

Урок 6. Профилирование памяти

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc88741882)

[ВВЕДЕНИЕ 3](#_Toc88741883)

[АСПЕКТЫ УПРАВЛЕНИЯ ПАМЯТИ 4](#_Toc88741884)

[ПОДРОБНЕЕ ПРО СБОРЩИК МУСОРА (GARBAGE COLLECTOR) 5](#_Toc88741885)

[ОСОБЕННОСТИ ХРАНЕНИЯ ОБЪЕКТОВ PYTHON В ПАМЯТИ 6](#_Toc88741886)

[СБОРКА МУСОРА ВРУЧНУЮ 9](#_Toc88741887)

[ПРОФИЛИРОВАНИЕ ПАМЯТИ 10](#_Toc88741888)

[ПОВЕДЕМ НЕБОЛЬШИЕ ИТОГИ 14](#_Toc88741889)

[СПОСОБЫ МИНИМИЗАЦИИ РАСХОДОВАНИЯ ПАМЯТИ 15](#_Toc88741890)

[Способ 1. Ленивые вычисления 15](#_Toc88741891)

[Способ 2. Слоты в ООП. 17](#_Toc88741892)

[Способ 3. Используйте NumPy 18](#_Toc88741893)

[Способ 4. Используйте кортежи 18](#_Toc88741894)

[Способ 5. Используйте возможности модуля recordclass 18](#_Toc88741895)

[Способ 6. Используйте возможности функции map 19](#_Toc88741896)

[Способ 7. Сериализация 20](#_Toc88741897)

[Способ 8. f-строки 21](#_Toc88741898)

[Способ 9. Используйте возможности функции filter 22](#_Toc88741899)



# ВВЕДЕНИЕ

В предыдущих уроках, а также на курсе основ, мы уже составляли несложные алгоритмы, т.е. определяли набор действий для работы с данными, который приводил нас из исходного состояния к итоговому результат. Но эти алгоритмы были зачастую несложными и цель у нас была сделать их хотя бы семантически правильными, т.е. чтобы код отрабатывал так, как требуется. Мы не думали о качестве наших алгоритмов, а именно о таких их критериях, как скорость выполнения, выделяемая память и лаконичность решения.

На уроке 4 мы уже работали над тем, чтобы научиться профилировать время выполнения алгоритмов и по возможности снижать его. А на данном уроке мы поговорим уже о профилировании памяти и способах уменьшения ее потребления в процессе работы алгоритма. Ну а что касается лаконичности решения, то об этом мы думаем постоянно и стремимся сделать решение более коротким при сохранении эффективности.

Все эти три критерия качества алгоритма, как время выполнения, выделяемая память, лаконичность кода не всегда получается одновременно воплотить в решении задачи, но мы будем работать над этим.

# АСПЕКТЫ УПРАВЛЕНИЯ ПАМЯТИ

Важно понимать, что физически (сам программный код) программа хранится на диске, но выполняется она в ОЗУ. Это ресурс и, хотя его можно увеличить, оперативная память не бесконечна и к ее распределению стоит подходить ответственно. Если раньше мы об этом не думали, то на текущем курсе самое время задуматься, особенно если речь идет о серьезных проектах.

**Итак, память компьютера не бесконечна!**

Приходится постоянно учитывать, что свободная память может закончиться. Эта проблема – наиболее распространенная, но и решается проще остальных: можно купить дополнительные модули и память появится. Но не стоит надеяться только на расширение памяти: все-таки оптимально использовать тот объем, что имеется на данный момент.

Получается, должен быть тот, кто выделяет память и освобождает ее. Освобожденную память необходимо утилизировать – **вернуть системе как свободную**. Но кто этим занимается? Сам разработчик? Вряд ли вы вспомните, что в рамках каких-либо курсов на Python занимались подобными вещами.

В современных языках программирования уже встроены автоматические механизмы распределения и освобождения памяти: new/delete в С++ и JAVA, автоматические средства управления в Python. Т.е. в глобальном понимании именно на интерпретатор возлагается обязанность обеспечивать доступ к памяти, распределять и освобождать ее.

