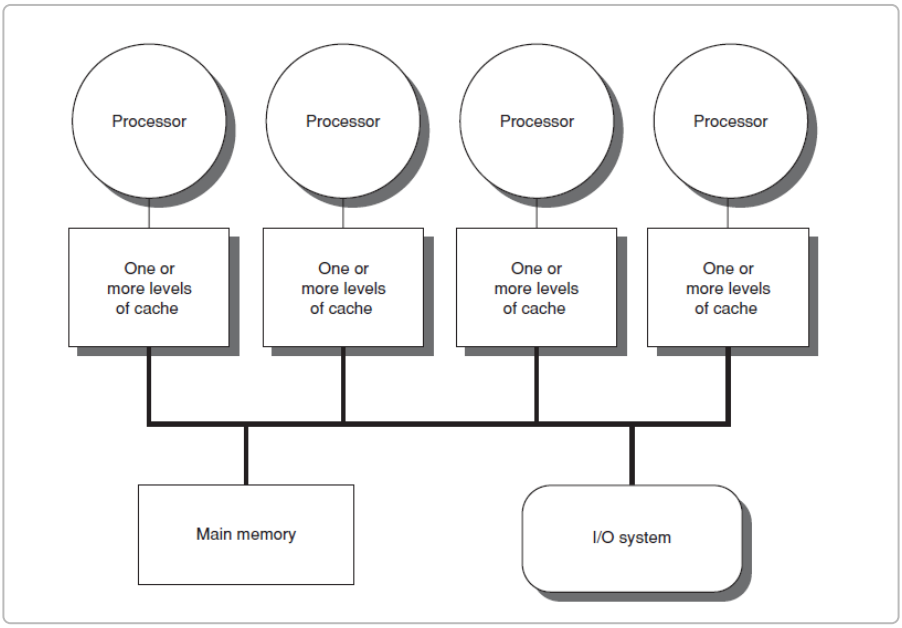
ChatGPT

Конспект: Архитектура исполнения Python и связанные технологии

1. Кэш процессора (CPU Cache)

Что такое кэширование? Кэш процессора – это небольшая сверхбыстрая память, расположенная ближе к ядрам CPU, которая хранит копии часто используемых данных из оперативной памяти [1]. При обращении к данным процессор сначала проверяет кэш: если нужные данные уже там (cache hit), они читаются мгновенно из кэша, что значительно быстрее доступа к ОЗУ; если же данных нет (cache miss), процессор вынужден обращаться к более медленной основной памяти [2]. Таким образом, кэширование уменьшает среднее время доступа к памяти, позволяя CPU меньше простаивать в ожидании данных.

Зачем нужны уровни L1, L2, L3? Разные уровни кэша образуют иерархию памяти: более высокий уровень (ближе к ядру) быстрее, но меньшего объема, тогда как низкие уровни больше по размеру, но медленнее. Например, **L1** – самый быстрый и маленький кэш (обычно десятки КБ) на каждом ядре, **L2** – средний по размеру (сотни КБ – несколько МБ) и скорость, может быть выделен на ядро или делиться между парой ядер, **L3** – еще больший (несколько МБ до десятков МБ) и более медленный, обычно общий для всех ядер на кристалле [3] [4]. Такая многоуровневая организация позволяет хранить максимально часто запрашиваемые данные в кэше верхнего уровня, а менее востребованные – в более емких нижних уровнях. В результате процессор реже обращается к ОЗУ и может работать на более высокой тактовой частоте [3].



Пример: архитектура многоуровневого кеша — каждый процессор имеет один или несколько уровней кэша, объединенных общей шиной с оперативной памятью.

Современные CPU обычно имеют как минимум три уровня кэшей: раздельный L1 для инструкций и данных, L2 (часто на каждое ядро) и общий L3 [5]. Уровни кэша работают прозрачно для

программиста – аппаратура сама управляет тем, какие данные хранить в кэше. Однако эффективность кэширования может зависеть от локальности данных: программы, которые повторно используют одни и те же данные (временная локальность) или обращаются к соседним участкам памяти (пространственная локальность), лучше задействуют кеш и работают быстрее. В целом, иерархия кэшей – ключевой механизм ускорения практически всех вычислений за счет компромисса между скоростью и размером памяти: маленькие быстрые кэши дополняются большими, но медленными, приближая часто нужные данные к процессору. **Вывод:** кеширование существенно повышает производительность, позволяя процессору работать со скоростью, близкой к скорости кеш-памяти, а не основной памяти.

2. Трансляция Python-кода: от исходника к байт-коду

Когда мы запускаем Python-код, происходит серия шагов **компиляции** и **исполнения**. Несмотря что Python считается интерпретируемым языком, **его интерпретатор не исполняет исходный код напрямую** – сначала код транслируется в промежуточное представление (байт-код) [6]. Рассмотрим основные этапы этой трансляции:

- **Лексический анализ (Lexer)** – исходный текст программы преобразуется в последовательность **токенов**. Токен – это атомарный фрагмент исходного кода: ключевое слово, идентификатор, число, оператор, отступ и т.п. Например, для строки `x = 2 + 3` лексер выделит токены: имя `x`, оператор `=`, число `2`, оператор `+`, число `3` и разделитель новой строки. В Python лексический анализ можно продемонстрировать с помощью модуля `tokenize`: он читает строку и разбивает её на токены:

```
import io, tokenize

code = "x = 2 + 3\nprint(x)\n"
tokens = tokenize.tokenize(io.BytesIO(code.encode('utf-8')).readline)
for tok in tokens:
    if tok.type not in (tokenize.ENCODING, tokenize.ENDMARKER):
        print(tokenize.tok_name[tok.type], '->', tok.string)
```

В результате получим список токенов исходного кода:

```
NAME -> x
OP -> =
NUMBER -> 2
OP -> +
NUMBER -> 3
NEWLINE -> (перевод строки)
NAME -> print
OP -> (
NAME -> x
OP -> )
NEWLINE -> (перевод строки)
```

Здесь `NAME` обозначает идентификаторы (имена переменных, функций и т.д.), `OP` – операторские символы, `NUMBER` – числовые литералы, `NEWLINE` – конец строки. Лексер также

распознаёт отступы (INDENT/DEDENT) и другие служебные токены. Полученная последовательность токенов передаётся следующему этапу.

- **Синтаксический анализ (Parser)** – на основе токенов строится **дерево разбора**, а затем **абстрактное синтаксическое дерево (AST)**. Python использует PEG-грамматику (Parsing Expression Grammar) для разбора исходника [7]. Парсер проверяет корректность синтаксиса и структура программы представляется в виде дерева узлов, отражающих вложенность управляющих конструкций, выражений и т.д. Узлы AST описывают программу в виде иерархии объектов: например, узел типа `Module` содержит список операторов верхнего уровня, узел `Assign` представляет присваивание с дочерними узлами для переменной и выражения, узел `BinOp` – бинарную операцию с подузлами для операндов и т.д. Модуль `ast` в Python позволяет работать с AST. Например:

```
import ast
tree = ast.parse("x = 2 + 3")
print(ast.dump(tree, include_attributes=False, indent=4))
```

Вывод покажет древовидную структуру AST: корневой узел `Module`, внутри него `Assign` с целевой переменной `x` и значением, представленным узлом `BinOp` с операцией `Add` и двумя числовыми константами. AST удобно использовать для статического анализа кода и различных трансформаций. На этапе компиляции AST – это основа для генерации байт-кода. *(Примечание: в CPython 3.9+ используется PEG-парсер; до этого была LL(1)-грамматика. AST-структуры определяются с помощью ASDL-описания и генерируются автоматически при сборке интерпретатора.)*

- **Семантический анализ** – этап, следующий за синтаксическим разбором. В динамичном языке Python он минимален по сравнению с компилируемыми языками, но включает некоторые проверки и подготовки: резолвинг имен (локальные или глобальные переменные), построение таблицы символов, вычисление некоторых констант на этапе компиляции. Например, компилятор определяет, какие имена являются локальными для функции, где нужны ячейки для замыкания, может выполнять оптимизации вроде объединения последовательных строк или константного разворачивания выражений (constant folding). Глубокого анализа типов в рантайме CPython не делает, однако существует сторонний инструмент **mypy** – статический анализатор типов. `mypy` выполняет семантический анализ аннотированного Python-кода перед исполнением: проверяет соответствие типов переменных, возвращаемых значений функциям и т.д. Это помогает отловить ошибки, не влияя на сам процесс исполнения (типизация Python остаётся динамической, и mypy – лишь внешняя проверка).

