self):
        # финализатор
        print("Node deleted")

a = Node()
b = Node()
a.ref = b
b.ref = a
del a, b
import gc
gc.collect()
print("garbage:", gc.garbage)
```

В старых версиях (до 3.4) эти два объекта с циклом и `__del__` попали бы в `gc.garbage` (и не были освобождены). В современном CPython их `__del__` вызовется безопасно ³³, и они не появятся в `gc.garbage`.

7. Слабые ссылки (`weakref`)

Слабая ссылка (`weakref`) – это ссылка на объект, которая не увеличивает счётчик ссылок. Если на объект больше нет **прямых (сильных)** ссылок, сборщик мусора может его уничтожить, даже если на него есть слабые ссылки. Это удобно для кешей, подписчиков и предотвращения циклов:

- **Кэши:** часто требуется хранить большие объекты в словаре, но не задерживать их жёстко. `weakref.WeakValueDictionary` или `WeakKeyDictionary` используются для таких целей ⁴⁰. Например, если объект хранится только в слабом словаре и нигде более, то после освобождения всех сильных ссылок он будет удалён GC, а соответствующая запись в слабом словаре автоматически уберётся ⁴⁰.
- **Подписчики (обработчики):** можно хранить подписчиков через `weakref.ref`; при уничтожении подписчика сборщик мусора автоматически «уведомляет» слабую ссылку (её вызов вернёт `None`). Это позволяет строить паттерн «подписка/уведомление», не создавая утечек из-за циклов.
- **Деревья и циклы:** при реализации деревьев часто ссылка от дочернего узла к родительскому создаёт цикл. Использование слабых ссылок для родителя (например, `child.parent = weakref.ref(parent)`) разрывает цикл: когда объект-родитель удаляется, слабая ссылка перестаёт быть валидной, и цикл не препятствует сборке.
- `weakref.finalize` – вспомогательный инструмент для регистрации функции очистки при удалении объекта (альтернатива `__del__`). Он основан на слабых ссылках и гарантирует вызов коллбэка при сборе объекта.

Пример использования `WeakValueDictionary`:

```
import weakref
cache = weakref.WeakValueDictionary()

class BigData:
    pass

obj = BigData()
cache['data'] = obj

print('before:', 'data' in cache) # True

del obj # удаляем единственную сильную ссылку
import gc; gc.collect()

print('after:', 'data' in cache) # False (запись удалена автоматически)
```

Этот пример показывает, что после удаления `obj` и сборки, слабый словарь уже не содержит объекта.

Не все объекты поддерживают слабые ссылки (например, число, кортеж без `__weakref__` не имеют слота для слабых ссылок) ⁴¹. Классы можно сделать поддерживающими слабые ссылки, добавив `'__weakref__'` в `__slots__` или просто не используя `__slots__`.

Итого: слабые ссылки позволяют строить структуры (кэши, подписки, деревья) без риска удержания объектов «просто потому что есть ссылка» ⁴⁰.

8. Многопоточность и GC/RC

В стандартном CPython (с GIL) механизм подсчёта ссылок **безопасен** относительно многопоточности: GIL обеспечивает атомарность операций изменения refcount, поэтому отдельной синхронизации не требуется. Сборщик мусора также запускается под защитой GIL и в одном потоке: пока выполняется сборка, другие потоки блокируются.

При использовании библиотеки `threading` или `multiprocessing`:

- **Потоки:** в обычном режиме GIL препятствует одновременному выполнению Python-байткода разными потоками, что упрощает модель памяти и GC. Общие структуры данных (списки, словари) часто имеют внутренние замки для некоторых операций, но детальное поведение неконкурентных изменений не гарантируется (лучше использовать собственные `threading.Lock`). В новых экспериментальных сборках (см. ниже) GIL может быть отключён, тогда вводятся более сложные механизмы (Biased RC и т.д.) для безопасного обновления refcount.

- **Процессы:** каждый процесс имеет свою собственную память и собственный GC/RC. Объекты и указатели не разделяются между процессами (кроме специальных подходов, типа `multiprocessing.shared_memory`). Поэтому GIL/GC одного процесса не влияют на другой.

Важно: в стандартном CPython без GIL изменение refcount не защищено, могут возникать гонки. Новые ветки разработки CPython (PEP 703) решают эту проблему введением локальных и разделяемых счётчиков (biased reference counting) и даже «иммортализацией» некоторых объектов ⁴² ⁴³ (см. раздел 11).

При форке процесса (особенно в UNIX) состояние GC может привести к «двойному освобождению» объектов. Для этого в Python существуют `gc.freeze()` и `gc.unfreeze()` ³⁹: обычно после `fork()` дочерний процесс замораживает GC (чтобы не производились сборки пока продолжает существовать унаследованная память) и затем размораживает.

9. Пороговые значения GC (`gc.set_threshold`)

Генерационный сборщик управляется порогами, определяющими частоту сборок. Функция `gc.set_threshold(th0, th1, th2)` устанавливает пороги для трёх поколений. По умолчанию, Python запускает сборку поколения 0, когда (число выделений – число освобождений) превысит `th0`. Если сборка поколения 0 происходила уже `th1` раз без сборки поколения 1, то следующая сборка затронет и поколение 1. Аналогично для поколения 2 и порога `th2` ⁴⁴. Установка `threshold0=0` полностью отключает автоматическую сборку ⁴⁴.

Например:

