CPython

Гейне М.А.

06.11.2024

CPython - это эталонная реализация языка программирования Python, написанная на языке C. Большинство людей, говоря о Python, имеют в виду именно его, и он является интерпретатором Python по умолчанию. Он компилирует код Python в байткод, который затем запускается на виртуальной машине CPython, что делает его наиболее широко используемой средой выполнения Python.

Почему C?

Компиляторы бывают двух видов:

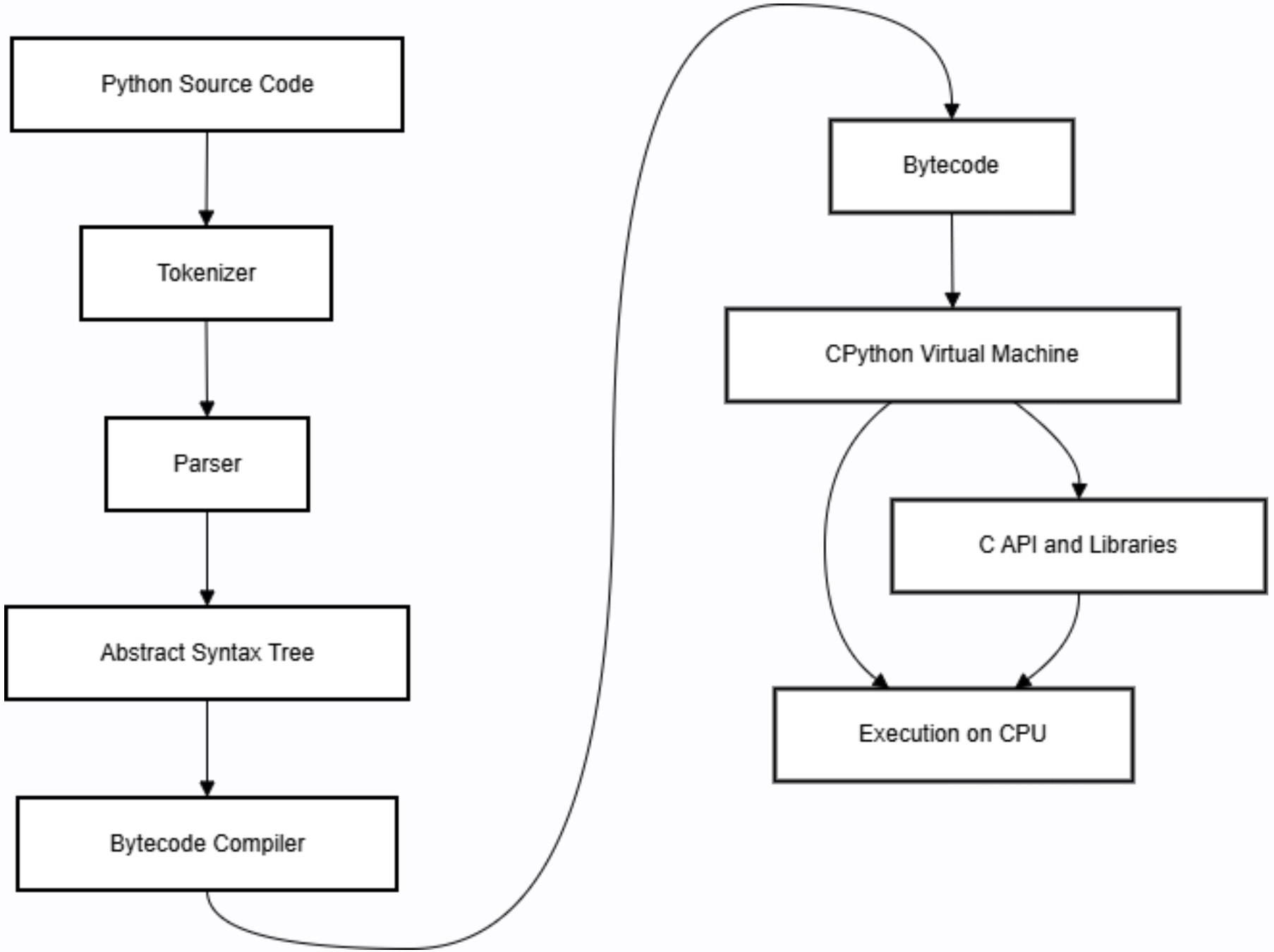
1. Самостоятельные компиляторы - это компиляторы, написанные на языке, который они компилируют, например, компилятор Go
2. Source-to-source компиляторы - это компиляторы, написанные на другом языке, для которого уже есть компилятор

Любой компилируемый язык проходит стадию сборки транспайлером: чтобы собрать свой новый компилятор, нужен другой компилятор. CPython решил сохранять это наследие

Компилируемый или интерпретируемый?

