# Введение

В предыдущих уроках мы изучали алгоритмы работы с различными данными. Теперь пришло время посмотреть, а как они действуют в памяти системы. Как размещаются и хранятся данные в памяти? Как управлять памятью для более эффективной работы программы?

# Управление памятью с точки зрения разработчика компилятора

## Основные фазы работы с памятью

Один из самых важных ресурсов компьютера – это память. Сейчас именно на интерпретатор языка программирования 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 (возможно, предварительно очистив);
* обеспечивать возможность последующего повторного использования освобожденной памяти.

## Проблемы управления памятью

Перечислим основные проблемы работы с памятью.

1. **Память компьютера не бесконечна.**

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

1. **Механизмы управления памятью (автоматические и ручные) в зависимости от языка программирования.**

В современных языках программирования уже встроены автоматические механизмы распределения и освобождения памяти: new/delete в С++ и JAVA, автоматические средства управления в Python.

1. **Работа с памятью, которая является общей для нескольких процессов.**

Не забываем: в многопоточных приложениях несколько процессов работают с общими ресурсами. Всегда держите в уме синхронизацию потоков, иначе возникнут проблемы с общей областью памяти.

Память становится свободной, как только выполняется оператор явного освобождения памяти (free/delete). Другой варианта – в момент окончания времени жизни последней переменной, использующей данную область памяти. Эти операции, делающие структуру данных логически недоступной, называются освобождением памяти. Но для компилятора в этот момент работа только начинается.

Освобожденную память необходимо утилизировать – **вернуть системе как свободную**. Сами операции уничтожения памяти и утилизации могут быть разнесены по времени.

Управление памятью в Python

Управление памятью — это неотъемлемая часть работы компьютерных программ. Python решает практически все задачи по управлению памятью незаметно для программиста. Python позволяет тому, кто пишет на этом языке, абстрагироваться от множества мелких деталей, касающихся работы с компьютерами. Это даёт программисту возможность работать на более высоком уровне, создавать свой код, не заботясь о том, где хранятся его данные.

Управление памятью Python разделено на две части.

* + Память стека
  + Память кучи

Не пугайтесь этих понятий, ведь в современном мире программирования нам все реже приходиться заниматься низкоуровневыми вещами и эти термины могут быть незнакомыми. Для начала поговорим об этих понятиях в фундаментальном контексте.

**Стек**

1. Область оперативной памяти, создаваемая для каждого потока.
2. Последний добавленный в стек блок памяти становится первым на вывод из стека.
3. Когда некоторая функция объявляет новую переменную, она помещается в стек, а при выходе переменной из зоны видимости (например, функция заканчивается), автоматом удаляется из стека.
4. Если стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

**Куча**

1. Область в оперативной памяти, допускающая динамическое выделение памяти и играющая роль хранилища для переменных.
2. В этом случае к переменной можно обратиться не только в потоке, но и на уровне всего приложения. Именно так определяются глобальные переменные.
3. При завершении работы приложения выделенные участки памяти освобождаются.
4. Размер кучи определяется при запуске приложения и ограничен лишь физически, что позволяет создавать динамические переменные.

Что же с Python?

Управление памятью в Python включает в себя **частную кучу**, содержащую все объекты и структуры данных Python. Важно понимать, что управление кучей Python выполняется самим интерпретатором и что у пользователя нет контролировать его, даже если она регулярно манипулирует указателями объектов на блоки памяти внутри этой кучи. Управление памятью Python разделено на две части.

* + Память стека
  + Память кучи
* Функции (методы) и переменные создаются в памяти стека.
* Значения переменных и экземпляров объектов создаются в памяти кучи.
* В стеке памяти кадр стека создается всякий раз, когда создаются методы и переменные.
* Эти блоки стеков автоматически уничтожаются при завершении работы функций/методов.
* Python имеет механизм сбора мусора, как только переменные больше не используются и функции завершены, сборщик мусора очищает мертвые объекты.

В программировании сборка мусора ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) garbage collection) — одна из форм автоматического управления [памятью](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C). Специальный процесс, называемый сборщиком мусора ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) garbage collector), периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложениями.

Итог: Python - как С#, Java, Perl, Ruby, Lua и многие другие языки — использует сборку мусора, а не ручное управление памятью. Вы просто произвольно создаете объекты и диспетчер памяти языков периодически (или когда вы специально направляете его) ищет любые объекты, которые больше не ссылаются на вашу программу.

**О том, что же на самом деле происходит, когда вы присваиваете значения переменным в Python и как он всем этим управляет.**

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.

И теперь при изменении значения переменной в реальности произойдет лишь перемещение ярлыка на другое значение. При это 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.