- **Генерация байт-кода** – на основе AST (и результатов семантического анализа) компилятор CPython генерирует последовательность **байт-кодовых инструкций**. Байт-код – это низкоуровневые инструкции виртуальной машины Python, независимые от конкретной архитектуры CPU [8]. В CPython байт-код представляет собой опкоды – числовые коды действий (например, загрузить константу, сложить два значения, вызвать функцию) с возможными аргументами. Эти инструкции реализованы в самом интерпретаторе. Байт-код CPython имеет стековую архитектуру: большинство команд берут операнды из стека и помещают результат обратно на стек. Например, высокоуровневая операция `c = a + b` развернется в последовательность байт-кода: загрузить значение `a`, загрузить `b`, выполнить бинарное сложение, сохранить результат в `c`. Рассмотрим конкретный пример и скомпилируем пару строк в байт-код вручную:

```
code_obj = compile("a = 1\nb = 2\nc = a + b", "<string>", "exec")
import dis
dis.dis(code_obj)
```

Вывод (для CPython 3.11) будет примерно таким:

```
0 RESUME                   0
2 LOAD_CONST               0 (1)
4 STORE_NAME               0 (a)
6 LOAD_CONST               1 (2)
8 STORE_NAME               1 (b)
10 LOAD_NAME                0 (a)
12 LOAD_NAME                1 (b)
14 BINARY_OP                0 (+)
18 STORE_NAME               2 (c)
20 LOAD_CONST                2 (None)
22 RETURN_VALUE
```

Здесь видно, как каждая строка Python разбилась на несколько инструкций: `LOAD_CONST 1` кладет константу `1` на стек, `STORE_NAME a` присваивает ее имени `a`; затем аналогично для `b`; затем `LOAD_NAME a`, `LOAD_NAME b` загружают оба значения, `BINARY_OP +` (аналогично старому `BINARY_ADD`) складывает их, результат помещается на стек, откуда `STORE_NAME c` сохраняет его в переменную `c`. Последние инструкции загружают `None` и возвращают его (завершение блока кода). Специальная инструкция `RESUME 0` отмечает начало выполнения нового фрейма (о ней подробнее в разделе про PVM и фреймы).

Таким образом, байт-код – это своего рода “ассемблер” для **Python Virtual Machine**, гораздо более компактный и быстрый для интерпретации, чем исходный высокоуровневый код [9]. CPython сохраняет сгенерированный байт-код в объекте типа **code object** (тип `code` в Python). Этот объект содержит: массив инструкций байт-кода, константы, имена переменных, количество локальных переменных, исходный filename/номер строки для отладки и другую служебную информацию. Code object можно получить и программно: например, функция `f.__code__` дает код-объект функции `f`, а встроенная функция `compile()` возвращает код-объект из строки исходника.

- **Сохранение байт-кода (.pyc)** – при импорте модулей Python байт-код автоматически кешируется на диск, чтобы повторно не тратить время на компиляцию при следующем запуске. Скомпилированные файлы имеют расширение `.pyc` и хранятся в директории `__pycache__` рядом с исходниками [10] [11]. Например, при первом импорте модуля `example.py` Python создаст `__pycache__/example.cpython-311.pyc` (где `311` – версия интерпретатора). В `.pyc` файле содержится "сырый" байт-код (сериализованный объект code) и немного служебной информации – **magic number** версии Python и метка времени или хеш исходного файла для проверки актуальности. В дальнейшем, если исходник не изменялся, интерпретатор может загрузить байт-код из `__pycache__`, минуя этапы лексера/парсера [12]. Это ускоряет запуск модулей, особенно больших, но **не** влияет на скорость выполнения самого байт-кода [13]. Для явной компиляции файла в `.pyc` можно использовать модуль `py_compile`. Например:

```
import py_compile
py_compile.compile("script.py", cfile="script.pyc")
```

Это сгенерирует байт-код для `script.py` и сохранит его в указанном файле. Считать обратно байт-код можно модулем `marshal`: он используется внутренне для записи/чтения байт-кода. Модуль `marshal` умеет преобразовывать объекты Python (включая code objects) в байтовую строку. **Важно:** формат байт-кода не является частью спецификации языка и может меняться между версиями CPython, поэтому файлы `.pyc`, сгенерированные в одной версии, не гарантированно работают в другой.

- **Исполнение байт-кода** – выполняется интерпретатором Python Virtual Machine (см. следующий раздел). На этапе компиляции генерация `.pyc` может быть пропущена, если мы просто запускаем скрипт напрямую – в этом случае байт-код хранится только в памяти, и после завершения программы он удаляется. Но при многоразовых запусках через импорты механизм `__pycache__` обеспечивает кэширование. Отметим, что Python-приложение может ускоряться благодаря кешу байт-кода только в части времени загрузки (парсинга), но не в самом алгоритме, т.к. байт-код интерпретируется заново при каждом запуске.

Каждый из упомянутых этапов можно проанализировать и программно: модуль `tokenize` даст доступ к токенам, модуль `ast` – к синтаксическому дереву, модуль `dis` – к инструкциям байт-кода, `py_compile` сгенерирует pyc-файл, а `marshal` позволит сериализовать/десериализовать code object. Для статического анализа кода также полезны инструменты вроде `ast.dump()` для визуализации AST и `dis.Bytecode` для итерации по инструкциям. Завершая обзор: **исходный код .py → [lexer] → токены → [parser] → AST → [компилятор] → байт-код (.pyc) → [интерпретатор] → выполнение.** В конечном счете, интерпретатор приводит к выполнению эквивалентных машинных инструкций на CPU, но опирается на промежуточный слой байт-кода для переносимости и гибкости.

Вывод: трансляция Python-кода включает компиляцию в байт-код, благодаря чему интерпретатор может эффективно и платформенно-независимо исполнять программы. Знание этих этапов позволяет разработчику использовать встроенные модули (`tokenize`, `ast`, `dis` и др.) для анализа и оптимизации Python-кода.

3. Python: компиляция или интерпретация?

Является ли Python компилируемым или интерпретируемым языком? Правильный ответ: **и тем, и другим.** Стандартная реализация CPython выполняет *двухступенчатый* процесс: сначала происходит компиляция исходного кода в байт-код, а затем этот байт-код интерпретируется виртуальной машиной [14]. Однако этот процесс полностью автоматизирован и скрыт от программиста, поэтому Python обычно называют интерпретируемым языком. В отличие от C/C++ у Python нет отдельного шага ручной компиляции в машинный код – за нас это делает интерпретатор во время исполнения программы. В итоге, **CPython компилирует .py в .pyc (байт-код), затем интерпретирует .pyc на PVM** [15].

Практическое доказательство этого: при первом импорте модуля создается файл с байт-кодом в `__pycache__` (расширение .pyc). Например, если создать файл `hello.py` с простым кодом и импортировать его, появится `__pycache__/hello.cpython-311.pyc` – скомпилированный байт-код [16] [17]. Если запустить скрипт напрямую, Python тоже компилирует его в байт-код, но по

умолчанию не сохраняет его на диск (если только скрипт не импортируется где-то). Таким образом, **Python-программа прежде компилируется, затем выполняется**. С точки зрения спецификации языка, способ исполнения не фиксирован – другие реализации Python могут использовать иные стратегии (например, JIT-компиляцию), но принципиально везде необходим некоторый этап трансляции перед выполнением.

Отличие байт-кода от машинного кода. Байт-код – платформо-независимый двоичный формат инструкций, понятный виртуальной машине, а **машинный код** – это двоичные инструкции, непосредственно выполняемые процессором [18]. Машинный код специфичен для архитектуры CPU (x86, ARM и т.д.) и упакован в исполняемый файл (EXE, ELF и пр.). Байт-код же портируем: один и тот же .pyc можно исполнить на любой платформе с совместимым интерпретатором Python. Компиляция в байт-код, как правило, происходит быстрее и проще, чем полноценная компиляция в машинный код, и генерирует меньше кода [18] [19]. Зато байт-код не может выполняться без интерпретатора – сам по себе CPU не "понимает" эти инструкции, нужен слой PVM, который будет их анализировать и выполнять. Машинный код, наоборот, выполняется напрямую, зато требует отдельной генерации под каждую платформу. Если провести аналогию: байт-код – это как промежуточный псевдо-ассемблер, а интерпретатор играет роль "процессора" для него.