Строго говоря, Pyhton и компилируемый, и интерпретируемый одновременно:

1. **Компиляция в байткод:** Когда запускается скрипт Python, CPython сначала компилирует его в байткод, который является более низкоуровневым, платформенно-независимым представлением кода.
2. **Интерпретация:** Этот байткод затем интерпретируется виртуальной машиной CPython, которая выполняет инструкции в системе.


Python Source Code
Tokenizer
Parser
Abstract Syntax Tree
Bytecode Compiler
Bytecode
CPython Virtual Machine
C API and Libraries
Execution on CPU

Структура CPython

Документация

В `Doc/reference` содержатся полные описания всех возможностей Python, которые служат основой для официальной справки на docs.python.org

Грамматика

Вся спецификация языка Python записана в одном файле `Grammar/Grammar` . Грамматика Python записана в расширенных формах Бэкуса-Наура (EBNF)

Токены

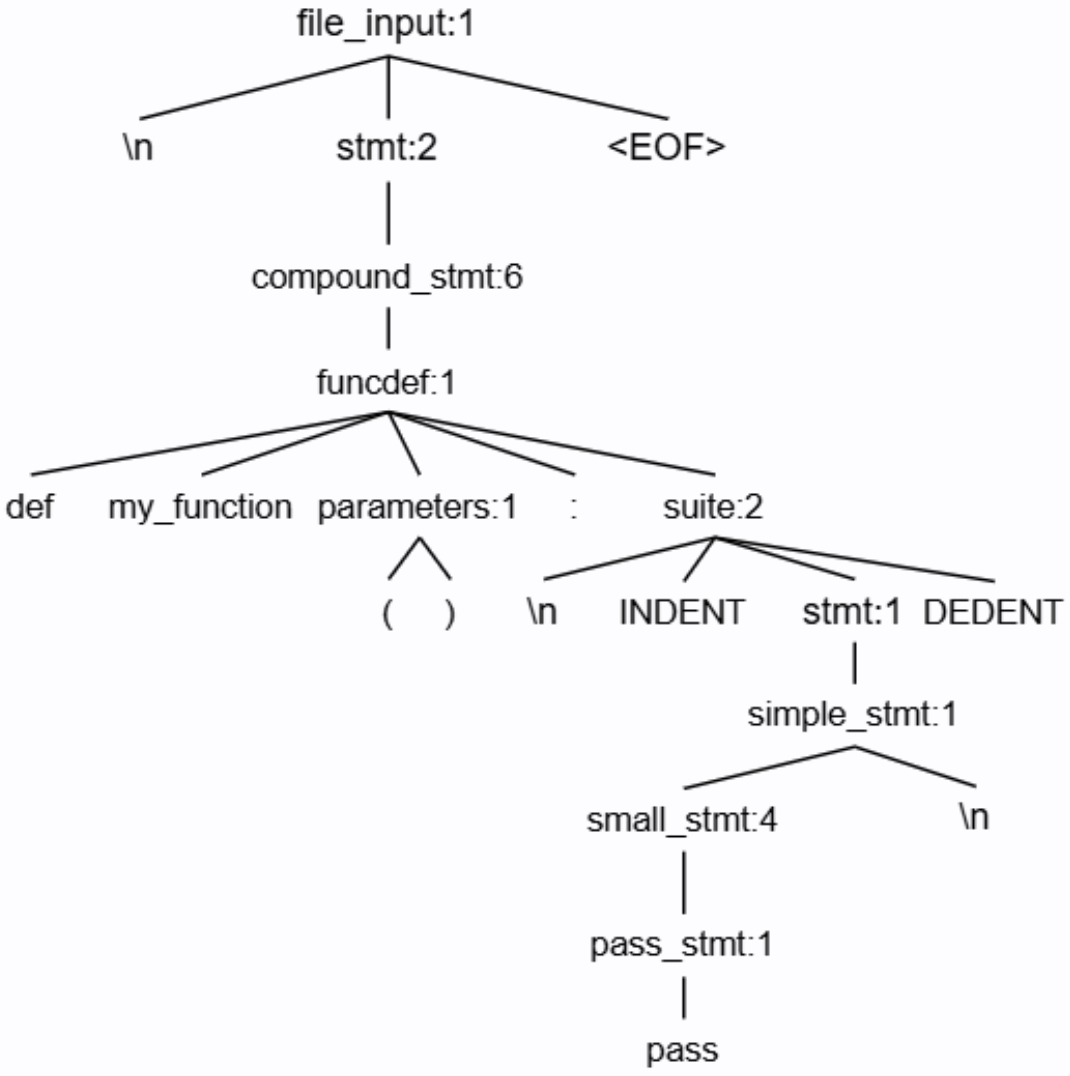
В дополнение к грамматике в файле `Grammar/Tokens` описаны токены: все возможные уникальные конечные выражения (листья AST-дерева) языка. Среди них:

Рассмотрим пример:

Abstract Syntax Tree (AST)

Абстрактное синтаксическое дерево (AST) - это древовидное представление синтаксической структуры исходного кода. В AST:

- Каждый **узел** представляет собой конструкцию в исходном коде, такую как выражения, операторы и управляющие структуры.
- Структура **дерева** отражает иерархическую, вложенную природу языков программирования, где одни утверждения или выражения содержат другие.



Объекты и типы данных

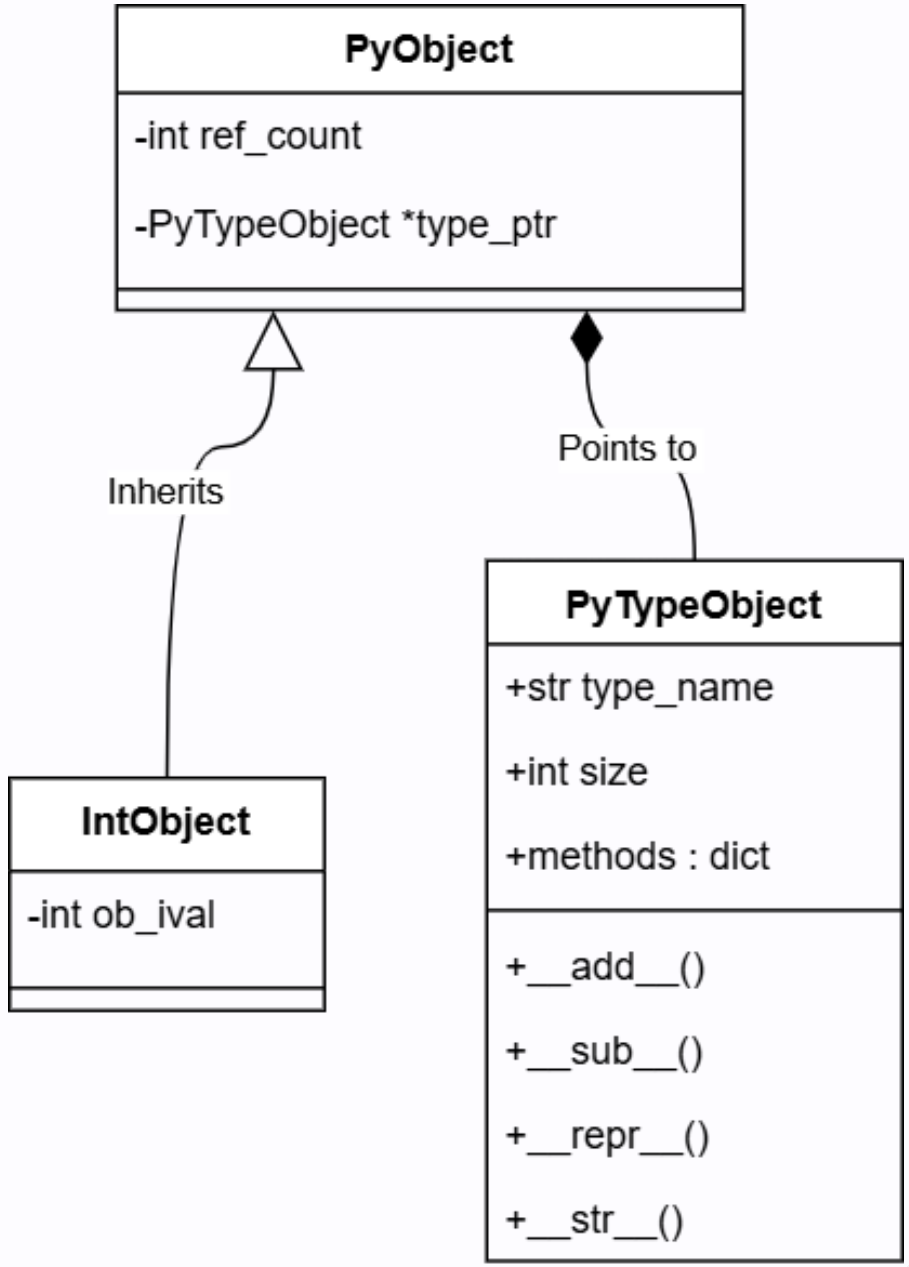
Каждая сущность в Python - это объект, и каждый объект имеет связанный с ним тип данных **type**.

