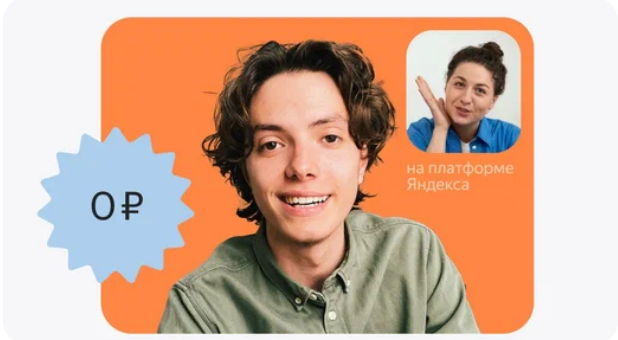


Оптимизация потребления памяти в Python



practicum.yandex.ru

РЕКЛАМА · 18+ Я

Бесплатное занятие английским в Яндекс Практикуме

Полноценное занятие с преподавателем, а не презентация курсов

[Узнать больше](#)

[Справочник по языку Python3.](#) / Оптимизация потребления памяти в Python

Когда дело доходит до оптимизации производительности, люди обычно сосредотачиваются только на скорости и использовании ЦП. Редко кого волнует [потребление оперативной памяти](#), но только до тех пор, пока она не кончится.

Предлагаемый материал рассматривает методы определения того, какие части приложений Python потребляют слишком много памяти. Также рассматривает способы сокращения потребления памяти, используя простые приемы и структуры данных, эффективно использующие память.

Существует множество причин, по которым стоит попытаться ограничить использование памяти:

1. Предотвращение сбоя приложения из-за ошибок нехватки оперативной памяти.
2. Ресурсы - как ЦП, так и ОЗУ стоят денег, зачем тратить память на запуск неэффективных приложений, если есть способы уменьшить объем памяти?
3. Данные имеют "массу", и если их много, то они будут перемещаться медленно. Если данные должны храниться на диске, а не в ОЗУ или быстром кэше, то загрузка и обработка займет некоторое время, что повлияет на общую производительность.

Таким образом, оптимизация использования памяти может иметь хороший побочный эффект в виде ускорения времени выполнения приложения.

Содержание:

- [Ищем узкие места;](#)
 - [Модуль memory_profiler;](#)
 - [Модуль Pympiler;](#)
- [Как экономить оперативную память;](#)
 - [Использование генераторов Python и/или модуль mmap.](#)

Ищем узкие места.

Прежде чем приступить к работе по уменьшению использования памяти приложением, сначала нужно найти узкие места или части кода, которые занимают всю память.

Модуль memory_profiler.

Первый инструмент, который предлагается использовать, это [сторонний модуль memory_profiler](#). Модуль memory_profiler измеряет использование памяти конкретной функцией построчно.

Чтобы начать использовать memory_profiler, установим его в [виртуальное окружение](#) с помощью [менеджера пакетов pip](#) вместе с пакетом [psutil](#), который значительно повышает производительность профилировщика.

```
# создаем виртуальное окружение, если нет
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# обновляем `pip`
(VirtualEnv):~$ python3 -m pip install -U pip
# ставим модули `memory_profiler` и `psutil`
(VirtualEnv):~$ python3 -m pip install memory_profiler psutil
```

Теперь, в тестируемом сценарии необходимо украсить/пометить функцию, которую нужно протестировать, при помощи декоратора `@profile`.

Например:

```
# абстрактный код
@profile
def memory_intensive():
    ...

if __name__ == '__main__':
    memory_intensive()
```

Далее запускаем тестируемый сценарий `test-code.py` следующим образом:

```
(VirtualEnv):~$ python3 -m memory_profiler test-code.py

# Line #      Mem usage      Increment  Occurrences   Line Contents
# =====
#      15    39.113 MiB    39.113 MiB           1   @profile
#      16                                     def memory_intensive():
#      17    46.539 MiB     7.426 MiB           1       small_list = [None] * 1000000
#      18   122.852 MiB    76.312 MiB           1       big_list = [None] * 10000000
#      19    46.766 MiB   -76.086 MiB           1       del big_list
#      20    46.766 MiB     0.000 MiB           1       return small_list
```

Вывод показывает использование/распределение памяти построчно для декорированной функции - в данном случае `memory_intensive()`, которая преднамеренно создает и удаляет большие списки. Первый столбец (`Line`) представляет номер строки кода, который был профилирован, второй столбец (`Mem usage`) - использование памяти интерпретатором Python после выполнения этой строки. Третий столбец (`Increment`) представляет разницу в памяти текущей строки по отношению к последней. Последний столбец (`Line Contents`) печатает профилированный код.

