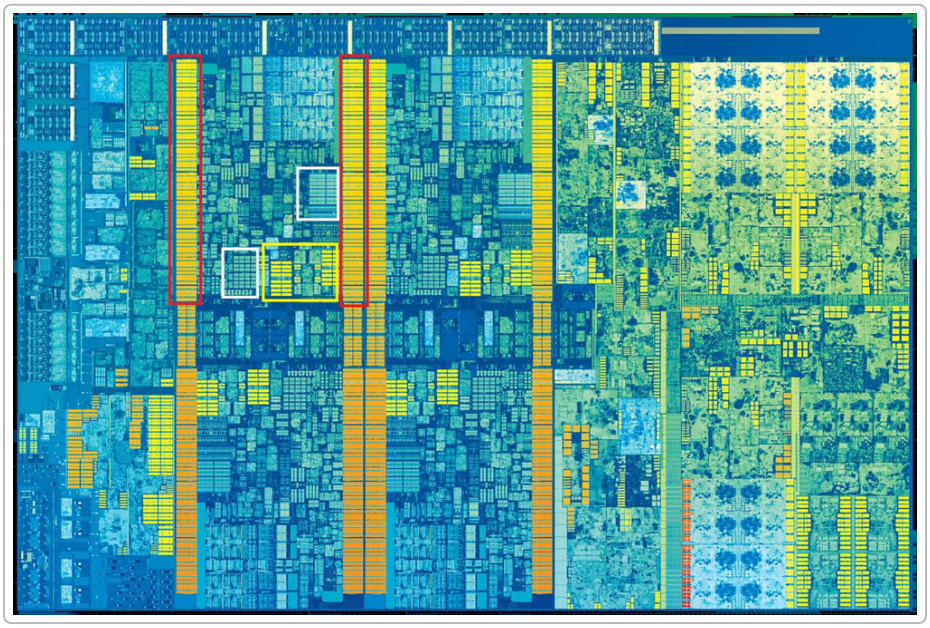
ChatGPT

Внутреннее устройство Python и производительность



Современные процессоры имеют иерархическую организацию кэш-памяти для ускорения доступа к данным. **Кэш** – это небольшая быстрая память, куда процессор копирует часто используемые данные из основной памяти [1]. При обращении к данным CPU сначала проверяет кэш: при попадании (*cache hit*) значение берётся из кэша – это значительно быстрее, чем при обращении в медленную оперативную память [2]. В случае промаха кэш возвращается к чтению из основной памяти, что заметно замедляет операцию.

- **L1** – самый быстрый кэш, обычно по несколько десятков килобайт на ядро (часто разделён на данные и инструкции); время доступа порядка 5 тактов [3].
- **L2** – больше L1 (до нескольких сотен килобайт на ядро) и медленнее (приблизительно в 2 раза дольше) [4].
- **L3** – ещё больше (несколько мегабайт на группу ядер) и медленнее L2 (≈30–40+ тактов) [5]; обычно общий для всех ядер процессора.

Кэширование значительно влияет на производительность приложений, включая Python. В CPython объекты хранятся в куче с многочисленными указателями, что приводит к растрепанному доступу к памяти. Например, при последовательном обходе списка Python (поэлементном суммировании) память читается подряд – это более «дружелюбно» к кэшу. А при случайном обращении к элементам списка кэш постоянно промахивается, и производительность падает [6] [7]. Рассмотрим пример на Python:

```
import random

def linear_sum(n):
    arr = list(range(n))
```

```
    s = 0
    for x in arr:
        s += x

def random_sum(n):
    arr = list(range(n))
    s = 0
    for _ in range(n):
        s += arr[random.randrange(n)]
```

Функция `linear_sum` проходит по списку последовательно (много попадает в кэш), а `random_sum` – выбирает случайные индексы (частые промахи). При больших `n` случайный доступ заметно медленнее из-за перегруза кэша [6].

Процесс трансляции Python-кода

При запуске скрипта Python сначала выполняются лексический и синтаксический анализ. Лексер (модуль `tokenize`) разбивает текст на **токены** (ключевые слова, идентификаторы, литералы, операторы и т.д.) [8]. Парсер строит **абстрактное синтаксическое дерево** (AST) – древовидное представление структуры программы [9] [10]. В CPython используется сгенерированный на основе `Grammar/python.gram` парсер (модуль `Parser/parser.c`) [9]. Семантический анализ включает проверку правильности конструкции (например, правил `try/except`, однозначности имён) и построение таблиц символов.