```
import gc
print("Default thresholds:", gc.get_threshold()) # коптеж (th0,th1,th2)
gc.set_threshold(700, 10, 10) # меняем пороги
print("New thresholds:", gc.get_threshold())
```

Пороговые значения можно настраивать, исходя из характера приложения: если активно создаются многие короткоживущие объекты, можно понизить `threshold0`, чтобы быстрее очищать мусор. Если объём памяти большой, пороги можно увеличить, чтобы реже тратить время на GC. Суммарно: `threshold0` контролирует частоту сборок поколения 0 (молодые объекты), `threshold1` – частоту включения поколения 1, `threshold2` – старшего поколения.

10. Утечки памяти: обнаружение неосвобождённых объектов

Даже с GC бывают «утечки» — объекты, которые приложение больше не использует, но сборщик не может удалить (чаще всего из-за циклов с `__del__` или специальных удерживающих ссылок).

- **Неосвобождённые объекты:** после `gc.collect()` объекты, которые GC не смог удалить из-за циклов, остаются в списке `gc.garbage` ³³. Его стоит проверять при отладке: если там накапливаются объекты, значит, они недоступны по ссылкам, но удерживаются циклически. При включённом флаге `gc.DEBUG_SAVEALL`, такие объекты при сборке попадают именно в `gc.garbage` вместо немедленного удаления ³³.
- **Выявление:** для поиска утечек часто делают следующее: 1) выключить автоматический сборщик (`gc.disable()`), 2) выполнить подозрительный участок кода, 3) вручную вызвать `gc.collect()`, 4) посмотреть `gc.garbage` или посчитать количество объектов `len(gc.get_objects())`. Если после цикла работы программы количество отслеживаемых объектов растёт, это признак утечки.
- **Инструменты:** `tracemalloc` может помочь найти «утечки» выросшего потребления, а `gc.get_referrers(obj)` / `gc.get_referents(obj)` позволят найти цепочки ссылок, приводящие к тому, что объект удерживается в памяти.

С 3.4 с PEP 442 большинство «циничных» циклов с `__del__` очищается корректно ³³. Однако могут остаться утечки из-за C-расширений или не прямых циклов. Регулярная проверка `gc.garbage` после `gc.collect()` позволяет обнаруживать такие случаи.

11. Free-threading, Biased RC, Deferred RC

В Python 3.13+ появилась экспериментальная «free-threaded» (свободнопоточная) сборка, избавленная от GIL ⁴⁵. Для этого вводятся новые модели подсчёта ссылок:

- **Free-threading (отсутствие GIL):** CPython может быть собран с ключом `--disable-gil`, позволяя нативный параллелизм потоков. В этом режиме внутри CPython добавлены тонкие локи для встроенных типов (чтобы много-поточные изменения, например, списков, были безопасными) ⁴⁶ ⁴⁷. При запуске расширений, не поддерживающих отсутствие GIL, режим GIL можно временно вернуть. В статье «What's New in 3.13» указано, что в режиме free-threaded объекты становятся «иммортальными» (не изменяют refcount) для избежания издержек атомарности ⁴³ ⁴⁷.