Интерпретатор должен выполнять следующие задачи:

* при объявлении переменной выделять под нее память;
* инициализировать выделенную память некоторым начальным значением;
* предоставлять программисту возможность использовать выделенную память;
* как только память перестает использоваться, освобождать ее;
* обеспечивать возможность последующего повторного использования освобожденной памяти.

Итак, управление памятью – очень важный процесс, обеспечивающий эффективное распределение памяти и очистку памяти от объектов, которые больше не используются. В Python заложены встроенные механизмы управления памятью. Диспетчер памяти интерпретатора обеспечивает выполнение данных задач. В Python управление объектами осуществляется через подсчет ссылок, т.е. диспетчер памяти контролирует число ссылок на объекты программы. При достижении счетчиком ссылок объекта нулевого значения, сборщик мусора **(часть диспетчера памяти, garbage collector)** выполняет освобождение памяти от данного объекта. Таким образом, разработчику не требуется заниматься управлением памятью, процесс ее выделения и освобождения проходит автоматически. Освобожденная память может быть задействована другими объектами.

# ПОДРОБНЕЕ ПРО СБОРЩИК МУСОРА (GARBAGE COLLECTOR)

Мы выяснили, что Python осуществляет удаление объекта, число ссылок на который достигает нулевого значения. Процесс освобождения не используемых блоков памяти называется «сборкой мусора» (garbage collection). Сборщик мусора работает в процессе выполнения программы и активизируется при достижении нуля счетчиком ссылок.

Этот же счетчик увеличивает свое значение в момент присваивания объекту значения, помещения в контейнер (кортеж, список, словарь), присваивания значения объекта другому объекту. Уменьшение же значения счетчика происходит при переназначении ссылки на объект, удалении объекта и т.д.

* Примеры увеличения ссылок:
* Операция присваивания.
* Передача параметров.
* Вставка объекта в контейнер.
* Операции вида a = b.

Память представлена в виде **кучи**, содержащей объекты и другие структуры данных приложения. Под кучей понимается структура данных, хранилище памяти, находящееся в ОЗУ и предусматривающее динамическое выделение памяти. Это не стек, а просто хранилище для переменных вашей программы. Размер кучи определяется при запуске программы и ограничен физически. При завершении работы приложения выделенные блоки памяти освобождаются.

Что же касается языков без сборщика мусора (например, С), то разработчик вынужден выполнять ручное освобождение участков памяти, в которых больше нет необходимости. Иначе существует вероятность появления утечек и фрагментации памяти, что может замедлить работу кучи.

# ОСОБЕННОСТИ ХРАНЕНИЯ ОБЪЕКТОВ PYTHON В ПАМЯТИ

Между переменными в C и Python есть разница. Точнее есть разница в том, как выделяется память под переменные.

В языке C, когда с переменной связывается некоторое значение, можно представить коробочку с именем переменной, в которую помещается это значение.

Получается, что для каждой переменной создается коробочка с именем переменной, хранящая значение. При изменении значения переменной мы помещаем в коробочку новое содержимое.

Присваивание одной переменной другой означает создание копии значения из одной коробочки и помещение этой копии в другую коробочку.

**В Python переменные работают немного иначе!**

Они больше напоминают ярлыки, чем коробочки. Когда в Python вы для переменной определяете значение, то Python как бы прикрепляет ярлык с именем переменной на значение.

**Листинг 1. task\_1.py**

|  |
| --- |
| *"""Простейший пример"""* new\_obj = 5 print(new\_obj)  **del** new\_obj print(new\_obj) *# -> NameError: name 'new\_obj' is not defined* |

Простейший пример уровня основ. В первых двух строках переменная-объект известна. Но если удалить объект и попытаться вновь обратиться, получим ошибку, в которой указано, что переменная не определена. О чем нам говорит этот простейший пример? О том, что сборка мусора в Python полностью автоматическая и программисту не требуется что-либо программировать на низком уровне.

Итак, в Python переменная – ссылка на область памяти, где хранится некоторое значение. При изменении значения переменной в реальности произойдет лишь перемещение ярлыка на другое значение. При этом разработчику не нужно беспокоиться об освобождении памяти, отведенной для хранения значений. Об этом позаботится автоматический сборщик мусора (garbage collector). При обнаружении в памяти значения, на котором нет ни одного ярлыка, он удалит это значение из памяти, тем самым освободив ее.

