

# СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ



К.Т.Н.  
Папулин Сергей Юрьевич  
*[papulin\\_bmstu@mail.ru](mailto:papulin_bmstu@mail.ru)*

# Лекция. CPython



- CPython
- Объекты и типы данных
- Компилятор
- Интерпретатор
- Управление памятью
- Сборщик мусор
- Глобальная блокировка интерпретатора (GIL)

# CPython

- Компилятор – транслирует исходный код на некотором языке в другой язык или в выполняемые низкоуровневые инструкции (ассемблер или машинный код)
- Интерпретатор – транслирует инструкции некоторого языка в машинный код в процессе выполнения программы
- Интерпретаторы, как правило, сначала транслирует исходный код в более эффективное низкоуровневое представление, которое затем используется в процессе выполнения
- Компилируемые языки программирования в меньшей степени переносимы на разные CPU и ОС
- Интерпретируемые языки программирования, наоборот, как правило, платформо-независимые

- CPython – это Python интерпретатор, написанный на C
- Является интерпретатором по умолчанию
- Интерпретатор – программа, которая запускает Python скрипты/программы
- Существуют несколько реализаций Python интерпретатора помимо CPython:
  - Jython (на Java)
  - IronPython (на C#/.NET)
  - PyPy (RPython + JIT)

Выполнение Python программы можно представить тремя стадиями:

- Инициализация

CPython инициализирует структуры данных, необходимые для запуска Python, загружает встроенные типы, модули и пр.

- Компиляция

CPython компилирует исходный код в байт-код:

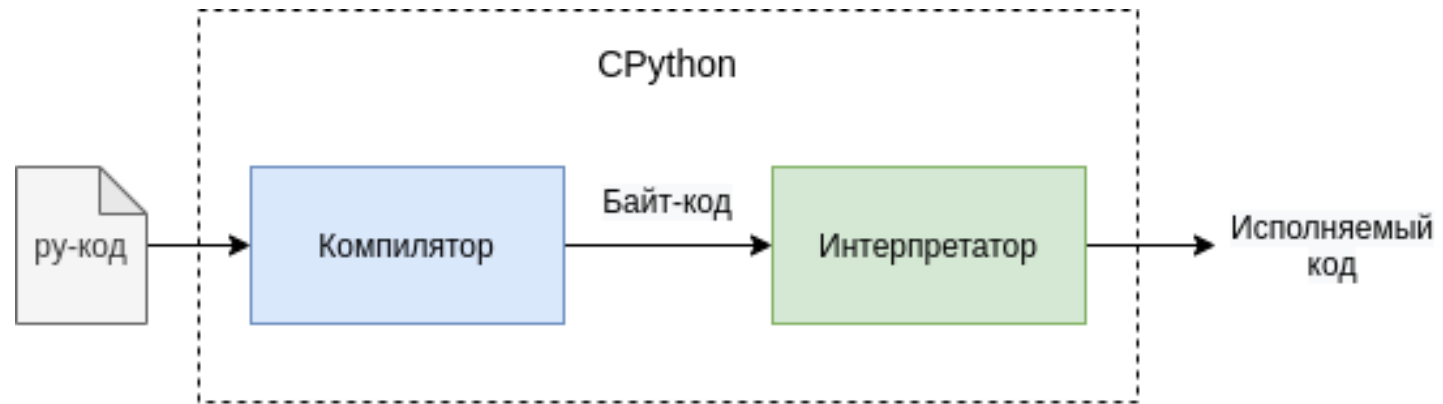
- Разбирает (парсит) исходный код
- Строит абстрактное синтаксическое дерево (AST)
- Генерирует байт-код из AST
- Оптимизация

- Интерпретация из байт-кода в машинный

- Байт-код – низкоуровневое платформо-независимое промежуточное представление исходного кода
- Часть CPython, отвечающая за выполнения байт-кода, называется виртуальной машиной. При выполнении виртуальная машина управляет объектами кода, фрейма, отслеживает состояния потоков, интерпретаторов и среды выполнения
- Таким образом, исходный код на Python компилируется в байт-код, который затем интерпретируется виртуальной машиной (средой выполнения) Python (PVM)



