Сборка мусора в Python за 5 минут

Описание

Хотите узнать, как Python управляет внутренней памятью? Если да, то это руководство по сборке мусора в Python для вас. При программировании на Python вам обычно не приходится беспокоиться о выделении и удалении памяти. Однако полезно понимать, как Python справляется с этой задачей под капотом с помощью процесса сборки мусора. В этом уроке мы рассмотрим сборку мусора и ее важность. Затем мы рассмотрим, как Python использует подсчет ссылок для удаления объектов, которые больше не используются.

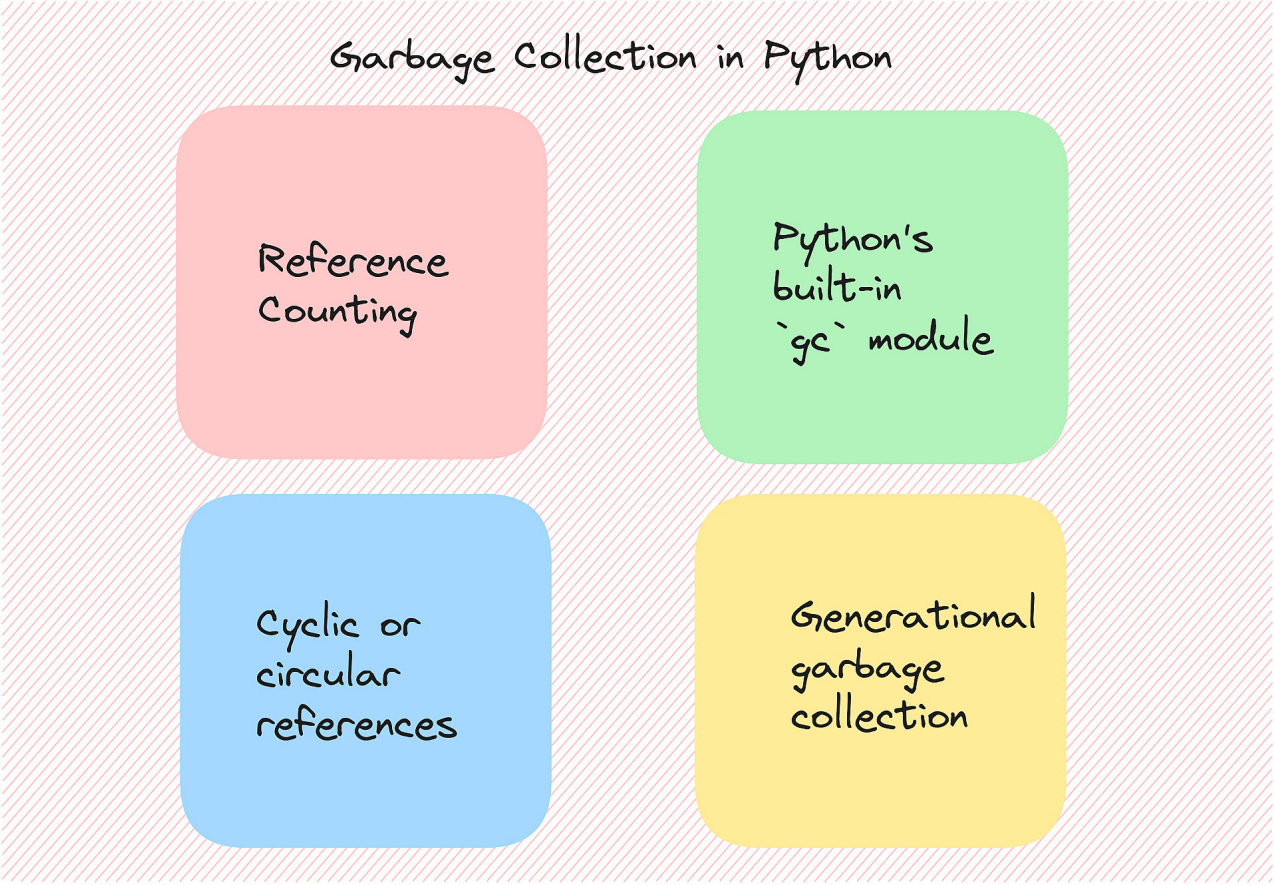
Что такое сборка мусора и почему она важна?

