Низкие уровни Питона

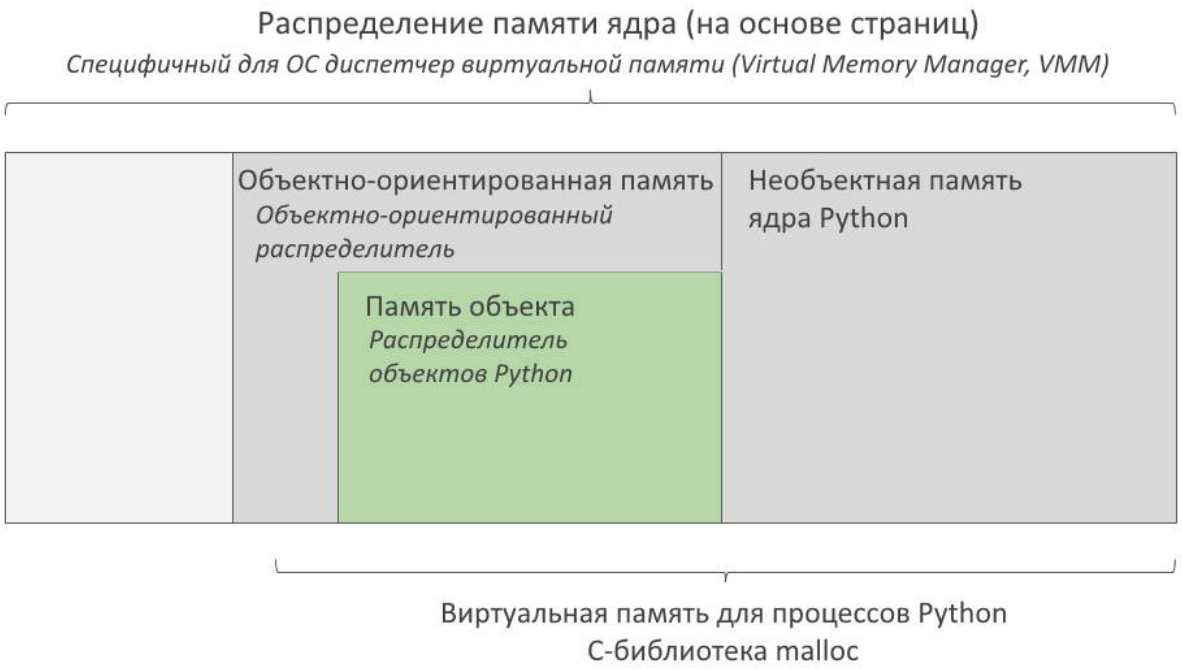
Память

При запуске Python-программы создается новый процесс, в рамках которого операционная система выделяет пул ресурсов, включая виртуальное адресное пространство. В эту память загружается интерпретатор Python вместе со всеми необходимыми ему для работы данными, включая код вашей программы.

Оговоримся, что CPython не взаимодействует напрямую с регистрами и ячейками физической памяти — только с ее **виртуальным представлением**. В начале выполнения программы операционная система создает новый процесс и выделяет под него ресурсы. Выделенную виртуальную память интерпретатор использует для 1) собственной корректной работы, 2) стека вызываемых функций и их аргументов и 3) хранилища данных, представленного в виде кучи.

Оставшаяся свободная виртуальная память может использоваться для хранения информации об объектах Python’а. Для управления этой памятью в CPython используется специальный механизм, который называется аллокатор. Он используется каждый раз, когда вам нужно создать новый объект.

Поверх него работают аллокаторы, реализующие стратегии управления памятью, специфичные для отдельных типов объектов. Объекты разных типов — например, числа и строки — занимают разный объем, к ним применяются разные механизмы хранения и освобождения памяти.