Как просмотреть байт-код? Разработчикам Python предоставлены инструменты для анализа байт-кода. Основной – модуль `dis` (дисассемблер). С его помощью можно, например, получить байт-код функции:

```
import dis
def greet(name):
    return f"Hello, {name}!"
dis.dis(greet)
```

Выше мы использовали `dis` уже, и он выводит список инструкций с их адресами, аргументами и комментариями (имена констант, переменных). Также можно скомпилировать выражение через `compile()` и дизассемблировать результат. Еще один способ – открыть `.pyc` файл. Он бинарный, но начинается с 16-байтного заголовка (версия, timestamp/хеш) и далее содержит сериализованный через `marshal` объект кода. С помощью модуля `marshal` и `types` можно загрузить байт-код, но гораздо удобнее использовать `dis` и высокоуровневое API. Например, `dis.hasnative(co)` проверит, содержит ли код объект `co` "нативный" код (актуально для JIT, см. ниже), а `co.co_code` вернет строчку байт-кода. В обычной практике знание конкретных опкодов не требуется, но при отладке и понимании производительности `dis` – незаменимый инструмент.

Python и компиляция в C (Cython). Хотя CPython сам по себе не компилирует код в машинные инструкции, существуют способы ускорить Python-программы путем статической компиляции. Один из популярных – проект **Cython**. Cython – это отдельный компилятор (и язык-наследник Python), который берет Python-подобный код и преобразует его в оптимизированный C-код, далее компилируемый в машинный код с помощью обычного C-компилятора. Результатом может быть *расширение* Python (динамическая библиотека .so/.pyd), которую можно импортировать в Python как модуль, либо самостоятельный исполняемый файл. Обычно Cython используется для ускорения критичных участков: разработчик добавляет статические типы для переменных, функций (`cdef int a` и т.п.), компилирует модуль – и получает прирост быстродействия за счет того, что часть работы выполняется на уровне скомпилированного Си без накладных расходов

интерпретатора. Пример минимального цикла компиляции Cython: допустим, у нас есть `hello.py` с кодом `print("Hello World!")`. Чтобы сделать из него исполняемый файл через Cython:

1. Установить Cython (`pip install cython`).
2. Сгенерировать C-код из Python: `cython --embed -3 hello.py -o hello.c` (опция `--embed` просит сгенерировать C-код, **включающий запуск интерпретатора**, для standalone программы [20] [21]).
3. Откомпилировать `hello.c` с включением Python-интерпретера: `gcc hello.c -o hello $(python3-config --includes --ldflags)`. Эта команда слинкует статически интерпретатор.
4. В итоге получится исполняемый файл `hello` (ELF-бинарь в Linux) [22], который можно запускать независимо от `python`.

Для библиотек (расширений) процесс проще: пишется файл `module.pyx` (Cython-расширение), затем либо через `distutils/setuptools` (setup.py) или команду `cythonize` генерируется .c и компилируется в .so, который подключается `import`-ом. **Таким образом, Cython позволяет превратить Python-код в скомпилированный двоичный модуль**, достигнув ускорения. Однако следует помнить, что Cython – не чистый Python: требуется добавлять аннотации типов, и полученный код уже не переносим между платформами без перекомпиляции (в отличие от байт-кода .pyc). Тем не менее, для задач, критичных к скорости, Cython – мощный инструмент, позволяющий получить производительность, близкую к C, сохраняя при этом синтаксис похожим на Python.

Архитектура байт-кода и зависимости от реализации. Важно понимать, что **байт-код – внутренняя деталь реализации, а не языковой стандарт**. CPython определяет один формат байт-кода, который может меняться от версии к версии. Другие реализации Python вовсе могут не использовать CPython-байт-код. Например, Jython компилирует Python-программы в байт-код **JVM (Java Virtual Machine)** – фактически в .class файлы Java; IronPython компилировал в байт-код .NET (IL); PyPy выполняет *трассировку* исполнения и компилирует часто выполняемые части в машинный код на лету (JIT), а исходный Python-байт-код у PyPy – свой собственный для интерпретатора RPython. GraalPy (Python на GraalVM) тоже применяет собственные оптимизации. То есть, **понятие "Python-код" на уровне байт-кода различно в CPython, PyPy, Jython и т.д.**, хотя на уровне исходного языка все совместимы (следуют спецификации Python). Даже внутри CPython версия 3.11 внесла существенные изменения в байт-код: например, отказ от инструкции `LOAD_FAST 0` в пользу `RESUME` + `LOAD_FAST`, объединение множества бинарных операций в одну `BINARY_OP` с операндом (для ускорения за счет более компактного диспетчера) и пр. Разработчикам, пишущим под конкретную реализацию, стоит знать эти нюансы, но в целом при переносе кода между разными Python-реализациями эти детали скрыты.

Вывод: Python сочетает в себе черты интерпретируемых и компилируемых языков. Код сначала компилируется в переносимый байт-код, а затем исполняется виртуальной машиной. Это дает баланс между удобством (отсутствие ручной компиляции) и эффективностью (кеширование и оптимизация байт-кода). При необходимости критичные части могут быть переведены в машинный код (через Cython или альтернативные реализации), но основной подход CPython – интерпретация байт-кода на лету.

4. PVM – Python Virtual Machine

Что такое PVM? Python Virtual Machine – это **виртуальная машина Python**, основной исполнительный движок, который берет байт-код и выполняет его. Проще говоря, PVM – это **интерпретатор байт-кода**. В CPython PVM реализован на C внутри файла `ceval.c` и других частей, и представляет цикл, который читает из массива байт-кода инструкцию за инструкцией и выполняет соответствующие операции. Именно PVM ответственна за то, чтобы инструкция `BINARY_OP +` привела к сложению двух чисел, `STORE_NAME` – к сохранению объекта в таблице переменных, и т.д.

Исполнение кода на PVM. Когда вы запускаете программу, компилятор сгенерировал байт-код (в памяти или взяв из `__pycache__`), после чего PVM берется интерпретировать этот байт-код. Это означает, что PVM **проводит соответствие между каждой инструкцией байт-кода и набором низкоуровневых операций на реальной машине** (обычно – вызовами функций на C, выполняющими то или иное действие). Например, встретив `LOAD_CONST 5`, PVM обращается к данному код-объекту, берет пятый константный объект и кладет его на стек (набор регистров/ячеек в памяти, который PVM использует как рабочую область). Если далее идет `BINARY_OP +`, PVM вытащит два верхних значения из своего стека, вызовет соответствующую C-функцию, реализующую сложение объектов Python (учитывая их типы – int, str и т.д.), затем результат (новый PyObject) положит обратно на стек. Этот цикл повторяется до тех пор, пока интерпретатор не дойдет до `RETURN_VALUE` или другой финальной инструкции.

Важно отметить, что PVM *не превращает весь байт-код сразу в машинный код*, а **исполняет его построчно (по одной инструкции)** [23]. Фактически, PVM – это большой `switch`/`loop` (в терминах C) либо более оптимизированный механизм (в Python 3.11+ введены т.н. “adaptive interpreter” – адаптивные инструкции, но суть та же) для обработки опкодов. Таким образом достигается платформенная независимость – один и тот же байт-код может выполняться на любом процессоре, для которого есть реализация PVM. Недостаток – интерпретация имеет расходы: каждый опкод требует определенной логики на C, проверки типов, возможны непрямые переходы (что хуже для современных процессоров). Тем не менее, **PVM предоставляет гибкость**: она может вести отладку, сборку мусора, отслеживать счетчики выполнения, что сложно при прямом машинном коде.