Примечание. Если импортировать декоратор `@profile` из модуля `memory_profiler`, то скрипт можно запустить без указания `-m memory_profiler` (как обычный сценарий `$ python3 test.py`).

```
# test.py
from memory_profiler import profile

@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

Теперь, когда область поиска проблем сузилась до определенных строк кода, можно копнуть немного глубже и посмотреть, сколько памяти использует каждая переменная. Для этого можно использовать функции [sys.getsizeof\(\)](#), но она дает сомнительную информацию для некоторых типов структур данных. Для [целых чисел int](#) или [массивов байтов bytearray](#) получим реальный размер в байтах, однако для контейнеров, таких как список, получим только размер самого контейнера, а не его содержимого:

```
>>> import sys
>>> sys.getsizeof(1)
# 28
>>> sys.getsizeof(2**30)
# 32
>>> sys.getsizeof(2**60)
# 36

>>> sys.getsizeof("a")
# 50
>>> sys.getsizeof("aa")
```

```
# 51
>>> sys.getsizeof("aaa")
# 52

>>> sys.getsizeof([])
# 56
>>> sys.getsizeof([1])
# 64

# Обратите внимание на вывод. Пустой список
# равен 56, а каждое значение внутри равно 28.
>>> sys.getsizeof([1, 2, 3, 4, 5])
# 96
```

Из примеров видно, что с простыми целыми числами, когда пересекается порог, то к размеру добавляется 4 байта. Точно так же с простыми строками, при добавлении нового символа, добавляется один дополнительный байт. Но со списками это не работает - `sys.getsizeof` не "обходит" структуру данных и возвращает только размер родительского объекта, в данном случае списка.

Лучшим подходом является использование специального инструмента, предназначенного для анализа поведения памяти. Одним из таких инструментов является [сторонний модуль Pympler](#), который помогает получить более реалистичное представление о размерах объектов Python.

Модуль Pympler.

Установим модуль Pympler в виртуальное окружение:

```
# ставим модуль `Pympler`
(VirtualEnv):~$ python3 -m pip install Pympler
```

Для исследования того, сколько памяти потребляют определенные объекты Python можно использовать функцию `pympler.asizeof.asizeof()`. В отличие от от встроенной функции `sys.getsizeof()`, она рекурсивно изменяет размеры объектов. Кроме того, этот модуль также имеет функцию `pympler.asizeof.asized()`, которая дает дополнительную разбивку по размеру отдельных компонентов объекта.

```
>>> from pympler import asizeof
>>> asizeof.asizeof([1, 2, 3, 4, 5]))
# 256

>>> asizeof.asized([1, 2, [3, 4], "string"], detail=1).format()
# [1, 2, [3, 4], 'string'] size=344 flat=88
#     [3, 4] size=136 flat=72
#     'string' size=56 flat=56
#     1 size=32 flat=32
#     2 size=32 flat=32
```

Модуль Pympler имеет гораздо больше функций, включая [отслеживание экземпляров классов или выявление утечек памяти](#).

Как экономить оперативную память.

Теперь нужно найти способ исправить те проблемы, которые были обнаружены на предыдущем этапе. Потенциально самым быстрым и простым решением может быть переход на более эффективные с точки зрения памяти структуры данных.

[Списки list в Python](#) - один из наиболее требовательных к памяти вариантов, когда речь идет о хранении массивов значений.

Создадим простую функцию `allocate()`, которая создает список чисел, используя указанный размер `size`. Чтобы измерить, сколько памяти он занимает, используем [модуль memory_profiler](#). Он даст объем памяти во время выполнения этой функции с интервалом в 0,2 секунды.

```
>>> from memory_profiler import memory_usage
>>> def allocate(size):
...     some_var = [n for n in range(size)]

# `1e7` равно 10 в степени 7
usage = memory_usage((allocate, (int(1e7),)))
```

```
# Использование с течением времени
>>> usage
# [
#     43.671875, 43.73828125,
#     176.8671875, 314.8828125,
#     343.9140625, 43.8671875
# ]
# Пиковое использование
>>> max(usage)
# 343.9140625
```

Можно заметить, что для создания списка из 10 миллионов чисел требуется более 350 МБ памяти. Это однозначно много для списка цифр. Можно ли сделать лучше?