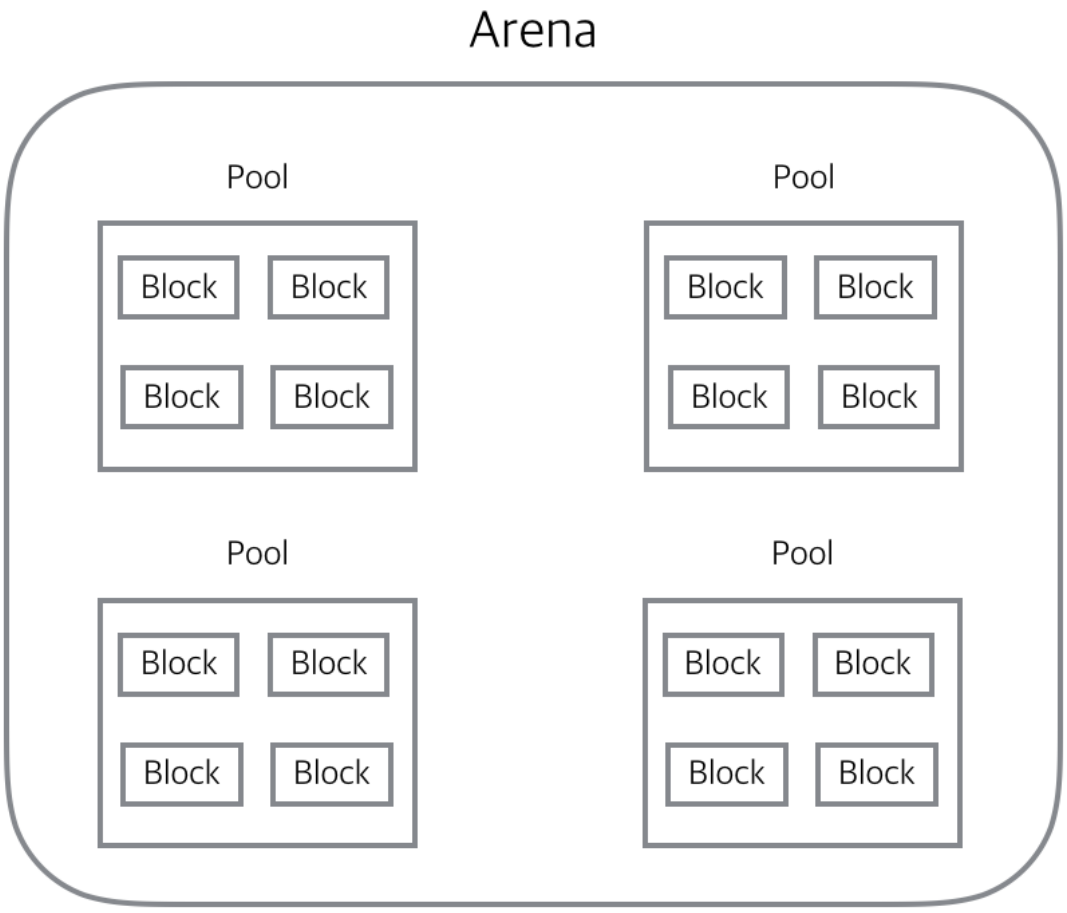


Системный вызов на выделение памяти — штука трудозатратная, зачастую связанная с переходом в контекст ядра операционной системы и т.п. Поэтому одна из главных задач аллокатора Python — оптимизация количества системных вызовов.

Для больших объектов (больше 512 байт) Python выделяет память напрямую у ОС. Обычно таких объектов не очень много в рамках программы, и создаются они нечасто. Поэтому накладные расходы на создание таких объектов напрямую в RAM не так высоки.

Как же устроен аллокатор внутри? Он состоит из частей трёх видов:

- **Арена** — большой непрерывный кусок памяти (обычно 256 килобайт), содержит несколько страниц виртуальной памяти операционной системы.
- **Пул** — одна страница виртуальной памяти (обычно 4 килобайта).
- **Блок —** маленький кусочек памяти, используемый для хранения одного объекта.