- В Python все, включая целые числа, строки, функции, классы и модули, представлено в виде объекта. Такая конструкция позволяет Python быть очень гибким, поскольку с каждым элементом можно взаимодействовать единообразно
- Каждый объект представлен внутренне структурой `PyObject`, которая является базовой структурой для всех объектов Python в CPython

- В CPython используется сильная и динамическая система типов, то есть типы ассоциируются с объектами (а не с переменными), а проверка типов происходит во время выполнения
- Каждый объект имеет атрибут `type`, указывающий на объект типа (например, `int`, `str`, `list`). Этот объект типа содержит информацию о том, как манипулировать или оперировать с экземплярами данного типа
- Каждый объект основан на C-структуре `PyObject`, которая содержит базовую информацию, такую как количество ссылок и указатели типов
- `PyTypeObject` определяет тип данных и включает методы для операций (таких как сложение или представление строк), характерных для этого типа

Пример: Целочисленные объекты в CPython

- Для `x = 5`, CPython создает объект `int`
- У `int` есть:
  - Счетчик ссылок (для отслеживания количества ссылок на него)
  - Указатель на объект типа `int`, который определяет методы, специфичные для целых чисел (например, сложение или сравнение)



Код как объект

Помимо привычных нам данных, упакованных в объект, в Python также возможно рассмотреть исполняемый код в виде объекта, что раскрывает ряд дополнительных возможностей. К примеру, такое представление позволяет использовать функции в качестве объектов первого класса: передавать их в функции в качестве аргументов, возвращать их как значения, связывать с переменными и так далее

Code Objects

Code Object в Python представляет собой скомпилированный байткод для блока кода (например, функции, класса или модуля). Объекты кода неизменяемы и содержат важную информацию, необходимую для выполнения кода

Ключевые атрибуты Code Object включают:

- `co_code` : Байткод, который будет выполняться
- `co_varnames` : Имена локальных переменных, используемых в блоке кода
- `co_names` : Имена любых других переменных или функций, на которые ссылается блок кода
- `co_consts` : Константы, используемые в коде, как литералы
- `co_filename` , `co_name` и `co_firstlineno` : Информация о том, откуда взят код (например, имя файла, имя функции и номер строки)

Важно: Code Object сам по себе ещё не является вызываемым объектом

Функции

Объект **функции** в Python - это вызываемый объект, который оборачивает объект кода вместе с дополнительным контекстом, таким как глобальная область видимости функции и значения по умолчанию для параметров. При определении функции Python:

1. Создает **Code Object** для тела функции
2. Обертывает этот объект в **объект функции**, который включает такие атрибуты, как:
   - `__code__` : Ссылка на объект кода
   - `__globals__` : Ссылка на глобальную область видимости, в которой была определена функция
   - `__defaults__` и `__kwdefaults__` : Значения аргументов по умолчанию
   - `__closure__` : Ссылка на любые включаемые переменные, если функция является замыкающей

Фреймы

Объект **frame** создается при каждом вызове функции, представляя собой однократное выполнение этой функции. Объекты фрейма отслеживают:

- **Code Object**: Конкретный выполняемый код
- **Локальные и глобальные пространства имен**: Сопоставления для локальных и глобальных переменных
- **Предыдущий фрейм**: Ссылка на фрейм вызывающей функции, позволяющая отследить вызовы
- **Стек и состояние исчислений**: Временные данные, необходимые для оценки выражений внутри функции.