Присваивание одной переменной другой приводит к тому, что создается еще один ярлык и навешивается на значение, хранящееся в памяти.

Таким образом то, что в других языках именуется переменными, в Python можно назвать именами или ярлыками. Очевидно, что за счет такого подхода в Python память расходуется более эффективно, чем в других языках, где каждая переменная это действительно переменная и под нее выделяется память.

**Листинг 2. task\_2.py**

|  |
| --- |
| *"""Для всех объектов в программе Python ведется подсчет ссылок. Счетчик ссылок на объект увеличивается всякий раз,  когда ссылка на объект записывается в новую переменную или  когда объект помещается в контейнер, такой как список, кортеж или словарь"""* **from** sys **import** getrefcount   *# --------------------------------Task\_1------------------------------------ #* **class** MyClass:  **pass** *# создаем экземпляр класса. увеличиваем счетчик ссылок на 1* mc = MyClass() *# создаем ссылку на объект. увеличиваем счетчик ссылок на 1* temp = mc *# при вызове getrefcount добавляется еще одна временная ссылка* print(getrefcount(mc))  *# --------------------------------Task\_2------------------------------------ #* a = 37 print(getrefcount(37)) b = a print(getrefcount(37)) c = [] c.append(b) print(getrefcount(37))  **""" Во многих случаях количество ссылок оказывается намного больше,  чем можно было бы предположить. Для неизменяемых типах данных,  таких как числа и строки, интерпретатор весьма активно стремится  использовать в разных частях программы один и тот-же объект,  чтобы уменьшить объем потребляемой памяти. """** *# --------------------------------Task\_3------------------------------------ #* **del** a print(getrefcount(37)) b = 42 print(getrefcount(37)) c[0] = 2 print(getrefcount(37)) |

По всем объектам Python-программы ведется подсчет ссылок. Когда ссылка на объект связывается с новой переменной или когда объект помещается в контейнер (например, список, кортеж, словарь), счетчик ссылок на объект увеличивается.

В приведенном примере создается объект, содержащий число 37. Переменная “a” – имя этого объекта. Далее переменная “b” становится еще одним именем того же объекта. Затем переменная “b” помещается в список. Все эти операции приводят к увеличению счетчика на единицу. Хотя на протяжении работы программы существует всего один объект – число 37. Все остальные операции – просто создание новых ссылок на него.

Счетчик ссылок уменьшается при вызове инструкции del. Но как же узнать число ссылок?

Для этого применяется функция getrefcount() модуля sys. Но при ее использовании мы можем получить весьма интересный результат. Число ссылок оказывается больше ожидаемого. Дело в том, что интерпретатор стремится весьма эффективно использовать память в разных частях программы, особенно если это касается неизменяемых типов данных, таких как числа и строки. Они используют в разных частях программы один и тот же объект. Объект везде один, а ссылок много. При использовании getrefcount() ссылка копируется по значению в аргумент функции, что приводит к временному увеличению счетчика ссылок объекта. Поэтому появляется дополнительная ссылка.

Чтобы память, выделяемая для объекта, освободилась, вы должны быть уверены, что число ссылок точно достигло нуля.

# СБОРКА МУСОРА ВРУЧНУЮ

**Листинг 3. task\_3.1.py**

|  |
| --- |
| **def** new\_func():  new\_lst = [1, 2, 3]  new\_lst.append(new\_lst)  **return** new\_lst   print(new\_func()) |

Описанный ранее механизм подсчета ссылок считается достаточно эффективным, но только в том случае, если в программе нет циклических ссылок. А если они есть? Тогда можно сказать о зацикливании ссылок, т.е. один или несколько объектов ссылаются друг на друга. В этом случае счетчик ссылок никогда не достигнет нуля. Так и происходит в приведенном примере. Объект-список ссылается сам на себя, а значит для этого объекта память не будет автоматически освобождена после завершения выполнения функции. При этом стандартный механизм подсчета ссылок нам здесь не поможет. Но проблема решаема, если изменить стандартное поведение сборщика мусора в нашей Python-программе.