После разбора AST компилятор проходит по дереву и генерирует **байткод** – низкоуровневые инструкции виртуальной машины Python. Байткод хранится в объекте типа `code` (PyCodeObject) и содержит последовательность опкодов и аргументов [11]. Например, выражение `x = 5 + 3` при компиляции выдаёт объект `code` с полями `co_code`, `co_consts`, `co_names` и т.д. Здесь `co_code` – собственно байткод, `co_consts` – кортеж констант (в данном случае `(8, None)` после оптимизации), `co_names` – имена переменных (например, `('x',)`), `co_varnames` – локальных имён (имена параметров и локальных переменных) [12] [13]. Комментарий: компилятор Python автоматически выполняет простые оптимизации (например, складывает `5+3` в `8`) [14].

Сгенерированный байткод выполняется на **виртуальной машине Python** (PVM). Однако перед исполнением код может быть сохранён для кеширования. При импорте модуля CPython проверяет наличие компилированного файла `.pyc` в папке `__pycache__`. Этот файл содержит сериализованный объект `code` (через модуль `marshal`) и служит для ускорения последующих запусков [15] [16]. Модуль `py_compile` позволяет явно сгенерировать файл `.pyc` из исходника [17] [16]. При перекомпиляции Python проверяет метку времени и размер исходного `.py` – если они совпадают, используется существующий `.pyc`, пропуская лексинг/парсинг.

- `tokenize`: модуль лексического анализа; генерирует токены из исходного кода Python [8].
- `ast`: модуль для работы с AST. Позволяет с помощью `ast.parse()` или флага `compile(..., ast.PyCF_ONLY_AST)` получить дерево синтаксиса программы [10].
- `mypy`: внешняя утилита статической типизации. Анализирует исходный код и аннотации типов (основана на `typed_ast` / `ast`).

- `dis`: модуль дизассемблера, печатает читабельное представление байткода CPython [18]. Позволяет увидеть инструкцию и аргумент (например, `LOAD_CONST 0 (8)`).
- `py_compile`: модуль для компиляции `.py` в `.pyc`. Знает формат `.pyc` (PEP 3147) и умеет поместить его в `__pycache__` [16].
- `marshal`: встроенный сериализатор Python-объектов. Используется для записи и чтения байткода `.pyc` (формат специфичен для Python и может меняться) [19].

Класс `code` (объект кода) представляет скомпилированный блок (модуль, функцию, выражение). У него есть атрибуты: `co_argcount` (число позиционных аргументов), `co_varnames` (имена параметров и локальных), `co_code` (байткод), `co_consts`, `co_names` и др. Например:

```
def f(a, b):
    x = a + b
    return x

c = f.__code__
print(c.co_argcount)    # 2
print(c.co_varnames)    # ('a', 'b', 'x')
print(c.co_consts)      # (<class 'NoneType'>, )
```

Здесь `co_argcount=2`, `co_varnames=('a','b','x')`, `co_consts` содержит `None` как константу возврата. Дизассемблер `dis` поможет отобразить `co_code`.

Компиляция или трансляция?

Python сочетает в себе концепции компиляции и интерпретации. **CPython** при запуске кода сначала компилирует его в байткод (файл `.py` → AST → `code`) [9], а затем виртуальная машина PVM интерпретирует этот байткод. Таким образом, исходник не транслируется непосредственно в машинные команды CPU, а проходит через промежуточный байткод.

Для сравнения, **Cython** позволяет превратить (часто сильно типизированный) Python-код в C и скомпилировать его в расширение для CPython. Пошагово это делается так: 1. Установить Cython: `pip install cython`.
2. Сохранить или переименовать скрипт `module.py` в `module.pyx`.
3. Создать `setup.py`:

```
from setuptools import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize("module.pyx"))
```

4. Запустить сборку: `python setup.py build_ext --inplace`.
5. После этого в текущем каталоге появится `module.so` (или `module.pyd`), который можно подключать через `import module`.

Таким образом Cython-код фактически собирается в машинный код (через компиляцию C), в отличие от байткода CPython.

Байт-код – это низкоуровневый, платформонезависимый код (стековые инструкции) для PVM, тогда как **машинный код** – набор инструкций конкретного CPU. Байткод Python сам по себе не выполняется CPU; интерпретатор (PVM) декодирует и исполняет его на машине [18]. Отличается между реализациями: байткод CPython – внутреннее представление, и другие реализации (PyPy, Jython и др.) могут иметь свои IR или JIT-компиляцию.