**Листинг 1. task\_1.py**

|  |
| --- |
| *"""Определяем количество ссылок на объект"""* **"""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"""  import** sys  *##############################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(sys.getrefcount(mc))  *##############################Task\_2##################################* a = 37 print(sys.getrefcount(37)) b = a print(sys.getrefcount(37)) c = [] c.append(b) print(sys.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(sys.getrefcount(37)) b = 42 print(sys.getrefcount(37)) c[0] = 2 print(sys.getrefcount(37))  a = 5 print(sys.getrefcount(5)) b = a print(sys.getrefcount(5)) **del** a *# print(a)* print(b) b = 5 print(b) print(sys.getrefcount(5)) |

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 или когда поток выполнения покидает область видимости ссылки (или при присваивании нового значения). Но как же узнать число ссылок?

Для этого применяется функция sys.getrefcount(). 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.

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

Свою долю в путаницу с числом ссылок вносит и функция getrefcount(). При ее использовании ссылка копируется по значению в аргумент функции, что приводит к временному увеличению счетчика ссылок объекта. Поэтому появляется дополнительная ссылка.

Определение расходования памяти

Для того, чтобы узнать какой объем памяти использует ваш Python-код, необходимо воспользоваться одним из инструментов, на пример, **memory\_profiler** (Профилировщик памяти). Это Python-модуль, отвечающий за выполнение мониторинга потребления памяти процессором, а также 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 1. task\_1.py**

|  |
| --- |
| *"""Профилировка затрат памяти"""* **import** copy **from** memory\_profiler **import** profile **from** sys **import** getrefcount   @profile **def** function\_1():  *"""Выделяет доп память, не освобождается"""* x = list(range(100000))  y = copy.deepcopy(x)  **return** y   @profile **def** function\_2():  *"""Выделяет доп память, освобождается"""* x = list(range(100000))  print(getrefcount(x))  y = copy.deepcopy(x)  print(getrefcount(y))  **del** x  y = **None  return** y   **if** \_\_name\_\_ == **"\_\_main\_\_"**:  function\_1()  function\_2()  “””  Line # Mem usage Increment Line Contents  ================================================  8 47.6 MiB 47.6 MiB @profile  9 def function\_1():  10 """Выделяет доп память, не освобождается"""  11 51.5 MiB 3.9 MiB x = list(range(100000))  12 52.4 MiB 0.9 MiB y = copy.deepcopy(x)  13 52.4 MiB 0.0 MiB return y  “”” |

Расшифровка результатов:

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

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

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

Четвертый столбец – **Line Contents**. Содержит профилируемый код.

**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. Для сборщика мусора наличие ссылок – сигнал «Объект нужен, используется». Зачем тогда освобождать память?

Еще примеры с профилировкой:

**Листинг 3. task\_3.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 4. task\_4.py**

|  |
| --- |
| *"""Профилировка затрат памяти"""* **import** math **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(math.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 5. task\_5.py**

|  |
| --- |
| *"""Профилировка затрат памяти"""* **from** copy **import** deepcopy **from** memory\_profiler **import** profile   @profile **def** function\_1():  *"""Значительный инкремент"""* x = list(range(10000000))  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 различными низкоуровневыми вещами.
* 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? Есть ли способы работы с памятью, алгоритмы, подходы, средства, которые позволяют минимизировать затраты памяти в программе?

Способы минимизации расходования памяти

**Способ 1. Ленивые вычисления.** В фундаментальном понимании, это стратегия вычисления, по которой вычисления следует откладывать до тех пор, пока не понадобятся их результат. В Python эта стратегия реализована через функции-генераторы и ключевое слово yield.

Старайтесь использовать генераторы для вычислений. Суть таких вычислений заключается в итерации: или через явное использование «for», или неявное. For передает это вычисление любой ф-ции 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 6\_1. task\_6\_1.py**

|  |
| --- |
| *"""Профилировка времени и памяти"""* **""" Генераторам же предшествуют итераторы.  Когда вы создаёте список, вы можете считывать его  элементы один за другим — это называется итерацией """** *# Выполнение заняло 15.625 сек and 703.0546875 Мб* **import** memory\_profiler **import** time   **def** check\_even\_1(numbers):  even = []  **for** num **in** numbers:  **if** num % 2 == 0:   even.append(num\*num)   **return** even   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   *# левые отсечки времени и памяти* t1 = time.process\_time()  m1 = memory\_profiler.memory\_usage()   cubes = check\_even\_1(range(100000000))   *# правые отсечки времени и памяти* t2 = time.process\_time()  m2 = memory\_profiler.memory\_usage()   time\_diff = t2 - t1  mem\_diff = m2[0] - m1[0]   print(**f"Выполнение заняло {**time\_diff**} сек and {**mem\_diff**} Mib"**) |