Фреймы очень важны для отслеживания **контекста выполнения** во время вызовов функций и управления **стеком вызовов**. Каждый фрейм соответствует одному вызову функции и содержит:

- `f_locals` : Словарь локальных переменных для этого фрейма
- `f_globals` : Глобальные переменные, доступные в области видимости этого фрейма
- `f_back` : Ссылка на предыдущий фрейм в стеке вызовов (т. е. на функцию, которая вызвала этот фрейм)

После завершения работы функции ее фрейм удаляется из стека вызовов, освобождая ресурсы. Фреймы также важны для отладки и трассировки стека, поскольку они обеспечивают подробный контекст для каждого уровня вызова функции

Стек вызовов

Стек вызовов - это стековая структура данных, хранящая информацию об активных вызовах функций в программе. При каждом вызове функции создается **фрейм**, который помещается в стек вызовов и представляет собой контекст функции (локальные переменные, аргументы и т. д.). Когда функция завершает свою работу, ее фрейм сбрасывается со стека, и управление возвращается к вызывающей функции

- **Создание фрейма**: Каждый вызов функции создает новый **фрейм**, который содержит все необходимые данные для этого вызова (локальные и глобальные переменные, адрес возврата и т. д.)
- **Рост стека**: Когда внутри функции вызывается новая функция, фрейм для новой функции добавляется в верх стека
- **Уменьшение стека**: Когда функция возвращается, ее фрейм удаляется из стека и возобновлется выполнение в фрейме вызывающей функции
- **Рекурсия и глубина стека**: Поскольку каждый вызов функции добавляет фрейм в стек, рекурсивные функции занимают место в стеке при каждом рекурсивном вызове. В Python существует максимальная глубина **рекурсии** (по умолчанию около 1 000), ограничивающая количество фреймов, которые могут быть добавлены в стек. Глубокая рекурсия сверх этого предела приводит к `RecursionError`.

Стек данных (Evaluation stack)

Внутри каждого фрейма есть **стек данных** (также называемый стеком **операнд**), используемый для управления промежуточными значениями во время оценки выражений. Этот стек специфичен для **модели виртуальной машины (PVM)** Python, основанной на стеках, где каждая операция кладёт или вынимает значения из стека для оценки выражений

Стек данных является ключевым в **выполнении байткода**:

- **Push**: Каждый операнд (например, переменные, литералы, результаты подвыражений) кладётся в стек оценки
- **Pop**: Каждая операция (например, сложение, вызов функции) снимает операнды со стека, выполняет операцию и помещает результат обратно в стек

В стеке временно хранятся промежуточные значения, которые будут использоваться для дальнейших операций в сложных выражениях

Пример стека данных

Для выражения `3 + 5 * 2`, Python оценивает его, сначала помещая операнды и операторы в стек, а затем выгружая их по мере необходимости для вычисления результата в соответствии с приоритетом операторов.

Инструкции байткода:

1. Поместить `3` в стек
2. Переместите `5` в стек
3. Переместите `2` в стек
4. Умножьте `5 * 2` (извлечь `5` и `2`, положить `10`)
5. Сложить `3 + 10` (извлечь `3` и `10`, положить `13`)

Управление памятью

В отличие от многих низкоуровневых языков, где задача управления памятью ложится на программиста, в Python за управление памятью отвечает интерпертатор, задействуя два механизма:

1. Подсчёт ссылок
2. Сборка мусора

Подсчет ссылок

- Каждый объект в Python имеет **счетчик ссылок**, который отслеживает, сколько ссылок на него указывают. Этот счетчик увеличивается при создании новой ссылки (например, при присвоении объекта переменной) и уменьшается, когда ссылка удаляется или выходит из области видимости
- Когда счетчик ссылок объекта падает до нуля, это означает, что на объект не осталось ссылок, поэтому CPython может освободить от него память

Garbage Collection (GC)

Хотя подсчет ссылок работает хорошо, он может дать сбой в ситуациях с **круговыми ссылками** - например, два объекта, которые ссылаются друг на друга. Циклические ссылки не могут быть решены только подсчетом ссылок, поскольку каждый объект в цикле всегда имеет ненулевой счетчик ссылок

В Python есть дополнительный **сборщик мусора**, который периодически ищет и освобождает круговые ссылки. Этот сборщик мусора:

- Работает как **поколенческий сборщик мусора**, который группирует объекты в три поколения на основе их "возраста" (как долго они существуют)
- Сборщик чаще фокусируется на младших поколениях, поскольку они чаще содержат объекты, которые можно быстро деаллоцировать