В Python вы не так часто сталкиваетесь с ошибками нехватки памяти, но это возможно. Когда вы создаете объекты в программе, они занимают память. И, скорее всего, будет много объектов, которые больше не используются. Что произойдет, если память, занимаемая такими объектами, никогда не будет освобождена? Ну, в конце концов, память закончится. Введите сборку мусора. Сборка мусора – это процесс, который отвечает за автоматическое определение и возврат памяти, занятой объектами, которые больше не используются, что позволяет предотвратить утечки памяти и повысить общую эффективность ее использования.

Как работает сборка мусора в Python?

Теперь, когда мы поняли, что такое сборка мусора и ее значение, давайте

посмотрим, как Python использует сборку мусора.



Понимание переменных и объектов в Python

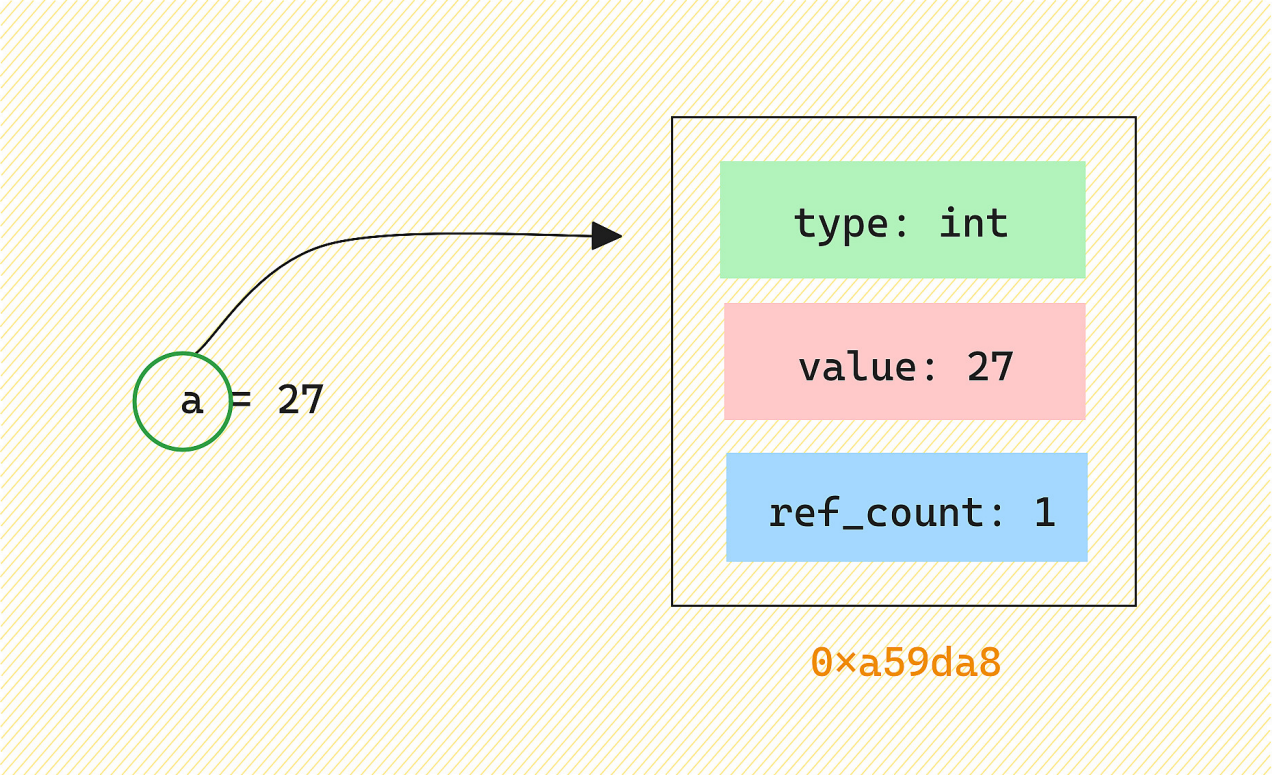
Вы, вероятно, привыкли думать о переменных как о контейнерах, в которых хранятся значения. Однако в Python полезно описывать переменные как **“имена”** или **“метки”** для объектов, а не как контейнеры, в которых хранятся объекты. Это фундаментальная концепция для понимания того, как работают переменные в Python. Давайте посмотрим поближе. Когда вы создаете переменную в Python, вы, по сути, создаете ссылку на объект в памяти. Переменная – это метка или имя, которое указывает на область памяти, где хранится объект. Она не хранит сам объект, но *“ссылается”* или *“указывает”* на него. Вот пример:

```
a = 27
```