Чтобы посмотреть байткод функции, можно использовать модуль `dis`. Например:

```
import dis
def f(x, y): return x + y
dis.dis(f)
```

Выдаст нечто вроде:

```
  2           0 LOAD_FAST    0 (x)
              2 LOAD_FAST    1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
```

Это означает: взять `x` и `y` со стека, выполнить `BINARY_ADD` (сложить), вернуть результат. Для численного примера `x = 5 + 3` байткод может выглядеть так:

```
0 LOAD_CONST 0 (8)
2 STORE_NAME 0 (x)
4 LOAD_CONST 1 (None)
6 RETURN_VALUE
```

Инструкция `LOAD_CONST 0 (8)` кладёт на стек число 8 (результат 5+3, вычисленного на этапе компиляции), `STORE_NAME 0 (x)` сохраняет его в переменную `x`, затем возвращается `None`.

PVM (Python Virtual Machine)

PVM – это внутренняя машина CPython, которая исполняет байткод. Она реализована как цикл «fetch-decode-execute»: последовательно берёт байт из `co_code`, декодирует в функцию C (например, `do_binary_add` для `BINARY_ADD`) и выполняет её, работая со стеком и объектами [20]. Главной «сердцевиной» является функция `PyEval_EvalFrameEx`, которая берёт текущий фрейм (код функции) и в цикле обрабатывает его инструкции [20]. Каждый фрейм содержит своё состояние: указатель на код (`co`), указатель на текущую инструкцию, стек вычислений, локальные переменные, ссылки на внешние и глобальные области.

Сборка мусора и ресурсы: CPython управляет памятью в первую очередь подсчётом ссылок (refcount) – у каждого объекта есть счётчик ссылок, и при его обнулении объект сразу освобождается [21]. Однако счётчик ссылок не освобождает циклические структуры, потому CPython использует и **сборщик мусора** (модуль `gc`) для выявления циклов ссылок [22]. GC в CPython – поколенческий: все объекты делятся на поколения 0–2 по числу сборок. Сборка циклов запускается автоматически, когда количество выделений превысит порог [23] (значение порога

можно узнать через `gc.get_threshold()`). Таким образом, пулы памяти, не достигаемые из активных объектов, обнаруживаются и очищаются.

JIT-компиляция

JIT (Just-In-Time) – технология динамической компиляции байткода в машинный код во время исполнения для ускорения работы. Идея в том, чтобы на «горячих» участках кода (hot paths) заменить интерпретацию байткода на непосредственный машинный код, близкий по скорости к статически скомпилированному [24]. При этом JIT-компилятор анализирует выполнение программы, собирает статистику, компилирует наиболее часто исполняемые функции («адаптивная оптимизация») и при необходимости выполняет динамическую рекомпиляцию [25].

Плюсы JIT: значительно повышает скорость «тяжёлых» задач и вычислений благодаря оптимизациям во время исполнения [24] [25]. **Минусы:** увеличивает время старта (надо сгенерировать код), расходует больше памяти на хранение скомпилированного кода, усложняет реализацию интерпретатора. В CPython прямой JIT отсутствует, но его идеи применяются в альтернативных реализациях (см. ниже). Связь с кэшем: часто генерируемый JIT-код может лучше укладываться в CPU-кэш и использовать оптимизированные инструкции процессора.

Реализации Python

Помимо CPython (стандартной реализации на C с GIL), существуют другие варианты:

- **CPython** – официальная реализация на C с GIL, самая распространённая.
- **Cython** – компилятор Python->C, позволяет добавлять статическую типизацию и генерирует расширения для CPython [26]. Ускоряет численные алгоритмы и интеграцию с C/C++ библиотеками.
- **PyPy** – реализация на RPython с трассирующим JIT-компилятором. Часто быстрее CPython на долгих вычислениях и не требует модификаций кода, но может проигрывать на коротких скриптах из-за накладных расходов на JIT [27].
- **Jython** – реализация на Java (JVM). Позволяет писать Python-код, использующий Java-классы, без GIL. Поддерживает большинство стандартных возможностей Python, но не может напрямую использовать CPython C-расширения [28].
- **GraalPython** (GraalPy) – реализация на GraalVM/Truffle (Java). Совместима с CPython-средой, но запускается на JVM с JIT-компилятором Graal [29]. Часто используется для интероперабельности с другими языками JVM.
- **nogil** – экспериментальная ветвь CPython без глобального интерпретатора (GIL), также называемая *free-threaded* CPython. Это прототип для многопоточной сборки Python 3.13 и выше [30].
- **Pyodide** – порт CPython на WebAssembly (Emscripten) для браузера. Позволяет запускать обычный Python и устанавливать пакеты в веб-окружении [31].
- **Brython** – транспилятор Python→JavaScript. Позволяет писать клиентский код в браузере на Python (конвертируется в JS) [32].
- **MicroPython** – облегчённая реализация для микроконтроллеров. Поддерживает подавляющее большинство синтаксиса Python, но с урезанной стандартной библиотекой для встраиваемых систем [33].