Для этого предназначена функция collect() модуля gc (garbage collector). Функция вернет количество собранных и удаленных объектов.

**Листинг 4. task\_3.2.py**

|  |
| --- |
| **import** gc   **def** new\_func():  new\_lst = [1, 2, 3]  new\_lst.append(new\_lst)  **return** new\_lst   print(new\_func())   obj = gc.collect() print(**"Количество скрытых объектов, собранных GC:"**, obj) *# -> Количество скрытых объектов, собранных GC: 1* |

Несмотря на возможность ручной сборки мусора, форсировать этот процесс слишком часто не рекомендуется, поскольку после освобождения памяти сборщик мусора тратит время на анализ пригодности объекта для сборки мусора. При этом тратятся ресурсы и процессорное время. Кроме того, осуществлять ручное управление сборкой мусора рекомендуется проводить только после полного запуска приложения.

# ПРОФИЛИРОВАНИЕ ПАМЯТИ

Для того, чтобы узнать какой объем памяти выделяется в процессе выполнения скрипта, необходимо воспользоваться одним из профилировщиков, например, **memory\_profiler**. Это сторонний модуль, выполняющий построчный анализ потребления памяти для программ на Python.

Установка:

|  |
| --- |
| pip install memory\_profiler  pip install psutil |

Использование пакета **psutil** позволяет существенно ускорить работу модуля memory\_profiler. Данный модуль требует декорирования нужной функции при помощи декоратора @profiler. Декоратор необходимо импортировать.

**Листинг 5. task\_4.py**

|  |
| --- |
| *"""Профилировка затрат памяти"""* **from** copy **import** deepcopy **from** memory\_profiler **import** profile  **"1 Mebibyte = 1048576 Bytes"** *# ссылки, поэтому gc не запускается* @profile **def** function\_1():  *"""Выделяет доп память, не освобождается"""* x = list(range(10000))  y = deepcopy(x)  **return** y   **""" Line # Mem usage Increment Line Contents ================================================  10 35.6 MiB 35.6 MiB @profile  11 def function\_1():  12 '''Выделяет доп память, не освобождается'''  13 36.0 MiB 0.3 MiB x = list(range(10000))  14 36.1 MiB 0.2 MiB y = deepcopy(x)  15 36.1 MiB 0.0 MiB return y """** @profile **def** function\_2():  *"""Выделяет доп память, освобождается"""* x = list(range(100000))  y = deepcopy(x)  **del** x  y = **None  return** y   **""" Line # Mem usage Increment Line Contents ================================================  18 35.9 MiB 35.9 MiB @profile  19 def function\_2():  20 '''Выделяет доп память, освобождается'''  21 39.7 MiB 3.8 MiB x = list(range(100000))  22 40.8 MiB 1.1 MiB y = deepcopy(x)  23 40.8 MiB 0.0 MiB del x  24 36.2 MiB 0.0 MiB y = None  25 36.2 MiB 0.0 MiB return y """  if** \_\_name\_\_ == **"\_\_main\_\_"**:  function\_1()  function\_2() |

Расшифровка результатов:

Первый столбец – **Line**. Это номер строки профилируемого кода.

Второй столбец – **Mem usage**. Это использование памяти интерпретатором после выполнения строки.

Третий столбец – **Increment**. Представляет разницу по памяти текущей строки относительно предыдущей.

Четвертый столбец – **Line Contents**. Содержит строки профилируемого кода.

**Внимание!**

Важно помнить, что если сборщик мусора отказывается уничтожить объект, то причина этого – наличие ссылки на этот объект. Удалите все ссылки и память будет очищена. Но на эту меру нужно идти, когда в вашем скрипте существуют ссылки на объекты, занимающие значительный объем памяти. При это объекты эти вам больше не нужны, а скрипт еще не закончил выполнения. Это вполне подходящий момент для того, чтобы освободить память.

В других случаях явно удалять ссылки для освобождения памяти не требуется, поскольку после завершения выполнения скрипта память и так будет освобождена.