Управление памятью и GC. PVM также отвечает за **управление памятью** в Python. В CPython используется комбинация **подсчета ссылок** и периодического **сборщика мусора**. Каждый объект Python в куче (heap) имеет счетчик ссылок (ob_refcnt). Когда вы присваиваете объект переменной или помещаете в список, счетчик увеличивается; когда ссылку удаляют (например, переменная уходит из области видимости или элемент списка удаляется), счетчик уменьшается. PVM следит за этими счетчиками постоянно во время исполнения: как только ob_refcnt объекта становится 0, PVM сразу освобождает память под объект (вызывает деинициализатор, освобождает его `PyObject*`) – это операция **освобождения памяти (deallocation)**. Таким образом, большая часть мелких объектов убирается сразу в момент, когда на них больше нет ссылок. Дополнительно, поскольку подсчет ссылок не способен обнаружить **цикл** объектов (когда объекты ссылаются друг на друга и недостижимы снаружи), в PVM встроен *циклический GC*: периодически (или при переполнении определенных поколений) запускается сборщик, который строит граф объектов и выявляет изолированные циклы, после чего освобождает их. Встроенный модуль `gc` позволяет настраивать этот процесс (например, `gc.disable()` отключит циклический сборщик, а `gc.collect()` вручную инициирует сбор).

Стоит упомянуть, что **управление другими ресурсами** (не памятью) – например, файлами, сетевыми соединениями – также во многом опирается на механизмы PVM. Благодаря подсчету ссылок, CPython чаще всего освобождает объект файл (и, следовательно, закрывает дескриптор) *сразу* при выходе переменной из области видимости. Это поведение можно считать частью "ресурсного менеджмента" PVM. Но для надежности Python предоставляет **контекстные менеджеры** (`with`-блоки) и инструкцию `finally` – они позволяют гарантированно освободить ресурс в определенном месте. Например, `with open("data.txt") as f:` откроет файл и автоматически закроет (f.close()) по выходу из блока, даже если произошло исключение. Внутренне это реализовано за счет вызова специальных методов объектов и встроенных инструкций `WITH_CLEANUP_START/FINISH` в байт-коде. Таким образом, PVM вкупе с протоколом контекст-менеджеров обеспечивает детерминированное освобождение внешних ресурсов.

PVM и GIL. Несмотря что вопрос отдельно не акцентирован на GIL, при обсуждении PVM важно знать: **Global Interpreter Lock (глобальная блокировка интерпретатора)** – механизм CPython, который *гарантирует, что одновременно выполняется байт-код только в одном потоке*. PVM всегда работает с учетом GIL: перед выполнением очередной порции байт-кода поток должен захватить GIL. Это упрощает написание интерпретатора и управление памятью (не нужны мьютексы на каждый объект), но мешает полноценно параллелить Python-вычисления на нескольких ядрах. В альтернативных реализациях (Jython, IronPython, PyPy) GIL может отсутствовать или работать иначе, но CPython до версии 3.12 включительно имеет GIL. Вопрос GIL актуален и упоминается в секции про NoGIL ниже.

Как PVM связан с фреймами исполнения? Когда PVM исполняет байт-код, он делает это в контексте **фрейма** (frame). **Фрейм** – это структура данных, представляющая выполнение конкретной функции или модуля. В CPython фрейм – объект `PyFrameObject`, содержащий: указатель на код-объект (того же типа, что хранит байт-код), указатели на глобальные и локальные переменные (для функции – массив или кортеж локальных переменных, а также стек для операндов), ссылку на предыдущий (вызывающий) фрейм и несколько полей для счетчиков, флагов и т.п. Каждый раз, когда вызывается функция, PVM создает новый фрейм и начинает исполнять байт-код этой функции. Когда функция завершается (возвращает значение или выбрасывает исключение), соответствующий фрейм уничтожается, а управление возвращается во внешний фрейм. Эти фреймы организованы в **стек вызовов** (call stack): текущий выполняемый фрейм – верхушка стека. При рекурсии вы будете иметь цепочку из многих фреймов одного и того же кода, каждый со своим набором локальных переменных. В Python можно introspect'ить стек вызовов (например, модуль `inspect` или `traceback` позволяют пройтись по цепочке фреймов при ошибке).

Инструкция RESUME и изменения в Python 3.11+. В Python 3.11 внутренняя реализация фреймов и байт-кода претерпела оптимизации. Появилась специальная инструкция `RESUME`, которая явно помечает начало выполнения нового фрейма или возобновление генератора. Это нововведение связано с механизмом адаптивного интерпретатора (PEP 659) и оптимизацией обработки исключений (zero-cost exceptions). Инструкция `RESUME` сама по себе ничего не выполняет (это **no-op**) и служит для внутренних целей – например, чтобы при запуске функции выполнить определенные проверки один раз, и чтобы средства трассировки понимали, откуда реально начинается пользовательский код. Как пояснил один из разработчиков CPython, *"инструкция RESUME сообщает интерпретатору, что начинается (или продолжается) выполнение нового фрейма; все, что идет в байт-коде до первого RESUME, относится к настройке фрейма и может игнорироваться при анализе"* [24]. В выходе `dis` эта инструкция имеет тот же номер строки, что и определение функции, а основной код функции идет после нее. Например:

```
def add(a, b):
    return a + b

import dis
dis.dis(add)
```

выводит:

```
0 RESUME                   0
2 LOAD_FAST                0 (a)
4 LOAD_FAST                1 (b)
6 BINARY_OP                0 (+)
10 RETURN_VALUE
```

Здесь байт-код `add` начинается с `RESUME 0`, за которым следуют инструкции для сложения. Если смотреть на старые версии (до 3.11), аналогичный код имел бы старт сразу с `LOAD_FAST`. Таким образом, `RESUME` ввели для удобства и оптимизации: он помогает синхронизировать состояние интерпретатора при входе в функцию, особенно с учетом новых возможностей специализации. Для обычного программиста это изменение почти прозрачно, но его следует помнить, разбирая байт-код или пиша инструменты, зависящие от номеров строк.

PVM и сборщик мусора (GC). Как отмечалось, PVM напрямую осуществляет подсчет ссылок. Циклический GC запускается не на каждой инструкции, а периодически – обычно когда создается достаточно много новых объектов (threshold). Это происходит внутри PVM – например, после завершения некоторой функции или аллокации серии объектов может запускаться `collect`. В Python 3.11 изменение организации фрейма позволило сделать вызовы сборщика еще более “ленивыми”, чтобы снизить накладные расходы. PVM отвечает за вызов финализаторов объектов (метод `__del__`), когда объект наконец собирается. Также PVM освобождает память обратно вAllocator/Python’s memory manager или OS через pymalloc/allocators.

Управление ОС ресурсами. PVM неявно вмешивается и в управление некоторыми системными ресурсами: например, при завершении программы PVM гарантированно вызывает `PyFinalize`, закрывая все оставшиеся открытые файлы, сокеты, очищая буферы вывода и т.д. Также модуль `atexit` позволяет зарегистрировать функции, которые PVM вызовет перед остановкой интерпретатора. В многопоточных прогаммах PVM управляет жизненным циклом потоков (например, при вызове `threading.Thread`, каждый поток запускает свой экземпляр PVM loop, но GIL сериализует их исполнение). При завершении потока PVM очищает все локальные данные потока.

Подводя итог: **Python Virtual Machine** – сердце интерпретатора, исполняющее байт-код и управляющее всем, что происходит во время выполнения программы: от вычислений и вызовов функций до распределения памяти и освобождения ресурсов. Благодаря PVM Python-код “абстрагируется” от аппаратуры – можно написать программу на Windows x86 или ARM Linux, и она будет выполняться на обеих платформах, поскольку PVM возьмет на себя подробности выполнения. С другой стороны, разработчику важно понимать, что из-за особенностей PVM (интерпретация, GIL, сборка мусора) поведение и производительность Python могут отличаться от низкоуровневых языков. **Вывод:** PVM – это промежуточный слой между вашим кодом и железом, реализующий модель исполнения Python. Он предоставляет удобство (автоматическое

управление памятью, переносимость), но накладывает некоторые ограничения (однопоточность в CPython, относительная медлительность интерпретации).