Стек и куча в контексте Python

В CPython все объекты размещаются в **куче** (выделяемой через C malloc) – это приватная область памяти, где хранятся данные (числа, строки, структуры данных и т.д.) [34]. В отличие от этого, **стек** используется для организации контекста выполнения: когда вызывается функция, для неё создаётся *фрейм* (активационная запись) со своими локальными переменными, аргументами и указателем на текущую инструкцию [35]. Фрейм (объект `PyFrameObject`) также содержит ссылку на объект кода, глобальные и встроенные области видимости. После возврата из функции её фрейм удаляется со стека, освобождая локальные переменные [35]. При рекурсии на стеке накапливаются несколько фреймов (по одному на каждый уровень вложенности), и они поочерёдно снимаются при выходе из функций.

С введением Python 3.11 добавлена инструкция `RESUME` для байткода. Она сигнализирует интерпретатору о начале или возобновлении исполнения фрейма [36]. Всё, что происходит до первой инструкции `RESUME`, считается подготовкой фрейма к выполнению (инициализацией локальных переменных, блока `finally` и т.д.) [36]. При анализе байткода обычно игнорируют инструкции до первого `RESUME`.

Итого: стек (фреймы) хранит временные данные выполнения (локальные переменные, адрес возврата), а куча – динамические объекты, создаваемые во время работы программы [34] [35]. Это разграничение важно для понимания управления памятью и того, как работает интерпретатор Python.

Источники: авторитетные статьи и документация Python [1] [3] [11] [18] [24] [26] [28] [30] [21] (см. сноски).

---

[1] [2] Кэш процессора — Википедия
https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0

[3] [4] [5] Зачем процессорам нужен кэш и чем отличаются уровни L1, L2, L3 / Хабр
https://habr.com/ru/companies/vdsina/articles/515660/

[6] [7] Memory Locality and Python Objects
https://www.naftaliharris.com/blog/heapobjects/

[8] tokenize — Tokenizer for Python source — Python 3.13.3 documentation
https://docs.python.org/3/library/tokenize.html

[9] [11] [12] [14] [15] [20] How Python Compiles Source Code into Bytecode and Executes It Using the PVM
https://www.sparkcodehub.com/python-bytecode-pvm-technical-guide

[10] ast — Abstract Syntax Trees — Python 3.13.3 documentation
https://docs.python.org/3/library/ast.html

[13] Code Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/code.html

[16] [17] py_compile — Compile Python source files — Python 3.13.3 documentation
https://docs.python.org/3/library/py_compile.html

[18] dis — Disassembler for Python bytecode — Python 3.13.3 documentation
https://docs.python.org/3/library/dis.html

19 marshal — Internal Python object serialization — Python 3.13.3 documentation
https://docs.python.org/3/library/marshal.html

21 22 23 Garbage Collection in Python | GeeksforGeeks
https://www.geeksforgeeks.org/garbage-collection-python/

24 25 JIT-компиляция — Википедия
https://ru.wikipedia.org/wiki/JIT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F

26 27 28 29 32 33 PythonImplementations - Python Wiki
https://wiki.python.org/moin/PythonImplementations

30 GitHub - colesbury/nogil: Multithreaded Python without the GIL
https://github.com/colesbury/nogil

31 Pyodide — Version 0.27.6
https://pyodide.org/

34 Does Python have a stack/heap and how is memory managed? - Stack Overflow
https://stackoverflow.com/questions/14546178/does-python-have-a-stack-heap-and-how-is-memory-managed

35 Python:Stack and Heap Memory. Memory management is a critical aspect... | by Divya Rai | Medium
https://medium.com/@raidivya/your-first-steps-in-python-stack-and-heap-memory-4bd587ec5b4c

36 The RESUME opcode has the same line number as the function definition and "breaks" previous behaviour - Core Development - Discussions on Python.org
https://discuss.python.org/t/the-resume-opcode-has-the-same-line-number-as-the-function-definition-and-breaks-previous-behaviour/21972