**Листинг 6\_2. task\_6\_2.py**

|  |
| --- |
| *"""Профилировка времени и памяти"""* **""" Генераторам же предшествуют итераторы.  Когда вы создаёте список, вы можете считывать его  элементы один за другим — это называется итерацией """** *# Выполнение заняло 9.5625 сек and 1932.86328125 Мб* **import** memory\_profiler **import** time   **def** check\_even(numbers):  *# списковое включение* mylist = [num \* num **for** num **in** numbers **if** num % 2 == 0]  **return** mylist  *#print(check\_even([1, 2, 3, 4, 5]))* **if** \_\_name\_\_ == **'\_\_main\_\_'**:   t1 = time.process\_time()  m1 = memory\_profiler.memory\_usage()   cubes = check\_even(range(100000000))   t2 = time.process\_time()  m2 = memory\_profiler.memory\_usage()   time\_diff = t2 - t1  mem\_diff = m2[0] - m1[0]  print(**f"Выполнение заняло {**time\_diff**} сек and {**mem\_diff**} Мб"**) |

**Листинг 6\_3. task\_6\_3.py**

|  |
| --- |
| *"""Профилировка времени и памяти"""* **""" Генераторы это тоже итерируемые объекты, но прочитать их можно лишь один раз.  Это связано с тем, что они не хранят значения в памяти, а генерируют их на лету. """  """ Всё то же самое, разве что используются круглые скобки вместо квадратных. НО: нельзя применить конструкцию for i in mygenerator второй раз,  так как генератор может быть использован только единожды:  он вычисляет 0, потом забывает про него и вычисляет 1,  завершаяя вычислением 4 — одно за другим. """** *# Выполнение заняло 0.0 сек and 0.01171875 Мб* **import** memory\_profiler **import** time   **def** check\_even(numbers):  **for** num **in** numbers:  **if** num % 2 == 0:  **yield** num \* num  print(check\_even([1, 2, 3, 4, 5])) mygenerator = check\_even([1, 2, 3, 4, 5]) **for** i **in** mygenerator:  print(i)  **if** \_\_name\_\_ == **'\_\_main\_\_'**:   t1 = time.process\_time()  m1 = memory\_profiler.memory\_usage()   mygenerator = check\_even([1, 2, 3, 4, 5])  **for** i **in** mygenerator:  print(i)   t2 = time.process\_time()  m2 = memory\_profiler.memory\_usage()   time\_diff = t2 - t1  mem\_diff = m2[0] - m1[0]  print(**f"Выполнение заняло {**time\_diff**} сек and {**mem\_diff**} Мб"**)  **""" Генераторы дают возможность производить «ленивые» вычисления.  Суть использования такого вычисления заключается в итерации:  как с помощью явного использования “for”, так и неявного.  For просто передает это вычисление любой функции или конструкции,  которая и осуществляет итерацию. Вы можете подумать, что генераторы возвращают любое количество  элементов наподобие списка, однако вместо того,  чтобы возвращать их все сразу они возвращаются один за другим.  Функция генератора останавливается до тех пор,  пока следующий элемент не будет запрошен. """** |

**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 при это строго ограничен. Вы не можете добавить новые атрибуты динамически. Для кого-то это минус, для кого-то – всего лишь особенность использования слотов.

**Листинг 7. task\_7.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 """** *# numpy* **from** pympler **import** asizeof **from** sys **import** getsizeof   **class** BasicClass:  *"""  В обычной ситуации в Python в объекты можно добавлять  новые атрибуты вне описания класса  """* **def** \_\_init\_\_(self, param\_x, param\_y):  self.param\_x = param\_x  self.param\_y = param\_y   BC\_OBJ = BasicClass(5, 6) print(BC\_OBJ.\_\_dict\_\_) print(asizeof.asizeof((BC\_OBJ))) BC\_OBJ.param\_z = 7 print(BC\_OBJ.\_\_dict\_\_) print(asizeof.asizeof((BC\_OBJ)))    **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(BC\_OBJ.\_\_slots\_\_) print(asizeof.asizeof(BC\_OBJ)) BC\_OBJ.param\_z = 7 print(BC\_OBJ.\_\_slots\_\_) |

**Способ 3. Используйте NumPy**

Эта библиотека отлично подходит для обработки большого объема чисел и (или) данных, поскольку она очень эффективно управляет ресурсами памяти.

**Листинг 8. task\_8.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(asizeof.asizeof(lst\_obj)) *# -> 200096 байт* lst\_obj = array([randint(0, 100) **for** \_ **in** range(50000)]) print(asizeof.asizeof(lst\_obj)) *# -> 200096 байт* |

Для профилирования памяти, выделяемой под отдельные объекты, вы можете воспользоваться еще одним интересным инструментом.

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Профилировка затрат памяти""" # https://pypi.org/project/guppy3/* **from** guppy **import** hpy **from** copy **import** deepcopy  h = hpy()  x = list(range(100000)) y = deepcopy(x)  print(h.heap()) |