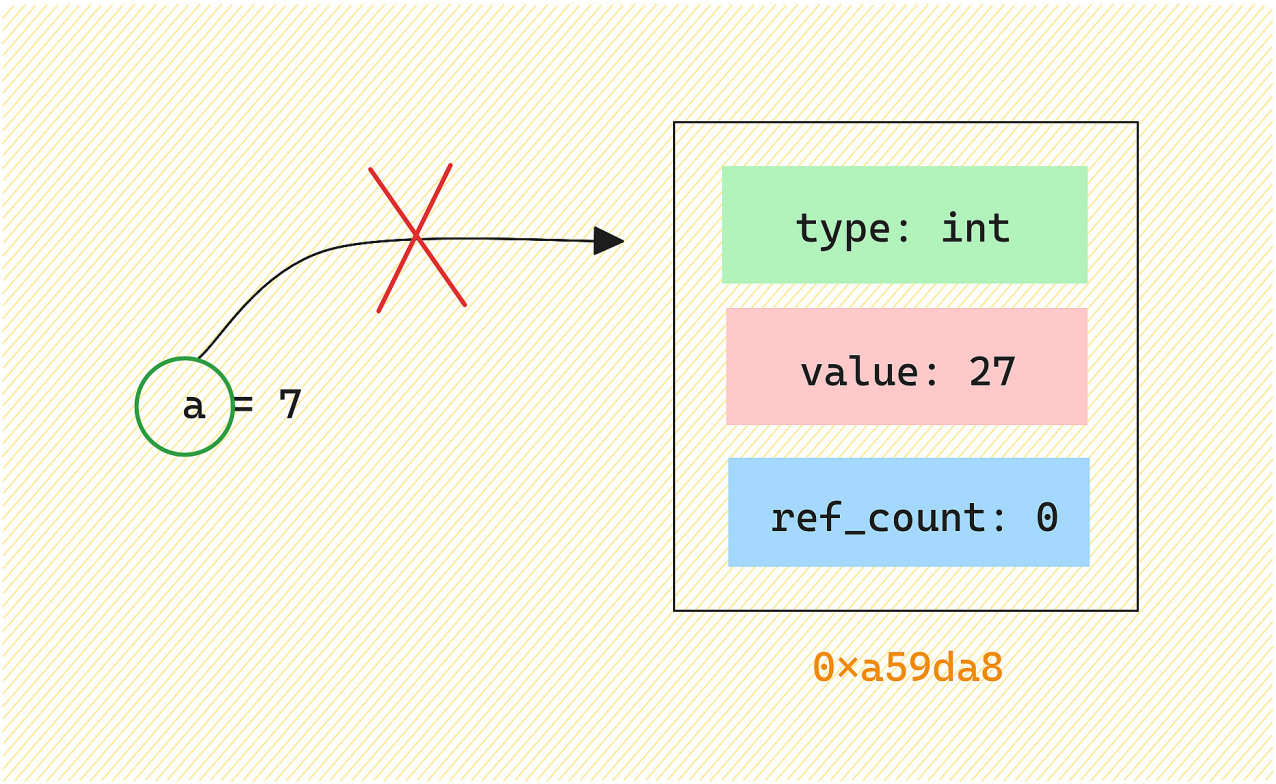
Когда вы создаете `= 27`:

- В памяти создается объект типа `int`.
- Он принимает значение 27.