# Интерпретатор CPython (4)

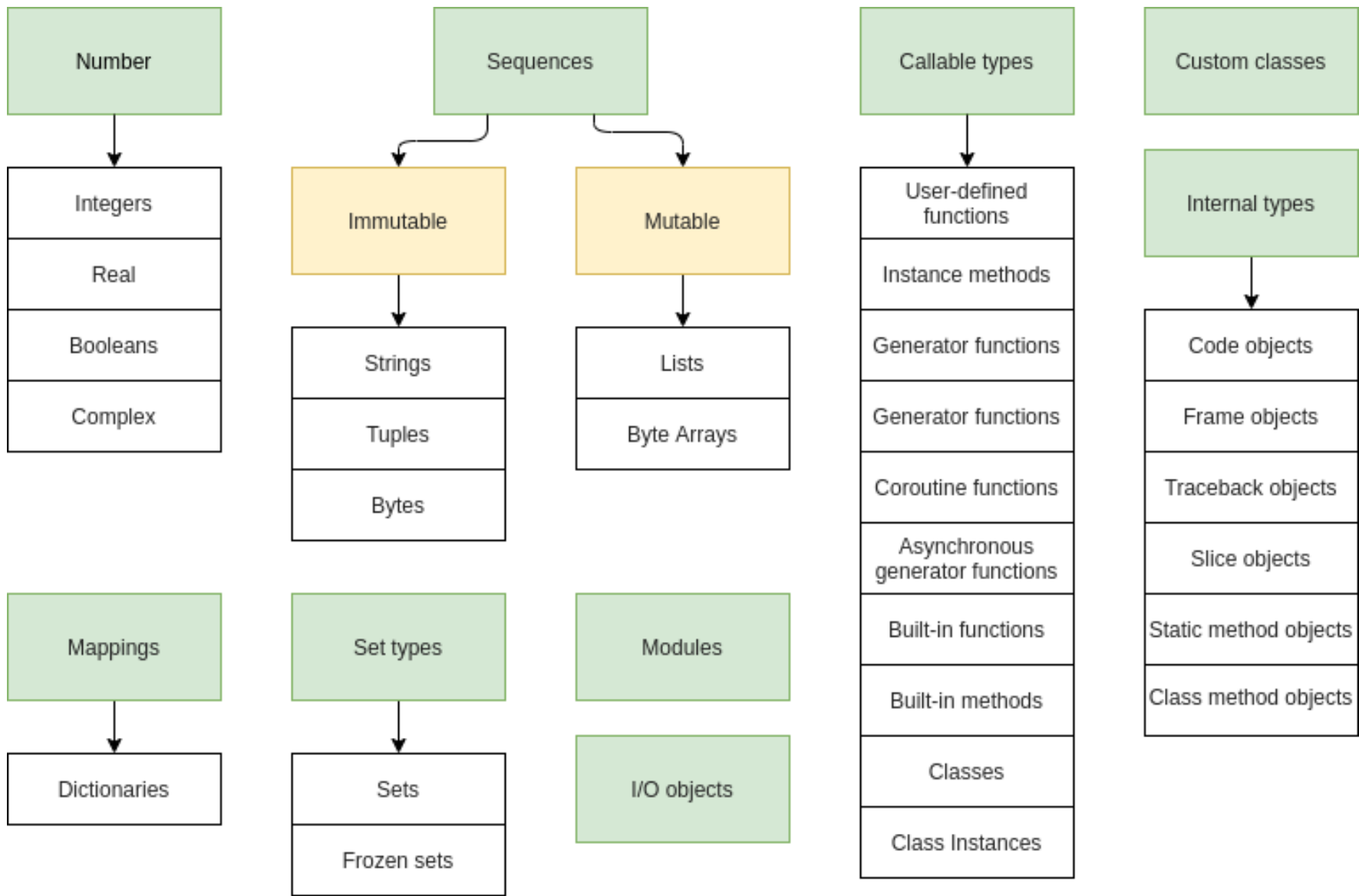


# Объекты и типы данных (1)

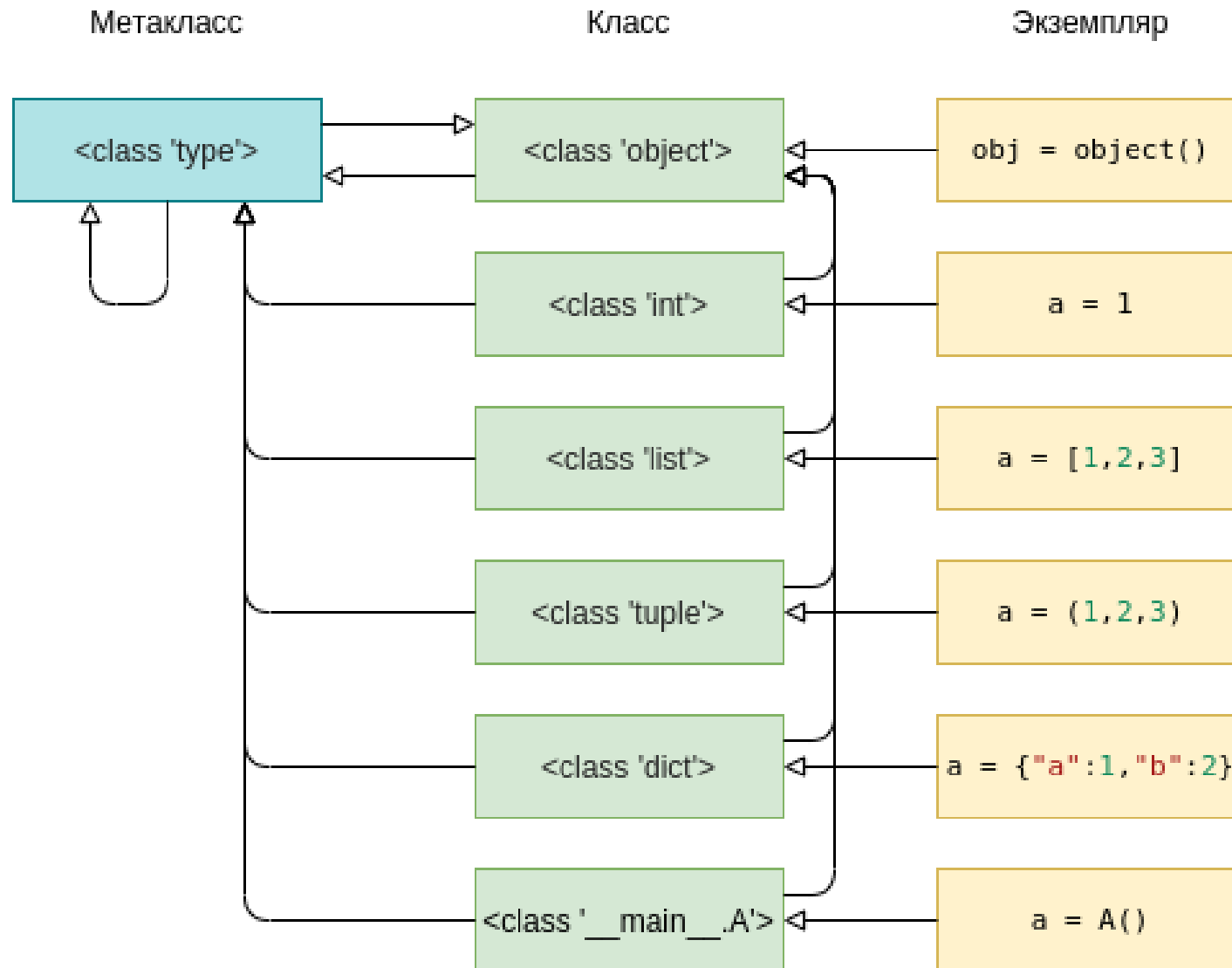
- Языки программирования со статической типизацией (C, Java) требуют в явном виде указывать тип переменных. Python относится к языкам программирования с динамической типизацией, в которых это не требуется
- Каждый объект в CPython имеет идентификатор, тип и значение. Идентификатор объекта не меняется после создания
- Объекты: целые числа (int), список (list), класс (class), модуль (module), функция (function), python байткод (code) и пр.
- Тип объекта определяет набор операций, которые поддерживает объект, и возможные значения
- Тип есть объект типа type

- Значение некоторых объектов может меняться
- Объекты, чьё значение может меняться, называются изменяемые (mutable): словари, списки
- Объекты, чьё значение не изменяется, называются неизменяемые (immutable): строки, числа, кортежи
- `object` является базовым классом для всех объектов, `type` тоже объект (экземпляр объекта)
- Объекты никогда не удаляются в явном виде. За это отвечает сборщик мусора

# Объекты и типы данных (3)



# Объекты и типы данных (4)



# Объекты и типы данных (5)

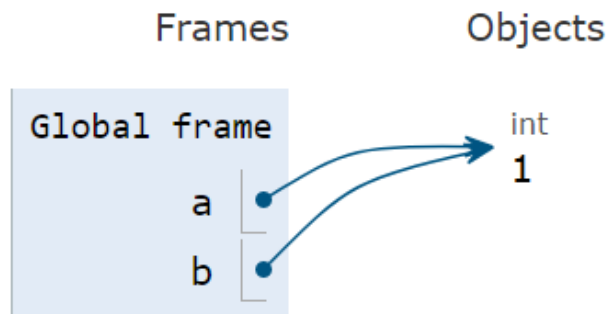
Выражения присваивания в Python:

- Создает объект (если не существует)
- Формирует связь между объектом и именем

Код

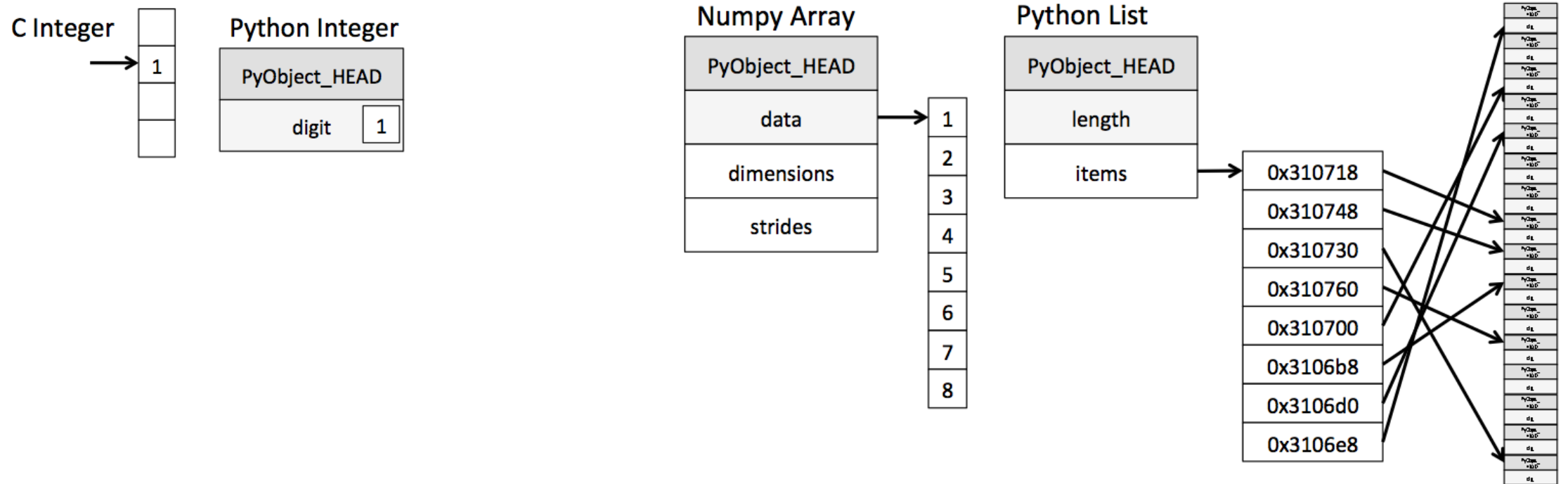
```
a = 1  
b = 1
```

Визуализация



- В CPython объекты соответствуют структуре PyObject
- Все Python объекты (PyObject) имеют:
  - ob\_refcnt – счетчик ссылок (используется для сборщика мусора)
  - ob\_type – определяет тип объекта: указатель на структуру описания Python объекта (например, int, list, dict, функция и пр.)
- Почти все Python объекты находятся в куче. Для работы с объектами объявляются переменные-указатели типа PyObject\*
- Исключением являются объекты типов. Представляются статическими объектами PyTypeObject

# Объекты и типы данных (6)





В CPython компиляция исходного кода в байт-код происходит в несколько шагов:

- Токенизация исходного кода (Parser/tokenizer.c)
- Разбор потока токенов в абстрактное синтаксическое дерево (AST) (Parser/parser.c):  
Абстрактное синтаксическое дерево (AST) – высокоуровневое представление структуры программы. Каждый узел дерева обозначает структуры, встречающуюся в исходном коде
- Преобразование AST в граф потока управления (CFG) (Python/compile.c): Граф потока управления – направленный граф, который моделирует поток программы посредством базовых блоков. Каждый блок содержит соответствующий ему байт-код программы.
- Формирования байт-кода на основе графа (Python/compile.c): Код напрямую генерируется из базовых блоков (a post-order depth-first search)

Исходный код на Python

```
def isum(a, b):  
    c = a + b  
    return c
```

Байт-код – последовательность инструкций. Каждая инструкция состоит из двух байтов:  
код операции (opcode) и аргумент (arg)

```
b'|\x00|\x01\x17\x00}\x02|\x02S\x00'
```

Исходный код на Python

```
def isum(a, b):  
    c = a + b  
    return c
```

Дисассемблер

2	0	LOAD_FAST	0 (a)
	2	LOAD_FAST	1 (b)
	4	BINARY_ADD	
	6	STORE_FAST	2 (c)
3	8	LOAD_FAST	2 (c)
	10	RETURN_VALUE	

## Дисассемблер

```
2      0 LOAD_FAST      0 (a)
      2 LOAD_FAST      1 (b)
      4 BINARY_ADD
      6 STORE_FAST      2 (c)

3      8 LOAD_FAST      2 (c)
     10 RETURN_VALUE
```

## Таблица сопоставления кодов

OPNAME	OPCODE	BYTECODE	ARG	BYTEARG
LOAD_FAST	124	b' '	0	b'\x00'
LOAD_FAST	124	b' '	1	b'\x01'
BINARY_ADD	23	b'\x17'	None	b'\x00'
STORE_FAST	125	b'}'	2	b'\x02'
LOAD_FAST	124	b' '	2	b'\x02'
RETURN_VALUE	83	b'S'	None	b'\x00'

# Компилятор (4)

Исходный код



Абстрактное синтаксическое дерево (AST)

```
def isum(a, b):
    return a + b
isum(1, 2)
```