Еще примеры с профилировкой:

**Листинг 6. task\_5.py**

|  |
| --- |
| *"""Профилировка памяти"""* **from** memory\_profiler **import** profile   **class** Point:  **def** \_\_init\_\_(self, x=0, y=0, lst=[]):  self.x = x  self.y = y  self.lst = list(range(100000))   **def** \_\_del\_\_(self):  class\_name = self.\_\_class\_\_.\_\_name\_\_  print(**f'{**class\_name**} уничтожен'**)   @profile **def** func():  pt1 = Point()  pt2 = pt1  pt3 = pt1  print(id(pt1), id(pt2), id(pt3))  **del** pt1  **del** pt2  **del** pt3   func() |

**Листинг 7. task\_6.py**

|  |
| --- |
| *"""Профилировка затрат памяти"""* **from** math **import** sqrt **from** memory\_profiler **import** profile   @profile **def** get\_prime\_numbers(count):  prime\_numbers = [2]  next\_number = 3   **while** len(prime\_numbers) < count:  **if** is\_prime(next\_number, prime\_numbers):  prime\_numbers.append(next\_number)  next\_number += 1   **return** prime\_numbers   @profile **def** is\_prime(num, prime\_numbers):  limit = int(sqrt(num)) + 1  **for** i **in** prime\_numbers:  **if** i > limit:  **break  if** num % i == 0:  **return False  return True** get\_prime\_numbers(5)   *# Проблем с памятью нет. Всё в пределах нормы.* |

**Листинг 8. task\_7.py**

|  |
| --- |
| *"""Профилировка затрат памяти"""* **from** copy **import** deepcopy **from** memory\_profiler **import** profile   @profile **def** function\_1():  *"""Значительный инкремент"""* x = list(range(100000))  y = deepcopy(x)  **return** y   function\_1() |

# ПОВЕДЕМ НЕБОЛЬШИЕ ИТОГИ

* До этого вы писали небольшие программы и об эффективности использования памяти не задумывались. Если же вы занимаетесь каким-то серьезным проектом, то этот вопрос имеет актуальность. Об этом вопросе стоит подумать уже с самого начала работы над проектом.
* За управление памятью в Python отвечает интерпретатор, поэтому разработчик не занимается управлением памятью на низком уровне.
* Управление памятью в Python включает в себя приватную кучу, состоящую из всех Python-объектов и структур данных.
* Менеджер памяти в Python предоставляет доступ к приватной куче. В процессе создания объекта виртуальная машина Python выделяет необходимый объем памяти и определяет куда в памяти поместить этот объект.

Получается вся низкоуровневая работа уже предопределена, но может ли что-то сделать разработчик? Он может влиять на использование памяти на высоком уровне, т.е. применять такие приемы программирования, алгоритмы, подходы, средства, которые позволяют минимизировать затраты памяти в программе?

# СПОСОБЫ МИНИМИЗАЦИИ РАСХОДОВАНИЯ ПАМЯТИ

## Способ 1. Ленивые вычисления

В фундаментальном понимании, это стратегия вычислений, при которой вычисления следует откладывать до тех пор, пока не понадобятся их результаты. В Python эта стратегия реализована через функции-генераторы и оператор yield.

Старайтесь использовать генераторы для вычислений. Суть таких вычислений заключается в итерации: или через явное использование «for», или неявное. For передает это вычисление любой функции или конструкции, осуществляющей итерацию. Генераторы не возвращают любое количество элементов сразу вместе, как списки, они возвращают элементы один за другим.

Сравните результаты работы кода в этих трех листингах и сравните результаты.

**Листинг 9. task\_8\_1.py**

|  |
| --- |
| *"""Профилирование времени и памяти"""* **from** memory\_profiler **import** memory\_usage   **def** decor(func):  **def** wrapper(\*args, \*\*kwargs):  m1 = memory\_usage()  res = func(args[0])  m2 = memory\_usage()  mem\_diff = m2[0] - m1[0]  **return** res, mem\_diff  **return** wrapper   @decor **def** check\_even\_1(numbers):  even = []  **for** num **in** numbers:  **if** num % 2 == 0:   even.append(num\*num)  **return** even   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   res, mem\_diff = check\_even\_1(list(range(100000)))  print(**f"Выполнение заняло {**mem\_diff**} Mib"**)  *# Выполнение заняло 2.2890625 Mib* |