Когда Python’у нужно расположить какой-то объект в оперативной памяти он ищет подходящую арену. Если такой нету, он запрашивает новую арену у операционной системы. Арены организованы в двусвязный список и отсортированы от самой заполненной к самой свободной. Для добавления нового объекта выбирается САМАЯ ЗАПОЛНЕННАЯ арена. Арены — единственное место, в котором происходит запрос и освобождение памяти в Python. Если какая-то арена полностью освобождается от объектов, она возвращается назад операционной системе и память освобождается. Таким образом, чем компактнее живут объекты в рамках арен, тем ниже общее потребление оперативной памяти программой.

Внутри арен расположены пулы. Каждый пул предназначен для хранения блоков одинакового размера (то есть в одном пуле могут быть только блоки одного и тоже размера, при этом в арене могут быть пулы с разными размерами блоков).

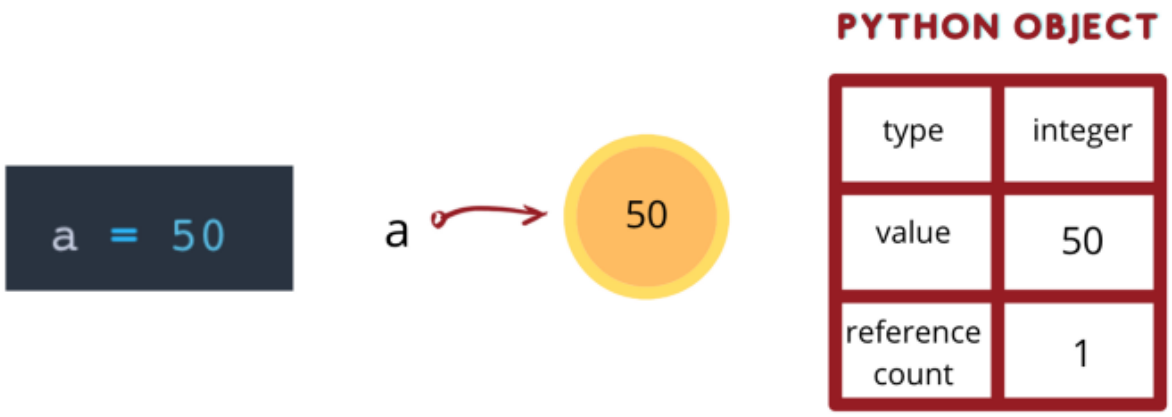
Каждый пул может быть в одном из трех состояний — used, full и empty.

Внутри пула живут блоки. Каждый блок предназначен для хранения одного объекта. В целях унификации размеры блоков фиксированы. Они могут быть размером 8, 16, 24, 32 …. 512 байт. Если Вам нужно 44 байта для хранения объекта, то он будет расположен в блоке на 48 байт. Когда вы хотите разместить новый объект, берется первый свободный блок, и объект располагается в нем. Когда объект удаляется, его блок помещается в список свободных.

Время жизни объекта и при чем тут GIL

Поговорив о том, как происходит выделение и освобождение памяти в Python, стоит отдельно поговорить о том, а как язык управляет временем жизни. Для этого в Python реализовано два механизма:

- Счетчик ссылок.
- Механизм сборки мусора.



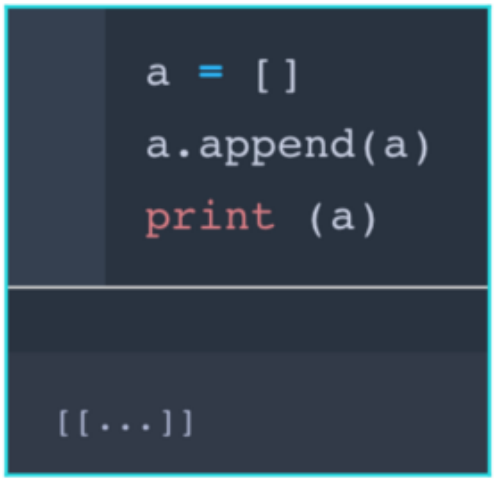
Каждый объект в Python — это, в первую очередь, объект, унаследованный от базового класса PyObject. Этот самый PyObject содержит всего два поля: `ob_refcnt` — количество ссылок, и `ob_type` — указатель на другой объект, тип данного объекта.