```
>>> from memory_profiler import memory_usage
>>> import array
>>> def allocate(size):
...     some_var = array.array('l', range(size))

# запускаем `allocate()`
>>> usage = memory_usage((allocate, (int(1e7),)))
# Использование с течением времени
>>> usage
# [
#     39.71484375, 39.71484375, 55.34765625,
#     71.14453125, 86.54296875, 101.49609375,
#     39.73046875
# ]

# Пиковое использование
>>> max(usage)
# 101.49609375
```

В этом примере использовался [модуль array](#), который может хранить примитивы, такие как целые числа или символы. Из результатов видно, что использование памяти достигло пика чуть более 100 МБ. Это огромная разница по сравнению с использованием списка. Можно дополнительно уменьшить использование памяти, указав при создании массива соответствующую [точность хранимого типа](#).

Одним из основных недостатков использования модуля array в качестве контейнера данных является то, что он поддерживает не так много типов. Если планируется выполнять множество математических операций с данными, то лучше использовать массивы |NumPy|:

```
>>> from memory_profiler import memory_usage
>>> import numpy as np
>>> def allocate(size):
...     some_var = np.arange(size)

# запускаем `allocate()`
>>> usage = memory_usage((allocate, (int(1e7),)))
# Использование с течением времени
>>> usage
# [
#     52.0625, 52.25390625, ...,
#     97.28515625, 107.28515625, ...,
#     123.28515625, 52.0625
# ]

# Пиковое использование
>>> max(usage)
# 123.28515625
```

Массивы NumPy работают довольно неплохо, использование памяти с пиковым размером массива - 123 МБ, что немного больше, чем array. Но с NumPy можно воспользоваться преимуществами быстрых математических функций, а также типов, которые не поддерживаются array, например, таких как комплексные числа.

Также можно внести некоторые улучшения в размер отдельных объектов, которые определяются пользовательскими классами. Это можно сделать с помощью [атрибута класса __slots__](#), который используется для явного объявления свойств класса. Объявление __slots__ в классе также имеет хороший побочный эффект, заключающийся в запрете создания атрибутов __dict__ и __weakref__:

```
>>> from pympler import asizeof
>>> class Normal:
...     pass
...
>>> class Smaller:
...     __slots__ = ()
...
>>> asizeof.asized(Normal(), detail=1).format()
# '<__main__.Normal object at 0x7fbf3cb2ecd0> size=152 flat=48'
#   __dict__ size=104 flat=104\n   __class__ size=0 flat=0'

>>> asizeof.asized(Smaller(), detail=1).format()
# '<__main__.Smaller object at 0x7fbf2b5890a0> size=32 flat=32'
#   __class__ size=0 flat=0'
```

Здесь видно, насколько меньшим оказался экземпляр класса Smaller(). Отсутствие __dict__ удаляет целых 104 байта из каждого экземпляра, что может сэкономить огромное количество памяти при создании экземпляров миллионов значений.

Приведенные выше приемы должны помочь при работе с числовыми значениями, а также с объектами классов. А что делать со [строками str Python](#)? Как правило, это зависит от того, что с ними будет делать программа. Если необходим поиск в огромном количестве строковых значений, то, [как видели ранее](#), использование [списка list](#) - очень плохая идея. Использование [множества set](#) может быть немного более подходящим, если важна скорость выполнения, но он будет потреблять еще больше оперативной памяти. Лучшим вариантом может быть использование оптимизированной структуры данных, такой как [trie](#), особенно для статических наборов данных, которые используются, например, для запросов. Для этого уже есть библиотека, а также для многих других древовидных структур данных, некоторые из которых можно найти на <https://github.com/pytries>.

Использование генераторов Python и/или модуль mmap.

Самый простой способ сэкономить оперативную память - вообще не использовать ее. Очевидно, что нельзя полностью избежать использования ОЗУ, но можно избежать загрузки всего набора данных сразу и вместо этого работать с данными постепенно, где это возможно. Самый простой способ добиться этого - использовать [генераторы](#), которые вычисляют элементы по требованию, а не все сразу. Например, [встроенная функция open\(\)](#) имеет поведение генератора, если не использовать [методы файлового объекта](#).