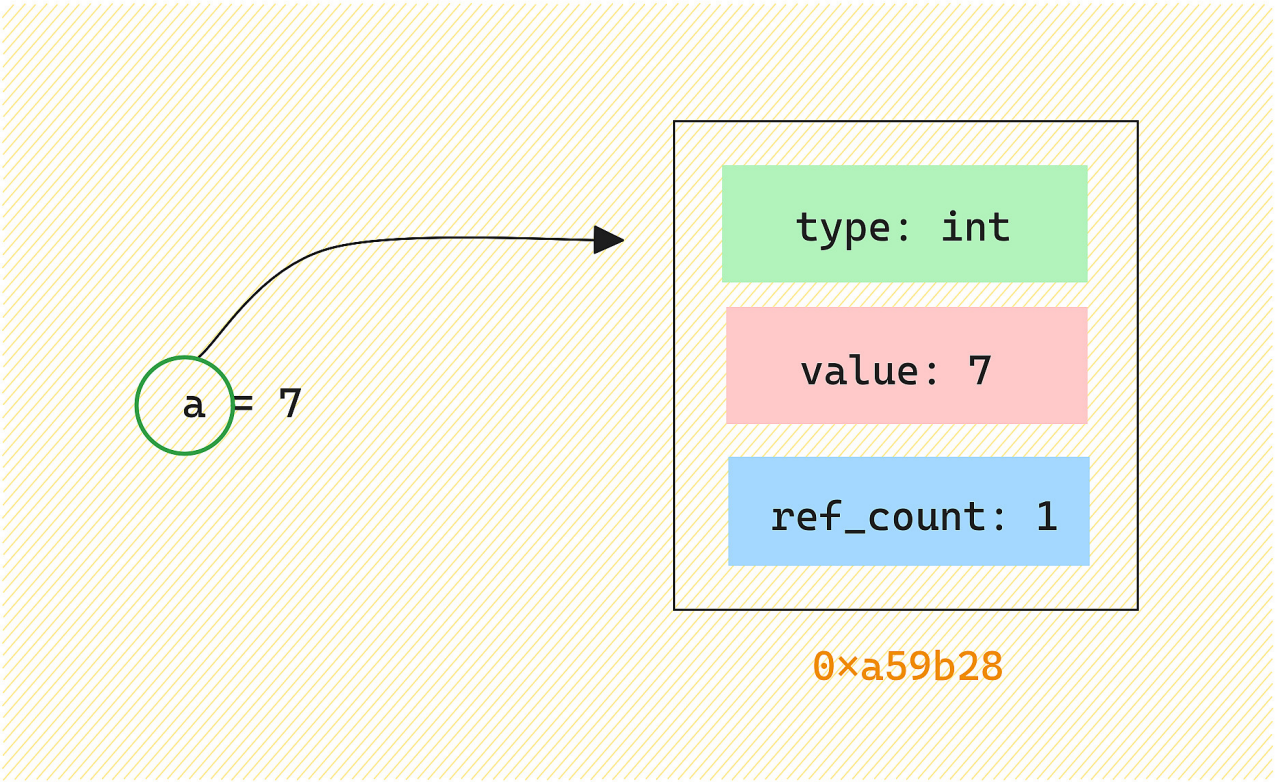
- Переменная a указывает на этот объект или ссылается на него. Поэтому счетчик ссылок, связанный с объектом, равен 1. (Мы подробно обсудим счетчик ссылок в следующем разделе).