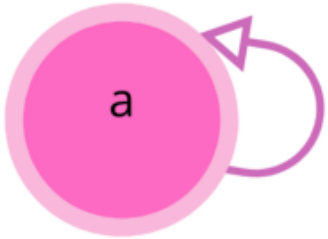
Нас интересует первое поле — `ob_refcnt`. Это счетчик ссылок на конкретный экземпляр данного класса. Каждый раз, когда мы сохраняем объект в новой переменной, массиве и т.п. (то есть когда мы сохраняем где-то ссылку на объект), мы увеличиваем счетчик ссылок на единицу. Каждый раз, когда мы перестаем

использовать переменную, удаляем объект из массива и т.п. (то есть когда ссылка на объект удаляется), мы уменьшаем счетчик ссылок на единицу. Когда он доходит до 0 — объект больше никем не используется и Python его удаляет (в том числе помещая блок, в котором располагался объект, в список пустых).

К сожалению, счетчик ссылок подвержен проблемам в многопоточной среде. Состояния гонки могут приводит к некорректности обновления этого счетчика из разных потоков. Чтобы этого избежать CPython использует GIL — Global Interpreter Lock. Каждый раз, когда происходит работа с памятью, GIL — как глобальная блокировка — препятствует выполнению этих действий одновременно из двух потоков. Он гарантирует, что сначала отработает один, потом другой.

Второй механизм очистки памяти — это сборщик мусора (garbage collector), основанный на идее поколений. Зачем он нам нужен, если есть счетчик ссылок? Дело в том, что счетчик ссылок не позволяет отслеживать ситуации с кольцевыми зависимостями, когда объект A содержит ссылку на B, а B — на A. В таком случае, даже если никто больше не ссылается на объекты A и B, их счетчик всё равно никогда не станет равным нулю и они никогда не удалятся через механизм счетчика ссылок. Для борьбы с такими зависимостями и появился второй механизм (как модуль gc, начиная с версии Python 1.5).





The object is referring to itself. So, the reference count can not be zero.

В отличие от счетчика ссылок, механизм сборки мусора не работает постоянно. Он запускается от случая к случаю на основе эвристик и удаляет только циклические циклы. GC разделяет объекты на три поколения. Каждый новый объект начинает свой путь с первого поколения. Если объект переживает раунд сборки мусора, он переходит к более старому поколению. В младших поколениях сборка происходит чаще, чем в старших. Эта практика является стандартной для такого рода сборщиков мусора и снижает частоту и объем очищаемых данных. Идея простая: чем дольше живет объект, тем с большей вероятностью он проживет еще. То есть новые объекты удаляются гораздо чаще, чем те, которые уже просуществовали достаточно долго.

Каждое поколение имеет индивидуальный счётчик и порог. Счетчик хранит количество созданных минус количество удаленных объектов с момента последнего сбора. Каждый раз, когда вы создаете новый объект, Python проверяет, не превысил ли счетчик поколения пороговое значение. Если это так, Python инициирует процесс сборки мусора.

Поколений всего 3.

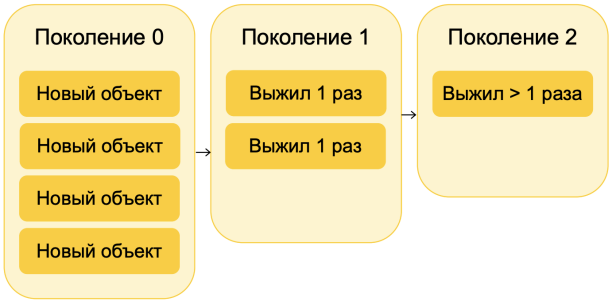
Модуль **gc** включает функции для изменения порогового значения, ручного запуска процесса сбора мусора, отключения процесса сбора мусора и т. д.