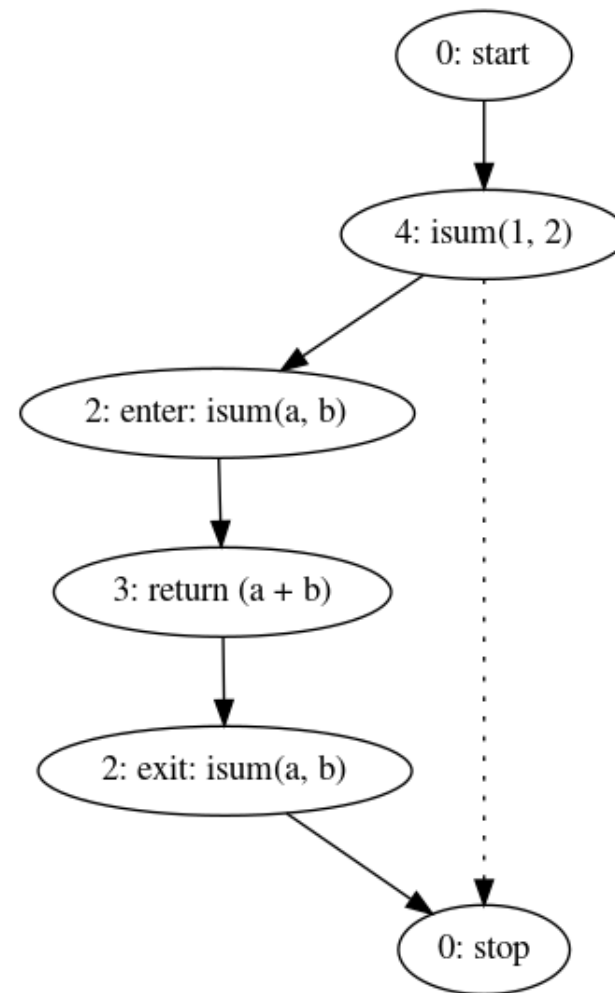
```
Module(
  body=[
    FunctionDef(
      name='isum',
      args=arguments(
        args=[
          arg(arg='a', annotation=None),
          arg(arg='b', annotation=None)],
        vararg=None, kwonlyargs=[],
        kw_defaults=[],
        kwarg=None, defaults=[]),
      body=[
        Return(
          value=BinOp(
            left=Name(id='a', ctx=Load()),
            op=Add(),
            right=Name(id='b', ctx=Load()))),
        ],
      decorator_list=[], returns=None),
    Expr(value=Call(
      func=Name(id='isum', ctx=Load()),
      args=[Num(n=1), Num(n=2)], keywords=[]))
  ]
)
```

## Абстрактное синтаксическое дерево (AST)

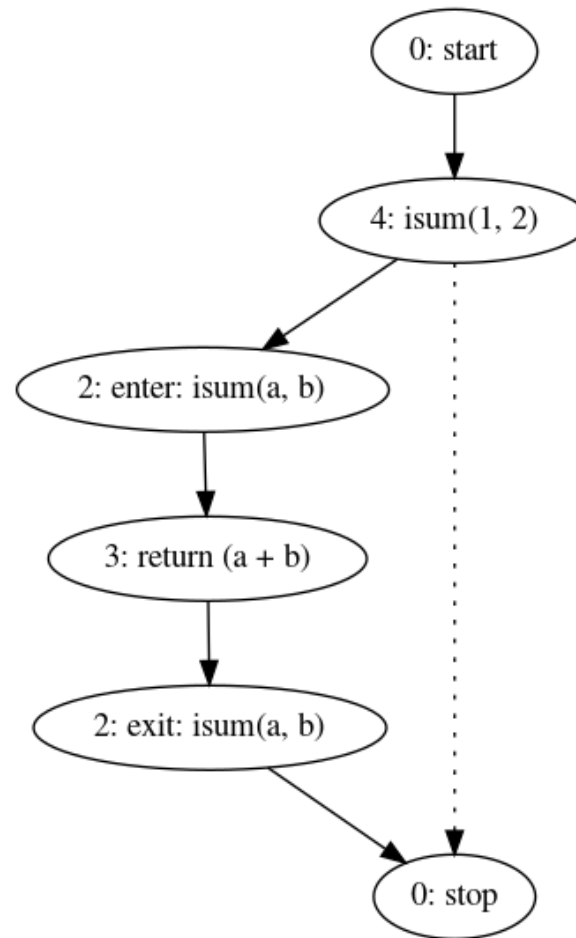
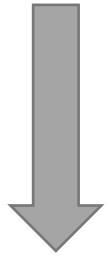


## Граф потока управления (CFG)

```
Module(
  body=[
    FunctionDef(
      name='isum',
      args=arguments(
        args=[
          arg(arg='a', annotation=None),
          arg(arg='b', annotation=None)],
        vararg=None, kwonlyargs=[],
        kw_defaults=[],
        kwarg=None, defaults=[]),
      body=[
        Return(
          value=BinOp(
            left=Name(id='a', ctx=Load()),
            op=Add(),
            right=Name(id='b', ctx=Load()))),
        ],
      decorator_list=[], returns=None),
    Expr(value=Call(
      func=Name(id='isum', ctx=Load()),
      args=[Num(n=1), Num(n=2)], keywords=[]))
  ])
)
```