Примечание: Вы можете выполнить команду `hex(id(a))`, чтобы получить адрес в памяти объекта, на который указывает метка a. Что произойдет, если вы измените значение a? Теперь a больше не указывает на предыдущий адрес памяти.



Переменная a теперь указывает на целочисленный объект со значением 7.



Подсчет ссылок

Подсчет ссылок – это техника управления памятью, используемая в Python для отслеживания количества ссылок на объект. Идея заключается в том, чтобы присвоить каждому объекту счетчик ссылок и увеличивать или уменьшать этот счетчик по мере создания или удаления ссылок на объект. Когда счетчик ссылок опускается до нуля, что означает, что на объект больше нет ссылок, память, занимаемая объектом, может быть освобождена. **Так что же такое счетчик ссылок?** Как уже говорилось, каждый объект Python имеет количество ссылок, связанных с ним. Это количество переменных, указывающих на него. Когда мы создаем новую ссылку на объект, счетчик ссылок увеличивается. Когда мы удаляем ссылку (переменная выходит из области видимости или явно устанавливается в `None`), счетчик ссылок уменьшается. **А как происходит сборка мусора?**

- Когда счетчик ссылок на объект падает до нуля, это означает, что на этот объект больше нет ссылок.
- После этого память, занятая объектом, может быть восстановлена.
- В Python используется сборщик мусора, который периодически определяет и собирает объекты, на которые больше нет ссылок, и освобождает связанную с ними память.

```
import sys

a = [1,2,3]
print(sys.getrefcount(a))  # Prints 2 because getrefcount itself creates a ref

b = a # reference count increases by 1
print(sys.getrefcount(a))  # Prints 3

del b # one reference removed
print(sys.getrefcount(a))  # Prints 2
```

Вы получите следующий результат:

```
# Output
2
3
2
```

Принцип работы довольно прост, но давайте пройдемся по шагам:

- Изначально количество ссылок в списке `[1, 2, 3]` равно 1.
- Поскольку в качестве аргумента функции `getrefcount()` используется

временная ссылка на a, счетчик равен 2 (на один больше, чем ожидалось).

- Когда мы присваиваем a к b, количество ссылок становится равным 3.
- Когда мы удаляем b, количество ссылок снова становится равным 2.

Хотя подсчет ссылок является основой управления памятью в Python, стоит упомянуть, что Python также использует циклическое определение ссылок для обработки более сложных случаев, таких как круговые ссылки (подробнее об этом позже).

Встроенный в Python модуль gc

Теперь давайте познакомимся с модулем gc для сборки мусора. Вы можете явно запустить сборщик мусора в своем Python-скрипте следующим образом:

```
import gc

class MyClass:
    def __init__(self, name):
        self.name = name

# Create some objects
obj1 = MyClass('Object 1')
obj2 = MyClass('Object 2')
obj3 = MyClass('Object 3')

# Get all objects tracked by the garbage collector
all_objects = gc.get_objects()

# Print information about each object
for obj in all_objects:
    if isinstance(obj, MyClass):
        print(f'{obj.name}:{obj}')

# Manually trigger garbage collection
gc.collect()
```

Использование функции `get_objects()` из модуля `gc` даст вам список всех объектов, известных сборщику мусора. Поскольку этот список может быть непомерно большим, мы постарались получить информацию только об объектах, принадлежащих `MyClass`.

```
# Output
Object 1:<__main__.MyClass object at 0x7f96e7421510>
Object 2:<__main__.MyClass object at 0x7f96e74219d0>
Object 3:<__main__.MyClass object at 0x7f96e7421990>
```

> **Примечание**: Вы можете использовать `gc.disable()`, чтобы вручную отключить автоматическую сборку мусора. Но это не рекомендуется, если вам не нужен детальный контроль над сценарием.