**Листинг 10. task\_8\_2.py**

|  |
| --- |
| *"""Профилирование времени и памяти"""* **from** memory\_profiler **import** memory\_usage   **def** decor(func):  **def** wrapper(\*args, \*\*kwargs):  m1 = memory\_usage()  res = func(args[0])  m2 = memory\_usage()  mem\_diff = m2[0] - m1[0]  **return** res, mem\_diff  **return** wrapper   @decor **def** check\_even\_2(numbers):  *# списковое включение* my\_list = [num \* num **for** num **in** numbers **if** num % 2 == 0]  **return** my\_list   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   res, mem\_diff = check\_even\_2(list(range(100000)))  print(**f"Выполнение заняло {**mem\_diff**} Mib"**)  *# Выполнение заняло 2.328125 Mib* |

**Листинг 11. task\_8\_3.py**

|  |
| --- |
| **from** memory\_profiler **import** memory\_usage   **def** decor(func):  **def** wrapper(\*args, \*\*kwargs):  m1 = memory\_usage()  res = func(args[0])  m2 = memory\_usage()  mem\_diff = m2[0] - m1[0]  **return** res, mem\_diff  **return** wrapper   @decor **def** check\_even\_3(numbers):  **for** num **in** numbers:  **if** num % 2 == 0:  **yield** num \* num   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   my\_generator, mem\_diff = check\_even\_3(list(range(100000)))  print(type(my\_generator))  **for** i **in** my\_generator:  print(i)   print(**f"Выполнение заняло {**mem\_diff**} Mib"**)  *# Выполнение заняло 0.01171875 Mib* |

## Способ 2. Слоты в ООП.

Речь идет о конструкции \_\_slots\_\_ при определении классов в Python. При этом мы как бы «говорим» Python не использовать динамический словарь для хранения атрибутов и их значений. Вы ведь помните, что словарь в Python – хеш-таблица, под которую всегда выделяется больше памяти, чем это нужно. Использование слотов позволяет сохранить атрибуты в менее затратном по памяти контейнере – списке, кортеже. При этом список атрибутов строго ограничен. Вы не можете добавить новые атрибуты динамически. Для кого-то это минус, для кого-то – всего лишь особенность использования слотов.

**Листинг 12. task\_9.py**

|  |
| --- |
| *"""Обычный класс и класс со слотами"""* **""" Функция sys.getsizeof возвращает размер переданного ей обьекта,  этот размер не включает в себя сложные структуры классов и т.д.  Функция pympler.asizeof - рекурсивно ищет все вложенные  поля и элементы, и отображает общий размер обьекта """  from** pympler **import** asizeof   **class** BasicClass:  **def** \_\_init\_\_(self, param\_x, param\_y):  self.param\_x = param\_x  self.param\_y = param\_y   BC\_OBJ = BasicClass(5, 6) print(asizeof.asizeof((BC\_OBJ))) *# -> 344* **class** BasicClass:  \_\_slots\_\_ = [**'param\_x'**, **'param\_y'**]   **def** \_\_init\_\_(self, param\_x, param\_y):  self.param\_x = param\_x  self.param\_y = param\_y   BC\_OBJ = BasicClass(5, 6) print(asizeof.asizeof(BC\_OBJ)) *# -> 120* |

## Способ 3. Используйте NumPy

Эта библиотека отлично подходит для обработки большого объема данных, поскольку она очень эффективно управляет ресурсами памяти.

**Листинг 13. task\_10.py**

|  |
| --- |
| *"""Используем NumPy"""* **from** random **import** randint **from** pympler **import** asizeof **from** numpy **import** array  lst\_obj = [randint(-100, 100) **for** \_ **in** range(50000)] print(asizeof.asizeof(lst\_obj)) *# -> 1165720 байт* lst\_obj = [randint(0, 100) **for** \_ **in** range(50000)] print(asizeof.asizeof(lst\_obj)) *# -> 409720 байт* lst\_obj = array([randint(-100, 100) **for** \_ **in** range(50000)]) print(type(lst\_obj)) print(asizeof.asizeof(lst\_obj)) *# -> 200096 байт* lst\_obj = array([randint(0, 100) **for** \_ **in** range(50000)]) print(asizeof.asizeof(lst\_obj)) *# -> 200096 байт* |

## Способ 4. Используйте кортежи

Это коллекции, не являющиеся хеш-таблицами и не требующие выделения дополнительной памяти, поэтому они более экономичны.