## Граф потока управления (CFG)



## Байт-код

`b'd\x00d\x01\x84\x00Z\x00e\x00d\x02d\x03\x83\x02\x01\x00d\x04S\x00'`

- Виртуальная машина CPython при выполнении команда использует стек для хранения и извлечения данных
  - `LOAD_FAST` – добавляет локальную переменную в стек (адреса `a` и `b`)
  - `BINARY_ADD` – извлекает два элемента из стека, складывает и добавляет в стек
  - `RETURN_VALUE` – извлекает значение и возвращает результат
- Выполнение байт-кода происходит в большом исполняемом цикле (evaluation loop) до тех пор, пока есть инструкции на выполнение



## Объект кода (PyCodeObject)

- После компиляции исходного кода на Python в байт-код, виртуальная машина берет инструкции (opcode) из PyCodeObject
- Часть кода, которая выполняется как единый блок (например, модуль или функция), называется блоком кода (code block)
- СPython хранит информацию о том, что выполняет некоторый блок кода, в структуре называемой объект кода (code object). Он содержит байт-код и такие данные как список имен переменных, используемых внутри блока. Запустить модуль или вызвать функцию означает начать выполнение соответствующего объекта кода

## Объект функции (PyFunctionObject)

- Объект функции помимо объекта кода включает дополнительную информацию, такую как имя функции, docstring, аргументы по умолчанию, значения переменных, объявленных внутри блока функции
- Для создания объекта функции используется инструкция `MAKE_FUNCTION`

## Объект фрейма (PyFrameObject)

- Когда виртуальная машина выполняет объект кода, она должна отслеживать значения переменных и постоянно изменяющиеся значения стека. Она также должна запоминать, где она завершила выполнения текущего объекта кода, чтобы начать выполнение другого, и куда затем вернуться
- CPython хранит эту информация в объекте фрейма. Фрейм отслеживает состояние выполнения объект кода

- Состояние среды выполнения

Глобальное состояние процесса, включает GIL и управление памятью

- Состояние интерпретатора

Группа потоков и некоторых данных, которые являются общими, например, импортированные модули

- Состояние потока

Данные относящиеся к одному потоку ОС, включает стек вызова

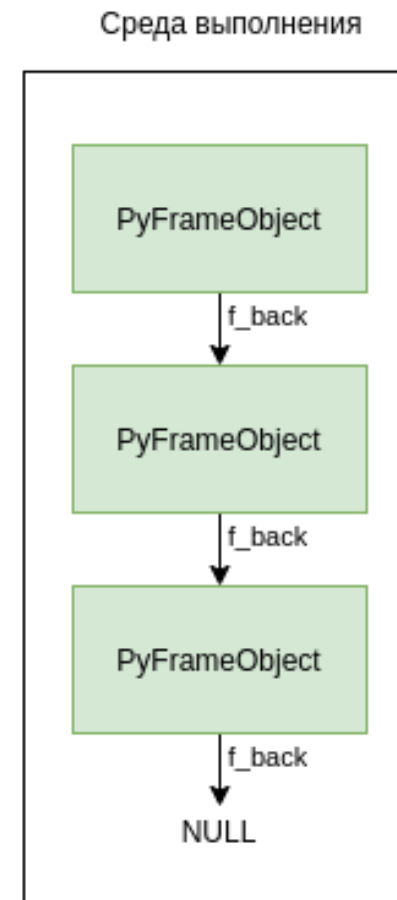
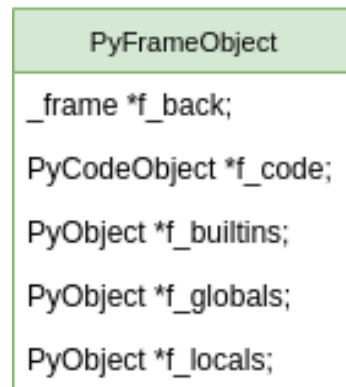
- Основной цикл выполнения

Место, где выполняются объекты фреймов



- Основная структура запущенной Python программы. Содержит один элемент – фрейм – для каждого вызова функции. Нижняя часть стека – точка входа программы. Каждый вызов функции добавляет новый фрейм в стек, а при завершении выполнения извлекается из него
- Первый фрейм создается для выполнения объекта кода модуля. CPython создает новый фрейм каждый раз, когда необходимо выполнить другой объект кода
- Каждый фрейм имеет ссылку на предыдущий фрейм
- Представляется как стек фреймов, в котором текущий фрейм находится сверху
- Данная структура называется стеком вызова (call stack)
- При возвращении из текущего выполняемого фрейма CPython продолжает выполнение следующих инструкций предыдущего фрейма

- Фрейм представляются структурой PyFrameObject. Для создания фрейма используется функция PyFrame\_New
- После получения инструкции (opcode) из PyFrameObject, функция PyEval\_EvalFrameEx обработает её и запустит



- Стек данных

В каждом фрейме есть стек выполнения – стек данных. Этот стек используется при выполнении инструкций с данными, например, загрузка аргумента, сложение элементов и пр.