```
from memory_profiler import profile

@profile
def read_file(file):
    with open(file, "r") as file:
        for line in file:
            print(line.strip())

if __name__ == '__main__':
    read_file("some-data.txt")
```

Запускаем:

```
(VirtualEnv) :~$ python3 test.py
...
... # здесь печатаются строки из файла
...
Filename: test.py

Line #      Mem usage      Increment  Occurrences   Line Contents
=====
      3      40.0 MiB       40.0 MiB           1   @profile()
      4                          0.0 MiB           1   def read_file(file):
      5      40.0 MiB        0.0 MiB           1       with open(file, "r") as file:
```


6	40.0 MiB	0.0 MiB	1328	for line in file:
7	40.0 MiB	0.0 MiB	1327	print(line.strip())

О чудо, память использует только декоратор @profile!

Дополнительно можно почитать материал "[Преобразование простой функции в генератор Python](#)".

Более сильным инструментом являются файлы с отображением в памяти, которые позволяют загружать только части данных из файла. Для таких целей стандартная библиотека Python предоставляет [модуль mmap](#). Отображаемые в памяти файлы ведут себя как файлы и байтовые массивы одновременно. Например их можно использовать как с файловыми операциями, такими как чтение, поиск или запись, так и со строковыми операциями:

```
import mmap

with open("some-data.txt", "r") as file:
    with mmap.mmap(file.fileno(), 0, access=mmap.ACCESS_READ) as m:
        print(f"Чтение с использованием метод '.read': {m.read(15)}")
        m.seek(0)
        print(f"Чтение с использованием среза: {m[:15]}")

# Чтение с использованием метод '.read': b'Lorem ipsum dol'
# Чтение с использованием среза: b'Lorem ipsum dol'
```

Загрузка/чтение отображаемого в память файла очень проста. Сначала [открываем файл для чтения](#), как обычно. Затем используем файловый дескриптор файла ([file.fileno\(\)](#)) для создания из него отображаемого в память файла. Оттуда можно получить доступ к его данным как с файловыми операциями, такими как чтение, так и со строковыми операциями, такими как срез.

В большинстве случаев используется только чтение файла, как показано выше, но также можно производить запись:

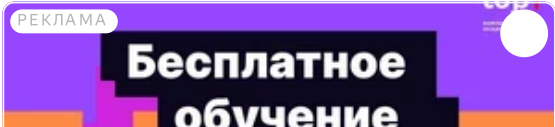
```
import mmap
import re

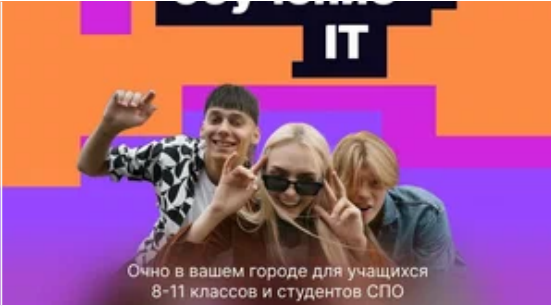
with open("some-data.txt", "r+") as file:
    with mmap.mmap(file.fileno(), 0) as m:
        # Ищем слова, начинающиеся с заглавной буквы
        pattern = re.compile(rb'\b[A-Z].*?\b')
        for match in pattern.findall(m):
            print(match)
            # b'Lorem'
            # b'Morbi'
            # b'Nullam'
            # ...

        # теперь удалим первые 10 символов
        start = 0
        end = 10
        length = end - start
        new_size = len(m) - length
        m.move(start, end, len(m) - end)
        m.flush()
        file.truncate(new_size)
```

Первое отличие в коде, это изменение режима доступа на "r+", что означает как чтение, так и запись. Сначала читаем из файла и используя RegEx ищем все слова, начинающихся с заглавной буквы. После этого демонстрируем удаление данных из файла. Это не так просто, как чтение и поиск, т.к. при удалении части содержимого необходима настройка размера файла в оперативной памяти. Для этого используется метод `move(dest, src, count)`, который копирует размер - конечные байты данных из конца индекса в начало индекса, что в данном случае приводит к удалению первых 10 байтов.

ХОЧУ ПОМОЧЬ
ПРОЕКТУ





code.top-academy.ru

**Бесплатные курсы
программирования
для школьников 8-11кл**

Обучим бесплатно
востребованной IT профессии
за счет государства. Очно.
Красноярск!

Блог

Новости

Вакансии

Узнать больше