**Листинг 14. task\_11.py**

|  |
| --- |
| **from** sys **import** getsizeof **from** pympler.asizeof **import** asizeof   d = {1: **'1'**, 2: **'2'**, 3: **'3'**} print(getsizeof(d)) *# -> 240* print(asizeof(d)) *# -> 504* t = (1, 2, 3) print(getsizeof(t)) *# -> 72* print(asizeof(t)) *# -> 168* |

## Способ 5. Используйте возможности модуля recordclass

Модуль recordclass применяется для создания переменных типа recordclass, который похож на namedtuple за одним исключением: он изменяемый. При этом переменные занимают либо одинаковое количество памяти, либо переменные recordclass экономнее в её потреблении. То есть, изменяемость типа не увеличивает потребление памяти в данном случае. Модуль сторонний и требует установки командой: pip install recordclass.

**Листинг 15. task\_12.py**

|  |
| --- |
| **import** sys  **from** collections **import** namedtuple **from** recordclass **import** recordclass   **def** rc\_demo():  var\_1 = recordclass(**'var\_1'**, (**'x'**, **'y'**, **'z'**))  var\_2 = namedtuple(**'var\_2'**, (**'x'**, **'y'**, **'z'**))  c = {**'x'**: 1, **'y'**: 2, **'z'**: 3}  a = var\_1(x=1, y=2, z=3)  b = var\_2(x=1, y=2, z=3)  a.x = 4   print(**f'Объём занимаемой объектом recordclass памяти: '  f'{**sys.getsizeof(a)**} байт(а)'**)  print(**f'Объём занимаемой объектом namedtuple памяти: '  f'{**sys.getsizeof(b)**} байт(а)'**)  print(**f'Объём занимаемой объектом dict памяти:'  f' {**sys.getsizeof(c)**} байт(а)'**)   rc\_demo()  **""" Объём занимаемой объектом recordclass памяти: 48 байт(а) Объём занимаемой объектом namedtuple памяти: 72 байт(а) Объём занимаемой объектом dict памяти: 240 байт(а) """** |

## Способ 6. Используйте возможности функции map

Для уменьшения расхода памяти, помогает встроенная функция map. Значительная экономия памяти происходит с увеличением количества данных.

Сравните варианты без функции map и с ее применением.

**Листинг 16. task\_13\_1.py**

|  |
| --- |
| *"""Профилирование времени и памяти"""* **from** memory\_profiler **import** memory\_usage   **def** decor(func):  **def** wrapper(\*args, \*\*kwargs):  m1 = memory\_usage()  res = func(args[0])  m2 = memory\_usage()  mem\_diff = m2[0] - m1[0]  **return** res, mem\_diff  **return** wrapper   @decor **def** check\_even\_1(lst):  *# списковое включение* new\_list = [str(i) **for** i **in** lst]  **return** new\_list   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   res, mem\_diff = check\_even\_1(list(range(100000)))  print(**f"Выполнение заняло {**mem\_diff**} Mib"**)  *# Выполнение заняло 8.33984375 Mib* |

**Листинг 17. task\_13\_2.py**

|  |
| --- |
| **from** memory\_profiler **import** memory\_usage   **def** decor(func):  **def** wrapper(\*args, \*\*kwargs):  m1 = memory\_usage()  res = func(args[0])  m2 = memory\_usage()  mem\_diff = m2[0] - m1[0]  **return** res, mem\_diff  **return** wrapper   @decor **def** check\_even\_2(lst):  new\_list = map(str, lst)  print(type(new\_list))  print(list(new\_list))  **return** new\_list   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   res, mem\_diff = check\_even\_2(list(range(100000)))  print(**f"Выполнение заняло {**mem\_diff**} Mib"**)  *# Выполнение заняло 3.08984375 Mib* |

## Способ 7. Сериализация

Если существует острая необходимость использовать словари для хранения объектов, можно выполнять сериализацию их в формат json-строк.