- Стек блока

Каждый фрейм содержит стек блоков. Используется для отслеживания определенных типов структур управления: циклов, обработчиков исключений (try/except). Помогает определить какой блок является активным в текущий момент и, например, правильно обрабатывать команды `continue` и `break`



Разбор примеров

<https://towardsdatascience.com/understanding-python-bytecode-e7edaae8734d>

Визуализация

<https://pythontutor.com/>

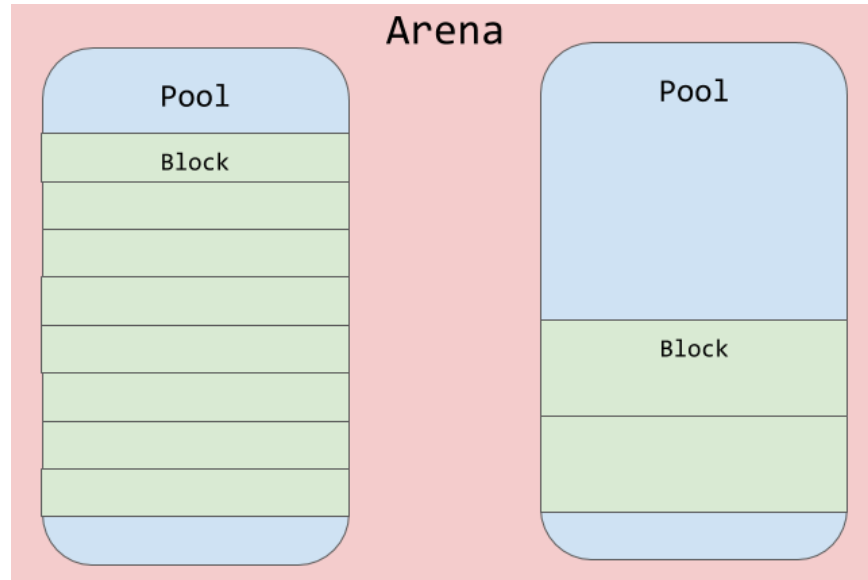
- Все Python объекты и структуры содержатся в частной куче (private heap). Управление этой кучей обеспечивается менеджером памяти CPython
- В CPython есть собственный распределитель памяти – **pymalloc**, который построен поверх C распределителя (C-runtime-allocator). Отвечает за выделение памяти небольшим объектам (меньше или равными 512 байт) с коротким временем жизни. Для этого используется так называемые арены (arenas) с фиксированным размером в 256 КБ
- Компоненты pymalloc: арены, пулы, блоки
- Для размещения объектов больше 512Б используются
  - PyMem\_RawMalloc(size\_t n)
  - PyMem\_RawRealloc(size\_t nelem, size\_t elsize)



# Управление памятью. `rumalloc` (1)

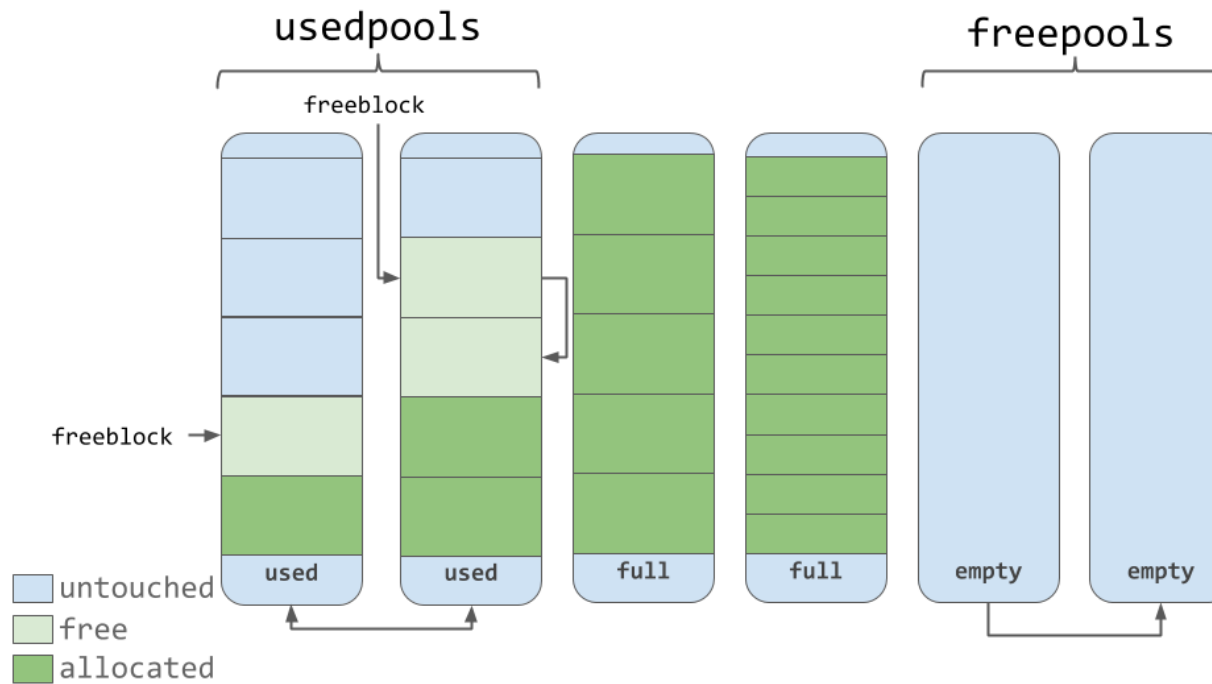
- Пулы состоят из блоков одного размера. Каждый пул поддерживает двойной связный список к другим пулам того же размера, что позволяет искать место нужного размера в разных пулах.
- Пулы могут находиться в трех состояниях:
  - Used (используется) – пул с доступными блоками для размещения данных
  - Full (полный) – пул, в котором нет свободных блоков не содержащих данных
  - Empty (пустой) – пустой пул, в котором при необходимости могут быть размещены блоки любого размера
- Если нам необходимо 8 байтный блок и нет доступных блоков в пулах с состоянием `used`, то пустой пул будет проинициализирован для хранения 8 байтных блоков. После размещения блока пул переходит в состояние использования (`used`) и доступен для размещения 8 байтных блоков в будущем
- Если заполненный пул в состоянии `full` освобождает некоторые свои блоки, так как записанные данные в память больше не нужны, то пул переходит в состояние `used` и добавляется в список используемых пулов данного размера.