- **Biased Reference Counting:** после отключения GIL простые инкремент/декремент счётчика ссылок недостаточны. В PEP 703 описывается схема *biased RC*: каждый объект хранит «локальный» счётчик ссылок для потока-владельца и «общий» счётчик для остальных. Операции в родном потоке не требуют атомарных операций – счётчик быстро растёт в локальном поле. Операции из других потоков корректируют «общий» счётчик атомарно. Поле `ob_ownership` указывает главный поток. Это позволяет снизить конкуренцию за

изменение счётчика ⁴². Некоторое снижение согласованности (задержка видимости удаления) компенсируется периодическим выравниванием.

- **Deferred Reference Counting:** для объектов, часто разделяемых между потоками (модули, функции, классы), в PEP 703 вводится режим «отложенного RC». Таким объектам (`Py_TPFLAGS_HAVE_DEFERRED_RC`) метит старшие биты счётчика, и их фактическое освобождение откладывается на циклы GC ⁴⁸ ⁴⁹. То есть освобождать их напрямую через `Py_DECREF` нежелательно: GC сам будет собирать их в своих циклах. Это снижает частоту инкрементов/декрементов для редко уничтожаемых объектов (убирая атомарные операции).
- **Иммортализация:** PEP 683 (Python 3.12) ввёл концепцию иммортальных объектов, у которых `refcount` никогда не меняется и они никогда не уничтожаются (например, некоторые внутренние объекты, малые числа, возможно синглтоны) ⁴³. Это повышает производительность: невозможно погасить `refcount` и повторно выделять их.

Эти изменения – большая часть новой GC-модели Python 3.12–3.13. Для программиста важно знать, что в стандартном режиме (с GIL) это всё невидимо: модели подсчёта ссылок и GC работают как раньше. Новый движок ориентирован на сценарии высокопараллельности: ожидаются улучшения производительности при многопоточности, но возможен сниз single-threaded speed.

12. Структура `PyObject` в CPython

В CPython каждый объект представляет собой C-структуру. Базовый тип `PyObject` содержит минимум два поля:

- `ob_refcnt` – счётчик сильных ссылок на объект.
- `ob_type` – указатель на описание типа (тип-объект) этого объекта.

В документации говорится: «в релизной сборке `PyObject` содержит только счётчик ссылок и указатель на объект типа» ⁵⁰. На эти поля можно посмотреть через макросы `Py_REFCNT(obj)` и `Py_TYPE(obj)` ⁵⁰.

Для объектов переменного размера (последовательности, строки и т.д.) используется `PyVarObject`, который **расширяет** `PyObject`, добавляя поле `ob_size` – размер или длину (например, количество элементов в списке) ⁵¹. Макрос `Py_SIZE(obj)` даёт значение `ob_size`. Таким образом, полная структура включает:

```
typedef struct {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;

typedef struct {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
    Py_ssize_t ob_size;
} PyVarObject;
```

Поле `ob_size` нужно только у переменных объектов; у объектов фиксированного размера его нет. С точки зрения памяти, у простого объекта в 64-битной системе уже есть немалый оверхед (обычно 16 байт: 8 байт для `refcnt` и 8 для указателя типа) ⁵⁰.

Тип объекта описывает, какие операции над ним возможны (вызывается через `ob_type->tp_*`). Поле `ob_type` позволяет быстро по типу объекта определить, например, как сравнивать или как его удалять.

Таким образом, внутренне объект CPython хранит указатель на свой класс и счётчик ссылок ⁵⁰. О реальном размере объекта (например, списка) дополнительно хранится длина/размер (в `ob_size`) ⁵¹. Эти детали влияют на поведение сборщика и на то, какие объекты отслеживаются (например, некоторые простые объекты вообще не отслеживаются GC, см. ниже в `gc.is_tracked` ⁵²).

13. Новый сборщик мусора в Python 3.12–3.13

Начиная с Python 3.12, сборщик мусора работает по-другому: **GC больше не запускается после каждого выделения**. Теперь он срабатывает только в определённых «точках останова» байткода – механизме *eval breaker* – и при проверке сигналов ⁵³. Это снижает накладные расходы на проверку сборщика после каждой аллокации (в предыдущих версиях CPython мог добавлять счётчик выделений по побайтовым циклам). Также GC теперь может срабатывать в ответ на `PyErr_CheckSignals()` – чтобы C-расширения, долго выполняющие код, тоже давали шанс GC запуститься ⁵³.

При этом поколения остались тремя, но теперь внутренние списки мусора реализованы иначе: в PEP 703 описано снятие необходимости в двусвязном списке `gc_head` (поля `_gc_prev`/`_gc_next` в структуре) ⁵⁴. Информация о том, отслеживается ли объект сборщиком, хранится теперь в битах внутри объекта, а не в отдельной структуре – это уменьшает оверхед объектов GC ⁵⁴.

Python 3.13 вводит экспериментальный *free-threading* (см. раздел 11), что тоже требует изменений в GC: некоторые объекты (модули, функции, классы) становятся иммортальными – никогда не деаллоцируются, а их `refcount` не меняется ⁴³ ⁴⁷. Это сделано для избежания гонок за `refcount`.

Таким образом, в Python 3.12–3.13 ключевые отличия старой и новой модели сборщика мусора:

- **Триггеры GC:** раньше GC запускался при достижении порога при каждом выделении, теперь – только при специальных «остановках» выполнения (*eval-breaker*, сигналы) ⁵³.
- **Учет поколений:** три поколения сохранены, но внутреннее представление изменено (отказ от ссылок `_gc_prev/_gc_next`, теперь используются флаги в объекте) ⁵⁴.
- **Immortal objects:** введены бессмертные объекты (PEP 683) ⁴³, что позволяет не тратить ресурсы на их сбор.
- **Free-threaded CPython:** в 3.13 появляется возможность отключать GIL, при этом мусорщик учитывает мультипоточность (*Biased* и *Deferred RC*, «заморозка» объектов в GC). Для разработчика это в основном невидимо, но нужно знать, что новые версии могут по-другому обращаться с `refcount` и GC под капотом.

Эти нововведения пока экспериментальны, но закладывают основу для более масштабных изменений подсчёта ссылок и сборки мусора в будущем.

Источники: официальная документация Python по языку и модулям (Data model, psutil, resource, tracemalloc, gc, weakref), статьи и РЕРы по внутреннему устройству CPython
1 2 3 55 6 13 16 24 56 23 25 57 58 31 32 33 50 43 42 53 .

1 2 3. Data model — Python 3.13.3 documentation

<https://docs.python.org/3/reference/datamodel.html>

3 Is `0 is 0` always `True` in Python? - Stack Overflow

<https://stackoverflow.com/questions/62175978/is-0-is-0-always-true-in-python>

4 5 6 7 8 9 10 11 12 13 14 15 16 psutil documentation — psutil 7.0.1 documentation

<https://psutil.readthedocs.io/>

17 18 19 20 21 22 23 24 56 resource — Resource usage information — Python 3.13.3 documentation

<https://docs.python.org/3/library/resource.html>

25 26 27 28 29 30 31 57 58 tracemalloc — Trace memory allocations — Python 3.13.3 documentation

<https://docs.python.org/3/library/tracemalloc.html>

32 Reference Counting — Python 3.13.3 documentation

<https://docs.python.org/3/c-api/refcounting.html>

33 34 35 36 37 38 39 44 52 gc — Garbage Collector interface — Python 3.13.3 documentation

<https://docs.python.org/3/library/gc.html>

40 41 weakref — Weak references — Python 3.13.3 documentation

<https://docs.python.org/3/library/weakref.html>

42 48 49 54 PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org

<https://peps.python.org/pep-0703/>

43 46 Python experimental support for free threading — Python 3.13.3 documentation

<https://docs.python.org/3/howto/free-threading-python.html>

45 47 What's New In Python 3.13 — Python 3.13.3 documentation

<https://docs.python.org/3/whatsnew/3.13.html>

50 51 Common Object Structures — Python 3.13.3 documentation

<https://docs.python.org/3/c-api/structures.html>

53 What's New In Python 3.12 — Python 3.13.3 documentation

<https://docs.python.org/3/whatsnew/3.12.html>

55 memoization - Does Python intern strings? - Stack Overflow

<https://stackoverflow.com/questions/17679861/does-python-intern-strings>