Сборщик мусора. Поколения

> **Чтобы часто не проверять долгоживущие объекты, выжившие объекты переносятся в последующие поколения**

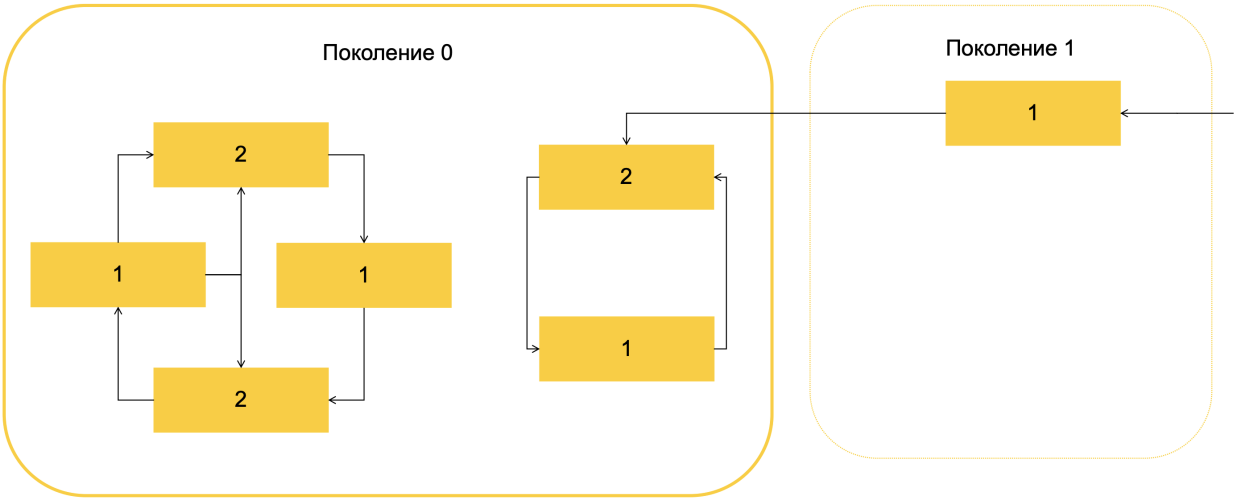
```
>>> import gc
>>> gc.get_threshold()
(700, 10, 10)
```

- `700` – количество **созданий минус удалений объектов**. При превышении – запускается сборка первого поколения
- `10` – количество **сборок мусора** в предыдущем поколении. При превышении – запускаем сборку в этом поколении. (справедливо для двух последних поколений)