5. Just-in-Time Compilation (JIT)

Зачем нужен JIT? *Just-in-Time компиляция* – это технология ускорения программ, которая комбинирует интерпретацию и традиционную компиляцию. Идея JIT: во время выполнения программы обнаруживать наиболее часто исполняющиеся участки кода (“**горячие точки**”, hot spots) и **динамически компилировать** их в машинный код, чтобы дальнейшие выполнения шли быстрее [25] [26]. Тем самым JIT-подход пытается взять лучшее от двух миров: как у интерпретаторов, мы получаем гибкость и способность работать с динамическими языками (можно выполнять код «как есть» сразу), а как у компиляторов – высокую скорость на критичных частях за счет выполнения непосредственно машинных инструкций. JIT-компиляция особенно эффективна для динамических языков программирования, т.к. позволяет проводить оптимизации с учётом профиля выполнения (runtime information) и обходить ограничения статической компиляции [27].

Как работает JIT и основные этапы. JIT-компилятор обычно встроен в интерпретатор или виртуальную машину. Его работа проходит несколько этапов:
1. **Профилирование исполнения:** сначала код исполняется интерпретатором обычным образом, но при этом система **собирает статистику** – считает, сколько раз выполняется каждая функция или цикл, какие типы данных встречаются и т.п. Выявляются **часто выполняемые участки** – те самые hot spots. Например, метод Java HotSpot помечает метод “горячим”, если он выполнен >10_000 раз [28]. В PyPy аналогично трассируются часто повторяющиеся циклы.
2. **Компиляция горячих участков:** когда некий фрагмент кода превышает порог частоты использования, JIT-транслятор вступает в дело. Он берет этот участок (например, тело часто вызываемой функции или цикл) и компилирует **прямо во время исполнения** в нативный машинный код. Компиляция включает классические оптимизации (распознание типов, разворачивание констант, устранение ненужных проверок, инлайн функций и др.), но выполняется в ограниченном *runtime-бюджете* времени – ведь программа продолжает работать. Поэтому JIT часто применяется поэтапно: сначала быстрая базовая компиляция (“warm-up”), позже при наличии времени – более глубокие оптимизации (в Java HotSpot есть уровни C1 (быстрый, базовые оптимизации) и C2 (медленный, агрессивные оптимизации)).
3. **Кэширование и исполнение скомпилированного кода:** полученный машинный код сохраняется в памяти и с этого момента при входе в данный участок управления вместо интерпретации байт-кода выполняется уже машинный код напрямую на CPU. Это дает значительный прирост скорости – в идеале, приближаясь к скорости программ на C. Код хранится в специальной области (JIT code cache) и может переиспользоваться многократно без повторной компиляции [29] [30]. Если впоследствии профиль исполнения изменится (например, типы аргументов функции поменяются), JIT может при необходимости **деоптимизировать** код (вернуться к интерпретации или перекомпилировать с учётом новых данных).
4. **Продолжение профилирования:** JIT-системы нередко продолжают мониторить даже скомпилированный код – вдруг появятся еще более горячие пути или, наоборот, какой-то заранее скомпилированный участок больше не используется. Это позволяет, например, выгружать из памяти неиспользуемый машинный код (чтобы не занимал место) [30] или повторно компилировать критичные части с еще большими оптимизациями (**адаптивная оптимизация** [31]).

Hot spots и связь с кэшированием. Понятие “горячих точек” означает, что **большая доля времени программы тратится на относительно небольшие фрагменты кода** (например, внутренность частого цикла). JIT нацелен именно на них: нет смысла тратить ресурсы на

компиляцию кода, который исполняется разово. Таким образом, JIT улучшает **локальность исполнения**: откомпилированный код выполняется многократно, и благодаря этому он остается в процессорных кэшах (L1/L2) дольше, что дополнительно ускоряет выполнение. Здесь возникает интересная связь: JIT не только сам по себе ускоряет программу, но и **способствует тому, чтобы "горячий" машинный код жился в CPU cache**. Кроме того, сам JIT-компилятор хранит сгенерированный код в специальном **кэше кода** (области памяти). Мы видели исторический пример: в Smalltalk-80 JIT-компилятор уже умел кешировать результат компиляции "по требованию" – и удалять часть машинного кода, когда память заканчивалась, с возможностью восстановить его при повторной надобности [30]. Современные JIT (Java, V8 для JS, PyPy для Python) также реализуют стратегии хранения/удаления JIT-кода, чтобы балансировать между использованием памяти и скоростью.

Недостатки JIT. За повышение скорости приходится платить:
- **Время компиляции при запуске (warm-up time):** JIT-компилятор выполняется во время работы программы, поэтому при старте наблюдается задержка на профилирование и первую компиляцию горячих участков. Простыми словами, первые итерации могут работать даже медленнее интерпретируемого кода. Со временем JIT "разогревается" и превосходит интерпретатор. В краткоживущих скриптах JIT может не успеть оправдать себя. Разработчики JIT ищут компромисс между качеством оптимизаций и задержкой запуска [32] [33]. В некоторых платформах (Java) предлагают "режимы" JIT: быстрый/клиентский (минимум оптимизаций, чтобы скорее стартовать) и медленный/серверный (агрессивные оптимизации, лучше производительность, но дольше старт) [34].
- **Повышенный расход памяти:** JIT-компиляция хранит сгенерированный машинный код в памяти. В отличие от интерпретатора, которому достаточно хранить байт-код и интерпретатор loop (размер фиксирован), JIT может породить значительный объем native-кода, дублируя, например, одну и ту же функцию в нескольких специализированных версиях (для разных типов или ситуаций). Это увеличивает потребление памяти (как ОЗУ, так и кэш-памяти CPU). Как отмечено в определении, JIT достигает скорости **ценой увеличения потребления памяти для хранения результатов компиляции** [35].
- **Сложность реализации и потенциальные баги:** JIT-компилятор – очень сложный компонент (фактически, runtime-компилятор со множеством оптимизационных фаз). Это повышает вероятность ошибок и уязвимостей. Также JIT-движки затрудняют отладку программ (код "меняется" на лету) и профилирование – профили CPU могут показывать загадочные фрагменты JIT-кода без связи с исходными строками.
- **Неуниверсальность оптимизаций:** JIT хорошо ускоряет вычислительно интенсивный код, где много повторяющихся операций. Но если программа в основном ввод-выводовая (IO-bound) или однократного действия, выгода может быть минимальной. К тому же JIT не устраняет фундаментальных ограничений вроде глобальной блокировки (GIL) – например, PyPy с JIT ускоряет вычисления внутри одного потока, но не поможет задействовать несколько ядер, если есть GIL (в PyPy GIL тоже есть).
- **Непредсказуемость таймингов:** В системах реального времени JIT не всегда применим, т.к. в произвольный момент может вклиниться пауза на компиляцию. Хотя существуют **профилирующие JIT** (tiered JIT), которые стараются распределять нагрузку, гарантий жёсткого времени нет.

Несмотря на эти минусы, **в динамических языках JIT обычно существенно повышает производительность** на длинных запусках. В JavaScript-движках (Chrome V8, Mozilla SpiderMonkey) JIT-компиляция позволила достигнуть скоростей, делающих JS пригодным даже для тяжелых приложений. В мире Python наиболее известна JIT-реализация – **PyPy**, о ней далее. Также проекты Pyston, Numba (JIT для научных вычислений), LuaJIT в мире Lua – показывают огромные ускорения на определенных задачах. JIT-компиляция – активная область исследований; новые

подходы (например, **репозитарная JIT/AOT**, как в .NET Native, или специализированные JIT под GPU) стараются сгладить упомянутые недостатки.

Вывод: Just-in-Time компиляция динамически ускоряет программу, компилируя часто исполняемый код в машинный. Она обеспечивает компромисс: быстродействие близкое к скомпилированному коду при сохранении интерактивности интерпретатора. Однако JIT не бесплатен – он усложняет систему и требует дополнительных ресурсов (времени и памяти), поэтому эффективность JIT проявляется на длительных задачах с повторяющимися вычислениями.