Поколения объектов

Сборщик мусора (GC) в Python делит объекты на **три поколения** в зависимости от их возраста и продолжительности жизни. Идея состоит в том, что **молодые объекты** (созданные недавно) с большей вероятностью станут недоступными раньше, в то время как **старые объекты** сохраняются дольше и реже нуждаются в сборе

Поколение 0 (самое молодое поколение):

- Это начальное, "молодое" поколение, в котором размещаются новые объекты
- Объекты в этом поколении недолговечны, поэтому многие из них быстро становятся недоступными и требуют сбора
- Сборщик мусора часто сканирует поколение 0, что делает его более эффективным для быстрого сбора объектов, которые больше не используются

Поколение 1 (среднее поколение):

- Если объект пережил цикл сборки мусора в поколении 0, он переходит в поколение 1
- Поколение 1 сканируется реже, чем поколение 0, потому что объекты, пережившие первоначальную сборку, с меньшей вероятностью станут недоступными
- Это поколение является "золотой серединой" для объектов, которые могут существовать более длительное время, но все равно могут быть отброшены

Поколение 2 (старшее поколение):

- Объекты, пережившие несколько циклов сборки мусора в поколении 1, переходят в поколение 2
- Поколение 2 сканируется еще реже, исходя из предположения, что эти старые объекты долгоживущие наименее вероятно станут мусором
- Это поколение содержит объекты, которые либо необходимы на протяжении всего времени работы программы (например, некоторые глобальные переменные и данные модулей), либо являются относительно статичными

Модель поколений использует **гипотезу поколений**: идею о том, что большинство объектов, создаваемых программой, либо очень недолговечны, либо будут существовать в течение всего времени работы программы. Благодаря такому подходу сборщик мусора Python может более эффективно направлять свои ресурсы:

- **Частый сбор поколения 0** отлавливает и удаляет многие недолговечные объекты на ранней стадии, минимизируя использование памяти
- **Нечастый сбор поколений 1 и 2** снижает расходы на повторную проверку долгоживущих объектов, поскольку они с меньшей вероятностью будут нуждаться в сборе

Memory Pools and Arenas

Чтобы повысить эффективность использования памяти, Python использует механизм **пула памяти**. Вместо того чтобы напрямую взаимодействовать с операционной системой при каждом выделении, Python управляет памятью с помощью **пулов** и **арен**:

- **Пулы**: Небольшие объекты (обычно < 512 байт) выделяются из пулов памяти, которые представляют собой блоки памяти, предварительно выделенные Python. Это уменьшает фрагментацию и ускоряет выделение памяти
- **Арены**: Пулы организованы в арены, которые представляют собой более крупные куски памяти, выделяемые ОС. Каждая арена делится на пулы для объектов разного размера

Global Interpreter Lock (GIL)

Глобальная блокировка интерпретатора (GIL) - это мьютекс, который защищает доступ к объектам Python в CPython, обеспечивая одновременное выполнение байткода Python только одним потоком даже на многоядерных системах. GIL упрощает управление памятью, особенно для системы подсчета ссылок в CPython, но ограничивает истинную многопоточную производительность для задач, привязанных к процессору

Почему существует GIL?

GIL был введен для упрощения управления памятью путем защиты внутренних структур памяти Python от одновременного доступа, что позволяет CPython управлять памятью без сложных механизмов потокобезопасности

1. **Подсчет ссылок**: CPython использует подсчет ссылок для управления выделением и удалением памяти для объектов. Без GIL каждому потоку потребовался бы свой собственный механизм для безопасного управления подсчетом ссылок, что может привести к значительным накладным расходам и сложности
2. **Компромисс с производительностью**: GIL упрощает и ускоряет написание расширений на C, не требуя дополнительной синхронизации в коде, что было преимуществом исторически, когда большинство процессоров были одноядерными

3. **Concurrency vs Parallelism**: Python не поддерживает concurrency (выполнение нескольких задач) с помощью GIL, но поддерживает parallelism (одновременное выполнение нескольких задач на разных ядрах) для задач, привязанных к процессору

Поскольку только один поток может выполнять байткод Python в любой момент времени, потоки Python, привязанные к процессору, не получают преимущества от использования нескольких ядер. В результате программы, требовательные к процессору, могут не получить значительного прироста скорости за счет дополнительных потоков

Спасибо за внимание