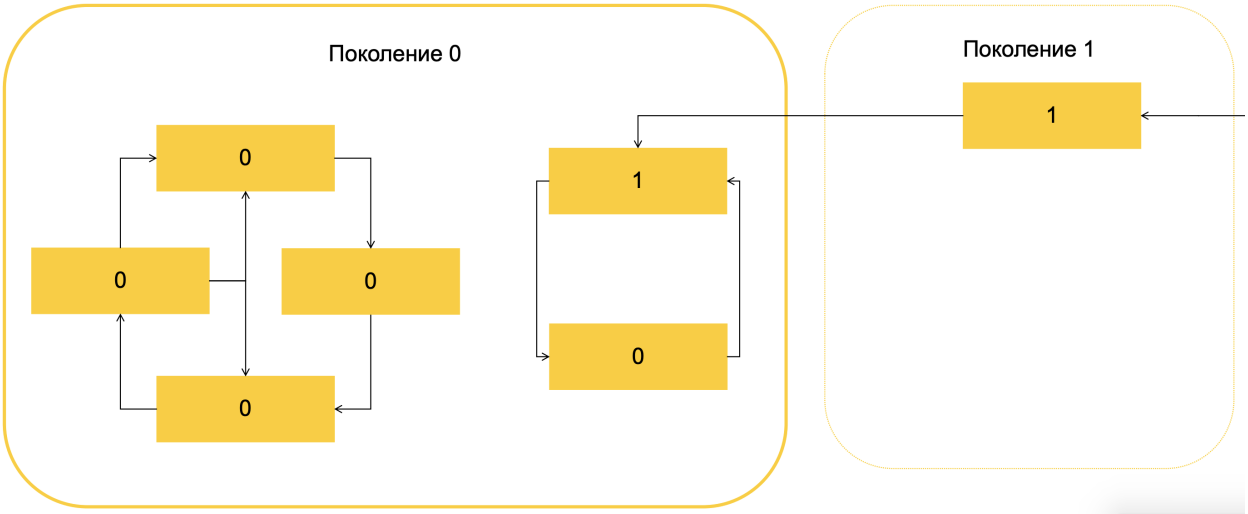


Алгоритм удаления циклических ссылок очень простой. Берем все объекты внутри покаления и вычитаем из счетчика ссылок количество ссылок на объект из текущего поколения.

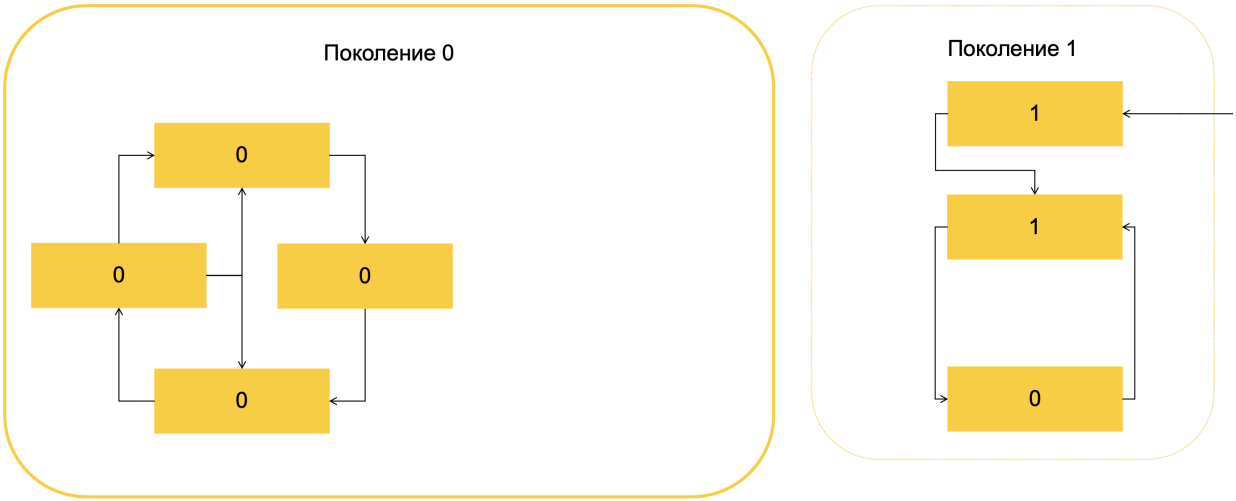
Сборка мусора. Алгоритм



Сборка мусора. Алгоритм



Сборка мусора. Алгоритм



Структуры кода

Питон компилирует свой код в байт-код. Байт-код уже обрабатывается виртуальной машиной. CPython использует стековую виртуальную машину: хранение промежуточных данных в стеке. Есть еще регистровые.

Так, есть интерпретаторы, выполняющие байт-код на JVM (JPython), использующие JIT (PyPy) и другие.

Код в токены/лексемы, они превращаются в дерево AST по правилам граматики, оно уже в байт код.

Байт-код - это код операции и аргумент. В C там большоооой свитч на эти коды.