6. Реализации Python: обзор и сравнение

Помимо стандартного **CPython**, существуют различные реализации Python, нацеленные на улучшение тех или иных аспектов (скорость, взаимодействие с другими платформами, ограниченные среды и т.д.). Рассмотрим основные из них и их особенности:

- **CPython** – **каноничная реализация** на языке C, разрабатываемая python.org. Это та самая версия, которую вы получаете, скачав Python с официального сайта. CPython следует всем стандартам языка и на сегодняшний день является наиболее совместимой и полной реализацией. Отличительные черты: *интерпретатор байт-кода с GIL*, используемый алгоритм сборки мусора – подсчет ссылок + поколенческий GC, поддержка всех расширений на C (C-API). CPython обычно не самый быстрый по производительности, но наиболее надежный и совместимый. Именно CPython мы имеем в виду, говоря просто “Python”, если не уточнено обратное. Большинство библиотек написаны с расчетом на CPython C-API.

- **PyPy** – альтернативная реализация, написанная на подмножестве Python (RPython), с интегрированным **JIT-компилятором**. PyPy стремится к максимальной скорости выполнения чисто-питонячего кода. На типичных алгоритмических задачах PyPy может работать в 2-5 раз быстрее CPython (а в некоторых случаях и в 10+ раз), благодаря тому, что после “прогрева” компилирует горячие участки в машинный код. PyPy поддерживает большую часть стандартных библиотек Pure Python, но совместимость с расширениями на C ограничена (он не использует CPython API напрямую; для многих популярных библиотек есть специальные версии или cffi-обертки). PyPy славится быстрой работой циклов и вычислений, однако запуск (старт) у него медленнее. PyPy – отличный выбор для долгоживущих процессов на Python, где важна скорость (например, серверные приложения). Глобальная блокировка (GIL) в PyPy тоже присутствует, поэтому он, как и CPython, не ускоряет CPU-bound задачи многопоточностью.

- **Jython** – реализация Python для **Java Virtual Machine (JVM)**. Jython компилирует исходный Python в байт-код Java (байты .class), который выполняется на JVM. Благодаря этому, Jython может свободно использовать все библиотеки Java: Python-код может импортировать Java-классы, и наоборот, Java-код может вызывать Jython-скрипты. Это обеспечивает отличную интеграцию в Java-экосистему. Однако версия языка Jython несколько отстает (последний стабильный Jython соответствует Python 2.7; проект Jython 3 находится в разработке). Также Jython не поддерживает библиотеки, написанные на C для CPython (например, NumPy) – ему нужны Java-аналоги. По скорости Jython иногда превосходит CPython за счет оптимизирующей JVM (JIT HotSpot), особенно на долгих вычислениях, и при этом у Jython **нет GIL** (потоки маппятся на потоки JVM, а там – на системные потоки, поэтому истинная

параллельность возможна). Тем не менее, Jython сейчас менее популярен из-за стагнации развития и ограничения на версию языка.

- **IronPython** – аналог Jython, но для платформы **.NET/CLR (Common Language Runtime)**. Писался на C# и тесно интегрирован с .NET: Python-код компилируется в MSIL (Intermediate Language) и работает на CLR, что позволяет использовать библиотеки .NET из Python и наоборот. IronPython был популярен в середине 2000-х, особенно в связке с WPF, Silverlight (для написания скриптов под .NET). Как и Jython, IronPython страдает от отставания по версии (последние релизы соответствуют Python 2.7, хотя есть активность по Python 3.x в IronPython 3). IronPython снимает проблему GIL (CLR управляет потоками), но не совместим с CPython C-расширениями. В современных Windows/.NET экосистемах IronPython применяют редко, уступив место Python.NET (встраивание CPython в .NET-процесс) или опять же использованию CPython + interop.

- **GraalPy (GraalPython)** – новая высокопроизводительная реализация на основе **GraalVM** (проект компании Oracle). GraalPy (ранее называвшийся GraalPython) целится быть полностью совместимым с Python 3.11+ и при этом обеспечивать высокую скорость за счет оптимизаций GraalVM. Internals: GraalPy строится на фреймворке **Truffle**, который позволяет реализовать язык с автоматическим получением JIT-компиляции и интеропа с другими языками Graal (Java, JavaScript, etc.). GraalPy умеет **комбинировать Python и Java** – можно вызывать Java-классы из Python и наоборот, с очень низкими накладными расходами (в GraalVM разные языки бегут в единой runtime). Текущие бенчмарки показывают, что GraalPy значительно быстрее CPython на ряде задач – в разы (иногда упоминается ~5-8x быстрее, а на некоторых тестах и ~2x быстрее PyPy [36]). Более того, GraalPy может компилировать Python-приложения в **самостоятельные бинарники** благодаря Native Image (подобно тому, как это делается для Java приложений с GraalVM Native). Ограничения: GraalPy пока не поддерживает C-расширения CPython напрямую (хотя есть экспериментальная поддержка через эмуляцию CPython C-API), размер runtime довольно большой (весь GraalVM), и поведение может отличаться в деталях (например, у него другой сборщик мусора – от JVM). Но GraalPy – очень перспективный проект для тех, кому нужна скорость и интеграция с Java. (Применение: например, высоконагруженные сервисы, где часть логики удобно на Python, а развертывать хочется как единый Java-native образ).

- **Python NoGIL (Nogil/Сам Гросс)** – это не отдельная с нуля реализация, а **форк CPython**, модифицирующий интерпретатор для работы **без глобального интерпретера**. Создан разработчиком Sam Gross, он убирает GIL и защищает объекты другими механизмами (например, мелкозернистыми локами, атомиками). Цель – получить **полноценный многопоточный Python**, способный параллельно исполнять код на нескольких ядрах без уступки GIL. Форк Sam Gross'a показал, что это возможно, хотя и с ценой: в однопоточном режиме такой интерпретатор медленнее (10-20% замедление на ряде задач из-за накладных расходов синхронизации). Однако на многопоточных CPU-bound задачах выигрыш огромен (почти линейный рост с числом потоков). Проект вызвал большой интерес, и в 2023 году PEP 703 ("GIL: сделать опциональным") был одобрен – планируется слияние NoGIL или его идей в мейнлайн CPython, возможно к версии 3.13. Таким образом, **Nogil** – это путь к Python с реальной параллельностью. На сегодняшний день он остается экспериментальной веткой; многие расширения, зависящие от тонкостей GIL, нужно адаптировать. Но стоит ожидать, что через пару лет CPython сможет запускаться в режиме без GIL.

- **Pyodide** – порт Python (точнее, CPython) для **WebAssembly**, позволяющий запускать Python-код прямо в браузере. Pyodide берет исходники CPython, компилирует их с помощью Emscripten в WebAssembly-модуль, дополняет окружением (std libs) и предоставляет API для запуска Python из JavaScript. Проще говоря, Pyodide – это **Python в браузере**, работающий в sandboxe без доступа к файловой системе (кроме виртуальной FS) и с ограничениями браузера. Огромный плюс: Pyodide поддерживает огромное количество пакетов из Python-экосистемы, включая NumPy, Pandas, SciPy – они тоже скомпилированы в WebAssembly. Таким образом, можно писать код на Python и выполнять его в веб-странице, взаимодействуя с JS. Pyodide используется внутри проекта Firefox Iodide, а также лег в основу проекта **PyScript** (который позволяет вставлять код Python прямо в HTML). По производительности Pyodide уступает нативному CPython (WebAssembly накладывает overhead, особенно для вычислений, и пока нет JIT), но с развитием WebAssembly (например, появление wasm-thread и wasm-gc) это улучшается. Главное – Pyodide открыл двери для запуска существующих научных Python-библиотек в браузере без переписывания на JS. Для взаимодействия с JS объектами Pyodide предоставляет прокси-объекты; можно вызывать JS-функции из Python и наоборот.

- **Brython** – еще один способ запускать Python в браузере, но подход иной: Brython – это **транспилер Python-в-JavaScript**. Он написан на чистом JS и загружается в браузере, после чего берет код Python, написанный в скриптовых тегах или загруженный динамически, и **конвертирует его на лету в эквивалентный JavaScript**, который и исполняется движком браузера. Название расшифровывается как “Browser Python”. Brython ориентирован на замену JavaScript в веб-разработке: позволяет писать клиентские скрипты на Python, взаимодействовать с DOM (у Brython есть обертки, например, `document` доступен как `DOMDocument` в Python-коде). Преимущество – простота использования (достаточно подключить brython.js), полный доступ к браузерным API через Python. Недостаток – производительность определяется тем, насколько эффективен сгенерированный JS; а также несовместимость с многими привычными библиотеками CPython (нет доступа к нативным модулям). Brython поддерживает только чистый Python на уровне синтаксиса (и не все, например, `asyncio` может не работать, хотя есть своя event loop интеграция с JS). В общем, Brython – хороший инструмент для тех, кто хочет писать небольшие веб-скрипты на Python вместо JS, но он не предназначен для тяжелых вычислений.