Циклические ссылки

Циклические (или круговые) ссылки возникают, когда группа объектов ссылается друг на друга таким образом, что образуется цикл. Если на эту группу не существует внешних ссылок, это может привести к утечке памяти. Потому что количество ссылок для каждого объекта никогда не достигает нуля, и объекты никогда не собираются в мусор. Вот простой пример круговой ссылки:

```
# Create an empty set
my_set = set()

# Add circular reference
my_set.add(my_set)
```

Теперь `my_set` содержит ссылку на себя, создавая круговую ссылку. Помимо подсчета ссылок, Python поддерживает циклическую сборку мусора для обнаружения таких круговых ссылок. Кроме того, в нем используется поколенческий подход к сборке мусора.

Поколенческий подход к сбору мусора

Поколенческая сборка мусора в Python основана на наблюдении, что большинство объектов в программе имеют короткий срок жизни. Объекты делятся на разные поколения в зависимости от того, как долго они живут – сколько раз они пережили сборку, – и сборщик мусора применяет к этим поколениям разные стратегии сбора. Поколенческая сборка мусора в Python обычно делит объекты на три поколения:

Поколение 0

- В этом поколении начинаются вновь созданные объекты.
- Объекты, пережившие цикл сборки мусора в этом молодом поколении, переходят в следующее, более старшее поколение.

Поколение 1

- Объекты, пережившие несколько циклов сборки мусора в молодом поколении, переходят в это среднее поколение.
- Сборка мусора в этом поколении происходит реже, чем в поколении 0.

Поколение 2

- Объекты, пережившие множество циклов сборки мусора в поколении 1, переходят в поколение 2, самое старое поколение.
- Сборка мусора в этом поколении происходит еще реже.

Таким образом, поколенческая сборка мусора основана на гипотезе поколений, которая гласит, что молодые объекты с большей вероятностью станут мусором, чем старые. Собирая молодое поколение чаще, а старое – реже, сборщик мусора может добиться большей производительности.

Преимущества сбора мусора

Мы уже обсуждали, почему сборка мусора важна. Давайте повторим преимущества:

- Сборка мусора автоматизирует процесс освобождения памяти, занятой объектами, которые больше не используются, освобождая разработчиков от ручного управления памятью.
- Он помогает предотвратить утечки памяти, выявляя и очищая недоступные объекты.
- Сборка мусора делает приложения Python более надежными, снижая риск сбоев и неожиданного поведения из-за проблем с памятью.

Лучшие практики сбора мусора в Python

Давайте перечислим несколько лучших практик, позволяющих использовать автоматическую сборку мусора и при этом писать эффективный Pythonic-код:

Используйте автоматическую сборку мусора: В Python встроена автоматическая сборка мусора, поэтому в большинстве случаев вам не придется вручную управлять памятью. Не отключайте сборщик мусора, если в этом нет

крайней необходимости.

Используйте контекстные менеджеры для ресурсов: При работе с внешними ресурсами, такими как файлы или сетевые соединения, используйте менеджеры контекста ? операторах, чтобы обеспечить правильное закрытие и освобождение ресурсов. Это снижает вероятность утечки ресурсов.

Мониторинг и профилирование использования памяти: Используйте модули gc и tracemalloc для мониторинга и профилирования использования памяти в ваших приложениях Python, чтобы выявить узкие места в производительности и потенциальные утечки памяти.

Оптимизируйте использование памяти: Сведите к минимуму создание ненужных объектов и рассмотрите возможность использования подходящих встроенных структур данных и итераторов, экономящих память, например генераторов. Кроме того, используйте подходящие типы данных, чтобы избежать лишних затрат памяти.

Заключение

В этой статье мы рассказали о том, как работают сборка мусора и управление памятью в Python. Давайте рассмотрим то, что мы узнали. Мы узнали, как Python использует подсчет ссылок для удаления ссылок на объекты, которые больше не используются. Затем мы рассмотрели, как Python обрабатывает циклические ссылки и генерационный подход к сборке мусора. Затем мы рассмотрели преимущества сборки мусора и в завершение обсудили некоторые лучшие практики сборки мусора в Python.

Дата Создания
24.04.2024