Есть такая структура - CodeObject. В нем грубо говоря хранится код функции.

Реализация в CPython: CodeObject

> **Содержит код без контекста**

Некоторые поля:

- `co_argcount`: количество аргументов к блоку
- `co_code`: байткод
- `co_consts`: константы (берутся через LOAD_CONST)
- `co_names`: не-локальные переменные
- `co_varnames`: локальные переменные

```
typedef struct {
 PyObject_HEAD
 /* ... */
 PyObject *co_code;      /* instruction opcodes */

 PyObject *co_consts;    /* list (constants used) */
 PyObject *co_names;     /* list of strings(names used)*/
 PyObject *co_varnames; /* local variable names */
 /* ... */
} PyCodeObject;
```

А есть FrameObject. В нем хранится выполнение функции. Так при определении функции создается CodeObject, а при каждом ее выполнении создается FrameObject, связанный с этим CodeObject. То есть при каждом выполнении мы создаем только фрэйм, а код - для всех один.

Реализация в CPython: FrameObject

> **Определяет контекст выполнения**

Некоторые поля:

- `f_back`: ссылка на предыдущий фрейм (используется, например, для построения стека вызовов).
- `f_code`: ссылка на code object
- `f_builtins`: ссылка на встроенный неймспейс
- `f_globals`: ссылка на глобальные переменные
- `f_locals`: локальные переменные
- `f_valuestack`: ссылка на стек выполнения

```
typedef struct _frame {
   PyObject_VAR_HEAD           /* тоже PyObject! */

   PyCodeObject *f_code;       /* code segment */
   PyObject *f_builtins;       /* builtin symbol table (PyDictObject) */
   PyObject *f_globals;        /* global symbol table (PyDictObject) */
   PyObject *f_locals;         /* local symbol table (any mapping) */
   PyObject **f_valuestack;    /* points after the last local */

   /* ... */
   int f_lasti;                /* Last instruction if called */
} PyFrameObject;
```

Фрэйм можно исследовать из питона, но кроме исследования питона, смысла никакого в этом нет.

Есть пример обработки простого кода в презе со второй ссылки.

Про объекты в питоне думаю все известно, что все наследуется от PyObject. В структуре его ничего особенного, есть в [2].

Интересно только про хэш-таблицу:

Кратко о хеш таблице

```
d = {'timmy': 'red', 'barry': 'green', 'guido': 'blue'}
```

Бакет	Хеш	Ключ	Значение
0			
1	8522787127447073495	timmy	red
2			
3			
4			
5	9092791511155847987	barry	green
6			
7	6480567542315338377	guido	blue

Берется хэш - по нему ищем бакет, заполняем его. Если он занят, берем следующий свободный. Поэтому чтобы у нас хэш таблица была эффективной, треть бакетов (хз как посчитали, но вот так) должно быть свободным. В питоне доп память под это не выделяется (хотя до 3.6 выделялось), а хранится распределенно.

Ну и генератор. Это просто объект, который дополнительно сохраняет FrameObject - контекст генератора. Вот и все.

Генераторы. Определение

```
/* _PyGenObject_HEAD defines the initial segment of generator
   and coroutine objects. */

#define _PyGenObject_HEAD(prefix)
    PyObject_HEAD
    struct _frame *prefix##_frame;
    char prefix##_running;
/* ... */
```

Ссылки

1. https://habr.com/ru/company/domclick/blog/530804/ (про память неплохо)
2. https://proglib.io/p/pomnit-vse-kak-rabotaet-pamyat-v-python-2021-03-14 (в принципе про все низкие уровни питона, сама преза тут https://disk.yandex.ru/i/dcNx5Sgix4axOA)
3. https://docs.python.org/3/library/mmap.html (мемори маппед чтение из файла)

Стоит рассказать про то межпроцессное взаимодействие через сокеты беркли

Упомянуть про корутины. Что про них было бы неплохо рассказать в след раз (как используются прям)