- **MicroPython** – ультра-легковесная реализация Python 3, предназначенная для **микроконтроллеров и встроенных систем**. MicroPython написан на C и вмещает ядро интерпретатора в сотни килобайт памяти. Его главная цель – исполнение Python-кода на устройствах с ограниченными ресурсами (например, микроконтроллеры с 256KB флэш и 16KB RAM). MicroPython реализует подмножество Python 3 (близко к версии 3.4, но с расширениями для низкоуровневого доступа к железу, например `machine` модуль). Обрезаны сложные возможности: нет динамической загрузки расширений, ограниченная рекурсия, упрощенный garbage collector (обычно простой mark-and-sweep). Особенности: MicroPython **не имеет GIL** – обычно он однопоточный, хотя поддерживает co-routines или простейший многопоточный sheduler на некоторых платформах. Управление памятью – **полностью автоматическое (GC)**, но без подсчета ссылок; программисту иногда приходится вызывать `gc.collect()` вручную на микроконтроллере, если знает, что впереди тяжелая аллокация. MicroPython компилирует исходники в байт-код, как CPython, но формат байт-кода свой и сильно оптимизирован по памяти (например, используются короткие 1-байтовые инструкции там, где возможно). Байт-код может храниться в pre-compiled виде (.mpy файлы) для быстрой загрузки. Из-за своих ограничений MicroPython несовместим с большинством Python-библиотек, но дает возможность использовать привычный Python-синтаксис для написания прошивок, работы с I2C/SPI, датчиками и пр.

Для разработчиков микроконтроллеров это огромное подспорье, т.к. сокращает время разработки (по сравнению с C). Отдельно существует **CircuitPython** (форк MicroPython от Adafruit, ориентированный на простоту обучения – там убраны некоторые “острые углы”, добавлены библиотеки для работы с электроникой). В целом MicroPython – пример того, как адаптировать Python к очень скромным аппаратным условиям, при этом сохраняя дух языка.

- **Другие** – Существует еще ряд менее распространенных реализаций. Например, **Pyston** – ответвление CPython с JIT (разрабатывался в Dropbox, сейчас open-source, показывает ускорение x2 на ряде нагрузок за счет JIT на основе LLVM, однако поддерживает ограниченный набор платформ). **Stackless Python** – вариант CPython с измененной моделью исполнения (без использования C-стека, благодаря чему поддерживаются миллионы “микро-потоков”, использовался для высокопараллельных приложений). **Berp** (Python на Haskell), **Skulpt** (интерпретатор Python на JavaScript) и др. В последние годы интерес вызывают проекты на базе **WebAssembly**: кроме Pyodide, существует **Wasmer Python** (встраивание Wasm-движка в CPython для изолированного запуска модулей), **EmPython** (еще один CPython на WASM). Однако, все эти реализации пока нишевые.

Сравнительная таблица (кратко):

Реализация	Язык основы	Особенности и преимущества	Ограничения/ совместимость
CPython	C (C89)	Эталонная реализация, максимальная совместимость, богатый C API для расширений.	GIL, умеренная скорость интерпретации.
PyPy	RPython (subset of Python)	JIT-компиляция, высокая скорость вычислений, автоматический GC, поддержка большинства Python фич.	Ограничена поддержка C-API (нужен cffi), память может потреблять больше, старт медленнее.
Jython	Java	Бегает на JVM, использует оптимизации HotSpot, тесная интеграция с Java-библиотеками, без GIL (настоящая многопоточность).	Python 2.7 (на момент 2023), нет поддержки C-расширений (нужны Java-аналоги).
IronPython	C# (.NET)	Работает на .NET/Mono, доступ ко всем .NET-ассембли, можно встраивать в C# приложения, многопоточность через CLR.	Отстает по версии (Python 2.7/3.4), несовместим с C-API модулями (только .NET).
GraalPy	Java (Truffle/ Graal)	Современный JIT на GraalVM, поддержка Python 3.11, высокое быстродействие, возможность компиляции в self-contained exe, интероп с Java/JS.	Требуется GraalVM, пока экспериментальный, C-расширения ограниченно через эмуляцию.

Реализация	Язык основы	Особенности и преимущества	Ограничения/ совместимость
Nogil (PEP 703)	C (fork CPython)	Отсутствие GIL → эффективная работа многопоточных задач на несколько ядер, высокая совместимость (база CPython).	Пока не влит, сниженная скорость в однопоточных задачах, потребует адаптации некоторых C-расширений.
Pyodide	C/C++ (Emscripten)	CPython в браузере (WebAssembly), поддерживает numpy/pandas и др., позволяет выполнять на клиенте тот же код, что и на сервере.	Скорость ниже нативной, нет доступа к полноценной ОС, размер загрузки (мегабайты wasm).
Brython	JavaScript	Транспиляция Python→JS, удобство для веб-разработки (писать фронтенд на Python), прямой доступ к DOM и браузерным API.	Не поддерживает C-модули, производительность зависит от JS, не предназначен для тяжелых вычислений.
MicroPython	C	Ультра-легкий интерпретатор для микроконтроллеров, малый объем, специализированные модули для GPIO, быстрый запуск.	Подмножество Python (нет complex libs), ограничение по памяти/рекурсии, своя стандартная библиотека.

Вывод: Экосистема Python богата альтернативными реализациями. Выбор зависит от задачи: для максимальной скорости на CPU – PyPy, Pyston или GraalPy; для интеграции с JVM/.NET – Jython или IronPython; для браузера – Pyodide или Brython; для микроконтроллеров – MicroPython. CPython же остается универсальной “золотой серединой” по совместимости. Зная об этих вариантах, разработчик может подобрать инструмент, подходящий под конкретные требования (например, ускорить вычислительный модуль на PyPy, а UI оставить на CPython + Tkinter, или в embedded-проекте предпочесть MicroPython вместо Си для логики).

7. Стек и куча: память в Python, фреймы, рекурсия

Стек vs Куча – в чем разница? В контексте организации памяти, **стек** (stack) – это область, где хранятся контекст вызовов функций, в том числе локальные переменные и адрес возврата. Стек работает по принципу LIFO (последним пришел – первым вышел): каждый вызов функции помещает на вершину стека новый блок (активный кадр), а по завершении функции этот блок убирается. **Куча** (heap) – это область динамической памяти, откуда выделяются участки “произвольно” и управляются вручную или сборщиком мусора. Объекты в куче живут независимо от контекста вызова – до тех пор, пока на них есть ссылки. Проще говоря, стек организует выполнение (вызовы функций), а куча хранит сами данные (объекты), которые могут переживать вызовы.

Где хранятся переменные и объекты в Python? Python реализован так, что **все объекты (числа, списки, пользовательские объекты и т.д.) создаются в куче**. Переменные же являются по сути *ссылками* на объекты. Когда в коде пишут `x = 100`, Python создаст объект-целое число `100` в куче (если он еще не создан ранее или не интернирован) и запишет в

текущую область видимости ссылку `x`, указывающую на этот объект. Таким образом, **объекты всегда в куче**, а переменные – лишь именованные ссылки, хранящиеся либо в *таблице символов* (словаре) объекта, либо в массиве локальных переменных фрейма (для функций). Например, локальные переменные функции хранятся не на C-стеке, а в структуре фрейма в куче (PyFrameObject), но логически относятся к стеку вызовов Python. Грубо говоря: Python-объекты → куча; имена и ссылки → часть контекста исполнения (стека).

Создание переменных в Python – это присваивание ссылки на существующий объект или создание нового объекта в куче и привязка его к имени. Например, `a = []` создаст новый пустой список (объект `list`) в куче и свяжет его с именем `a` в текущем фрейме. Если затем `b = a`, то **новый объект не создается** – имя `b` привязывается к тому же объекту списка (и его счетчик ссылок увеличивается). Такое различие (имена vs объекты) приводит к тому, что операции копирования, сравнения и т.п. ведут себя иначе, чем в низкоуровневых языках.

