Python Memory Management

1. https://docs.python.org/3/c-api/memory.html#:~:text=Memory%20management%20in%20Python%20involves,by%20the%20Python%20memory%20manager.
2. https://code.tutsplus.com/ru/tutorials/understand-how-much-memory-your-python-objects-use--cms-25609
3. https://www.geeksforgeeks.org/memory-management-in-python/
4. **https://codechick.io/tutorials/python/python-tuple-list**

Понимание распределения памяти важно для любого разработчика программного обеспечения, поскольку написание эффективного кода означает написание кода с эффективным использованием памяти. Выделение памяти можно определить как выделение блока памяти в памяти компьютера программе. В Python метод выделения и освобождения памяти является автоматическим, поскольку разработчики Python создали сборщик мусора для Python, чтобы пользователю не приходилось выполнять сборку мусора вручную.

Garbage Collection:

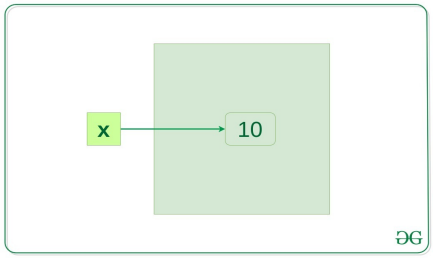
Сборка мусора — это процесс, в котором интерпретатор освобождает память, когда она не используется, чтобы сделать ее доступной для других объектов. Предположим, что ссылка не указывает на объект в памяти, т. е. он не используется, поэтому виртуальная машина имеет сборщик мусора, который автоматически удаляет этот объект из кучи памяти.
Примечание. Дополнительные сведения см. в разделе Сборка мусора в Python .

Подсчет ссылок:

Подсчет ссылок работает путем подсчета количества ссылок на объект другими объектами в системе. Когда ссылки на объект удаляются, счетчик ссылок на объект уменьшается. Когда счетчик ссылок становится равным нулю, объект освобождается.
Например, предположим, что есть две или более переменных с одинаковым значением, поэтому виртуальная машина Python вместо создания другого объекта с тем же значением в частной куче фактически заставляет вторую переменную указывать на исходную. существующее значение в частной куче. Следовательно, в случае с классами количество ссылок может занимать много места в памяти, в таком случае подсчет ссылок очень полезен для сохранения памяти, доступной для других объектов.

```
x = 10
```

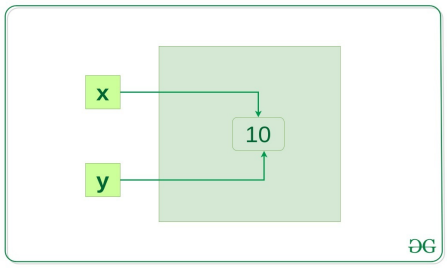
При `x = 10` выполнении в памяти создается целочисленный объект 10, и его ссылка присваивается переменной x, потому что в Python все является объектом.




```
x = 10
y = x
if id(x) == id(y):
 print("x and y refer to the same object")

Output:
x and y refer to the same object
```

В приведенном выше примере `y = x` будет создана другая ссылочная переменная y, которая будет ссылаться на тот же объект, поскольку Python оптимизирует использование памяти, выделяя ту же ссылку на объект новой переменной, если объект уже существует с тем же значением.



Memory Allocation in Python

There are two parts of memory:

- stack memory
- heap memory

Вызовы методов/методов и ссылки хранятся в **stack memory,** а все объекты значений хранятся в**private heap**.

Встроенная функция sys.getsizeof()

Модуль sys стандартной библиотеки предоставляет функцию getsizeof(). Эта функция принимает объект (и необязательный параметр по умолчанию), вызывает метод **sizeof()** объекта и возвращает результат, поэтому вы также можете сделать ваши объекты инспектируемыми.

Кортежи занимают меньше памяти, чем списки!

Список — изменяемая последовательность. Это означает, что в него можно добавлять новые элементы. Из-за этого Python приходится выделять для списка больше памяти, чем нужно. Это называется перераспределением. Избыточное выделение повышает производительность при расширении списка, но увеличивает объект памяти.

Кортеж — неизменяемая последовательность, поэтому количество его элементов фиксировано. Это позволяет Python просто выделить достаточное количество памяти для хранения элементов, заданных при создании.

В результате кортеж в большинстве случае занимает меньше памяти, чем список.

```
from sys import getsizeof
fruits = ['яблоко', 'апельсин', 'банан']
print(f'Размер памяти для списка — {getsizeof(fruits)} байтов.')
fruits = ('яблоко', 'апельсин', 'банан')
print(f'Размер памяти для списка — {getsizeof(fruits)} байтов.')

Output:
Размер памяти для списка — 80 байтов.
Размер памяти для кортежа — 64 байтов.
```

Копирование кортежа быстрее, чем списка

Когда вы копируете список, Python создает новый список. Рассмотрим этот процесс на следующем примере

```
fruit_list = ['яблоко', 'апельсин', 'банан']
fruit_list2 = list(fruit_list)
print(id(fruit_list) == id(fruit_list2))  # Вывод: False
```

Однако когда вы копируете кортеж, Python повторно использует уже существующий кортеж. Он не создает новый кортеж, поскольку кортежи неизменяемы.

```
fruit_tuple = ('яблоко', 'апельсин', 'банан')
fruit_tuple2 = tuple(fruit_tuple)
print(id(fruit_tuple) == id(fruit_tuple2))  # Вывод: True
```

В результате копирование кортежа всегда проходит немного быстрее, чем копирование списка. Давайте проверим это на следующем примере. Сравним время, необходимое для копирования списка и кортежа 1 миллион раз:

```
from timeit import timeit

times = 1_000_000

t1 = timeit("list(['яблоко', 'апельсин', 'банан'])", number=times)
print(f'Время для копирования списка {times} раз: {t1}')
```

```
t2 = timeit("tuple(('яблоко', 'апельсин', 'банан'))", number=times)
print(f'Время для копирования кортежа {times} раз: {t2}')

diff = "{:.0%}".format((t2 - t1)/t1)

print(f'Разница: {diff}')

Output:
Время для копирования списка 1000000 раз: 0.12854695800000115
Время для копирования кортежа 1000000 раз: 0.03695795800000212
Разница: -71%
```

Что нужно запомнить:

- Кортеж неизменяем, список — изменяем.
- Кортеж занимает меньше памяти, чем список.
- Копирование кортежа происходит немного быстрее, чем копирование списка.
- Используйте кортеж, если вам нужен список, который вы не собираетесь изменять.