**Листинг 17. task\_14.py**

|  |
| --- |
| **from** json **import** loads, dumps **from** pympler **import** asizeof  gen\_dict = {i: i \* 2 **for** i **in** range(100000)} dumped\_dict = dumps(gen\_dict) print(type(dumped\_dict))  out\_dict = loads(dumped\_dict) print(type(out\_dict))  print(**'Размер dict: '**, asizeof.asizeof(gen\_dict)) print(**'Размер json: '**, asizeof.asizeof(dumped\_dict))  **""" <class 'dict'> <class 'str'> Размер dict: 11638840 Размер json: 1633384 """** |

## Способ 8. f-строки

Для форматирования строк лучшим вариантов являются f-строки. Мы уже доказали, что они обрабатываются быстрее, чем другие способы форматирования (конкатенация, функция format и т.д.). Но оказывается в плане использовании памяти они также лучшие. И конечно не забудем про такие их достоинства, как читаемость и поддерживаемость.

**Листинг 18. task\_15.py**

|  |
| --- |
| *""" Выигрыш в использовании памяти дает использование форматированной строки вместо конкатенации, так как не происходит аллокация памяти  (выделение блока памяти в куче) под временные строки, когда поочередно конкатенируем строки. """* **from** memory\_profiler **import** profile   @profile **def** format\_str(a, b, c, d, e):  res = **f'{**a**}\_{**b**}\_{**c**}\_{**d**}\_{**e**}\_{**a**}\_{**b**}\_{**c**}\_{**d**}\_{**a**}\_{**b**}'  return** res   @profile **def** concat\_str(a, b, c, d, e):  res = (a + **'\_'** + b + **'\_'** + c + **'\_'** + d + **'\_'** + e + **'\_'** + a + **'\_'** + b + **'\_'** + c + **'\_'** + d + **'\_'** + a + **'\_'** + b)  **return** res   concat\_str(**'one'** \* (10 \*\* 4), **'two'** \* (10 \*\* 3),  **'three'** \* (10 \*\* 3), **'four'** \* (10 \*\* 3), **'five'** \* (10 \*\* 3)) format\_str(**'one'** \* (10 \*\* 4), **'two'** \* (10 \*\* 3),  **'three'** \* (10 \*\* 3), **'four'** \* (10 \*\* 3), **'five'** \* (10 \*\* 3))  **""" Line # Mem usage Increment Line Contents ================================================  11 35.7 MiB 35.7 MiB @profile  12 def concat\_str(a, b, c, d, e):  13 35.8 MiB 0.2 MiB res = (a + '\_' + b + '\_' + c + '\_'   + d + '\_' + e + '\_' + a + '\_' + b + '\_' + c + '\_' + d + '\_' + a + '\_' + b)  14 35.8 MiB 0.0 MiB return res    Line # Mem usage Increment Line Contents ================================================  5 35.8 MiB 35.8 MiB @profile  6 def format\_str(a, b, c, d, e):  7 35.8 MiB 0.0 MiB res = f'{a}\_{b}\_{c}\_{d}\_{e}  \_{a}\_{b}\_{c}\_{d}\_{a}\_{b}'  8 35.8 MiB 0.0 MiB return res """** |

## Способ 9. Используйте возможности функции filter

Помогает значительно оптимизировать память, использование, наравне с map.

**Листинг 19. task\_16.py**

|  |
| --- |
| **from** memory\_profiler **import** memory\_usage   **def** memory(func):  **def** wrapper(\*args, \*\*kwargs):  m1 = memory\_usage()  res = func(\*args)  m2 = memory\_usage()  mem\_diff = m2[0] - m1[0]  print(**f"Выполнение заняло {**mem\_diff**} Mib"**)  **return** res   **return** wrapper   @memory **def** check\_even\_1(lst):  new\_list = [i **for** i **in** lst **if** i % 2 == 0]  **return** new\_list   @memory **def** check\_even\_2(lst):  new\_list = filter(**lambda** x: x % 2 == 0, lst)  **return** new\_list   check\_even\_1(list(range(5000000))) check\_even\_2(list(range(5000000)))  **""" Выполнение заняло 19.453125 Mib Выполнение заняло 0.00390625 Mib """** |