Стек вызовов и фреймы функций. Каждый раз при вызове функции интерпретатор создает **фрейм** (кадр) – Python-объект, представляющий выполнение этой функции. В фрейме хранится:
- код-объект функции (`f_code`),
- указатель на **инструкцию** (program counter), которую предстоит выполнить (в CPython это offset в массиве байт-кода, поле `f_lasti`),
- **слоты локальных переменных** и ячейки для свободных переменных (для замыканий),
- ссылки на объекты глобального и встроенного пространства имен (для разрешения глобальных имен),
- **операндный стек** (evaluation stack) – используется для вычисления выражений по мере выполнения байт-кода,
- некоторые служебные флаги (например, признак, является ли фрейм генератором, и т.п.),
- ссылка на предыдущий фрейм (тот, который вызвал текущий).

Эти фреймы, создаваемые в куче, и образуют **стек вызовов Python**. Когда одна функция вызывает другую, текущий фрейм сохраняется, создается новый фрейм для вызванной функции, и управление переходит к нему. По завершении вызванной функции, ее фрейм уничтожается (или сохраняется, если это генератор, ожидающий возобновления) и управление возвращается к предыдущему фрейму, продолжая с той инструкции, где был вызов. Глубина рекурсии ограничена размерами стека: в CPython стоит лимит (по умолчанию ~1000), чтобы не переполнить C-стек (так как в CPython каждый Python-вызов рекурсивно вызывает функцию C eval, что занимает и системный стек). Фреймы Python упорядочены линейно – нет *параллельного* стека, кроме как при многопоточности (каждый поток имеет свой стек вызовов).

Как выглядит фрейм при рекурсии? При рекурсивном вызове функции `fact(n)` (факториал), где внутри `fact` вызывает сам себя `fact(n-1)`, для каждого уровня рекурсии создается новый фрейм с собственными локальными переменными `n` и т.д. Например: `fact(3)` вызывает `fact(2)`, вызывает `fact(1)`. В момент самого глубокого вызова (n=1) стек из трех фреймов: верхний (n=1), под ним (n=2), ниже (n=3). Каждый хранит свое значение `n` и ждет результата своего подвызыва. Когда глубокий возвращает 1, фрейм (n=1) завершается, управление возвращается в фрейм (n=2), там вычисляется `2*1` и возвращается, затем фрейм (n=2) завершается, и т.д. Если глубина рекурсии превысит лимит, Python вызовет исключение `RecursionError` и начнет разматывать стек, удаляя фреймы, как при исключении (unwinding).

Связь фреймов с байт-кодом и инструкция `RESUME`**.** Мы уже упоминали, что с версии 3.11 введена инструкция `RESUME` в начале каждого код-объекта функции. Практически это означает: когда создается новый фрейм и PVM готов начать его выполнение, первая операция будет

`RESUME`. Она не делает ничего в логике программы, но **для интерпретатора служит маркером границы фрейма**. В частности, Python 3.11+ объединяет инициализацию фрейма и начало исполнения. Раньше, до 3.11, первый исполняемый байт-код функции сразу соответствовал первой строке ее тела. Теперь же первая часть байт-кода – "построение фрейма" – помечена псевдо-операциями (включая `RESUME`), которые не отражаются как отдельные строки пользовательского кода. При дисассемблировании мы видели, что `RESUME` имеет номер строки, равный определению функции (на единицу меньше первой строки тела функции) [37]. Это сделано для корректности дебаггера и профилировщика: чтобы они не считали время на `RESUME` как время выполнения строки в теле функции.

С точки зрения работы фрейма, можно сказать, что **до** `RESUME` **происходит настройка фрейма** – загрузка констант default-аргументов, создание объекта генератора (если функция – генератор), размещение ячеек под локальные переменные. А после `RESUME` идет уже "реальный" код функции. Например, если функция имеет аннотации, Python может вставить специальный байт-код для обработки аннотаций до `RESUME`. Фрейм хранит указатель на текущую инструкцию; при вызове функции он указывает на начало, PVM видит `RESUME` и просто шагает дальше. Вся эта механика – под капотом, но важно понимать: **каждый фрейм имеет чётко определённый жизненный цикл** (создан – выполнен – уничтожен), и `RESUME` помогает интерпретатору управлять этим циклом эффективно.

Stack vs Heap применительно к Python-памяти: В Python программист не контролирует явно, что идет в стек, а что в кучу – это решает интерпретатор. Общие правила: - Локальные переменные (ссылки на объекты) хранятся в структуре фрейма (который живет в куче, но логически это "стековая" вещь, ибо время жизни ограничено вызовом). - Глобальные переменные хранятся в словаре модуля (объект `dict` в куче). - Объекты – всегда в куче (включая объекты, представляющие функции, классы, модули). - Вложенные функции и замыкания: свободные переменные хранятся в специальной куче (ячейки, `cell`), на которые ссылаются и внешний и внутренний фреймы.

Так как Python использует сборку мусора, программисту редко приходится думать, находится ли что-то на стеке или куче – все объекты, которые больше не нужны, рано или поздно будут удалены. Но есть влияющие моменты: например, глубокая рекурсия использует много памяти стека (C-стека) и может привести к переполнению (отсюда лимит). Или: крупные объекты в куче могут не освобождаться сразу, если остались циклические ссылки (до следующего прохода GC).

Итог: Стек в Python обеспечивает последовательность вызовов и хранит контекст исполнения (в виде фреймов), **куча** – хранит все данные (объекты). Каждый вызов функции создает фрейм (в куче) и помещает его в стек вызовов; рекурсия – наращивает стек множеством фреймов. Инструкция `RESUME` специально введена, чтобы разграничить и оптимизировать начало работы нового фрейма в байт-коде. Знание этих механизмов помогает понимать, почему, скажем, присваивание списка переменной не копирует список (оба имени указывают на один heap-объект), или почему слишком глубокая рекурсия падает, или как работает traceback (он просто проходит по связному списку фреймов, отображая файлы и номера строк).

Вывод: в Python переменные – это ссылки, объекты живут в куче, а выполнение программы организовано через стек вызовов (цепочку фреймов). Управление этими ресурсами берет на себя интерпретатор: выделяет память новым объектам в куче, освобождает невостребованное, создает/удаляет фреймы при вызовах, следит за тем, чтобы не переполнить стек. Понимание разницы между стеком и кучей позволяет писать более эффективный и безопасный код

(например, избегать излишней рекурсии или учитывать изменение изменяемых объектов при передаче).

---

1 2 5 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 Cache hierarchy - Wikipedia
https://en.wikipedia.org/wiki/Cache_hierarchy

4 CPU Cache Basics - DEV Community
https://dev.to/larapulse/cpu-cache-basics-57ej

6 9 10 11 12 13 18 19 What Is the __pycache__ Folder in Python? – Real Python
https://realpython.com/python-pycache/

7 A Tour of CPython Compilation - DEV Community
https://dev.to/cwprogram/a-tour-of-cpython-compilation-cd5

8 Understanding Python Bytecode and the Virtual Machine for Better Development - DEV Community
https://dev.to/imsushant12/understanding-python-bytecode-and-the-virtual-machine-for-better-development-55a9

14 15 16 17 Python | Compiled or Interpreted ? | GeeksforGeeks
https://www.geeksforgeeks.org/python-compiled-or-interpreted/

20 21 22 Compile main Python program using Cython - Stack Overflow
https://stackoverflow.com/questions/5105482/compile-main-python-program-using-cython

23 Internal working of Python | GeeksforGeeks
https://www.geeksforgeeks.org/internal-working-of-python/

24 37 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

25 26 27 29 Just-in-time compilation - Wikipedia
https://en.wikipedia.org/wiki/Just-in-time_compilation

28 How does the JVM decided to JIT-compile a method (categorize a ...
https://stackoverflow.com/questions/35601841/how-does-the-jvm-decided-to-jit-compile-a-method-categorize-a-method-as-hot

30 31 32 33 34 35 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

36 GraalPy – A high-performance embeddable Python 3 runtime for Java
https://news.ycombinator.com/item?id=41570708