# Управление памятью. pymalloc (2)



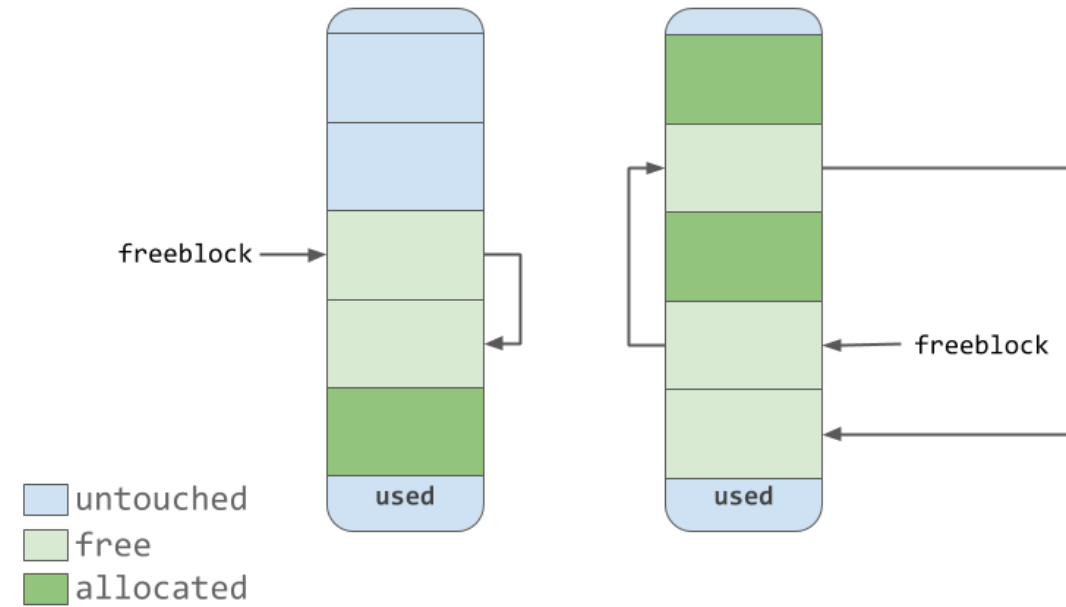
* Request in bytes	Size of allocated block	Size class idx
* 1-8	8	0
* 9-16	16	1
* 17-24	24	2
* 25-32	32	3
* 33-40	40	4
* 41-48	48	5
* 49-56	56	6
* 57-64	64	7
* 65-72	72	8
* ...	...	...
* 497-504	504	62
* 505-512	512	63

# Управление памятью. pymalloc (3)



- Блоки могут находиться в трех состояниях:
  - Untouched (нетронутые) – порция памяти, которая не было назначена/распределена
  - Free (свободные) – порция памяти, которая была назначена, но потом освобождена CPython'ом и больше не содержит релевантных данных
  - Allocated (назначенные) – порция памяти, которая содержит релевантные данные
- Свободные блоки организованы в виде связного списка. Свободные блоки доступны для размещения новых данных
- Если необходимо больше, чем имеющиеся доступные свободные блоки, то распределитель будет использовать нетронутые (untouched) блоки. Таким образом, распределитель пользуется нетронутыми блоками, только когда не хватает свободных

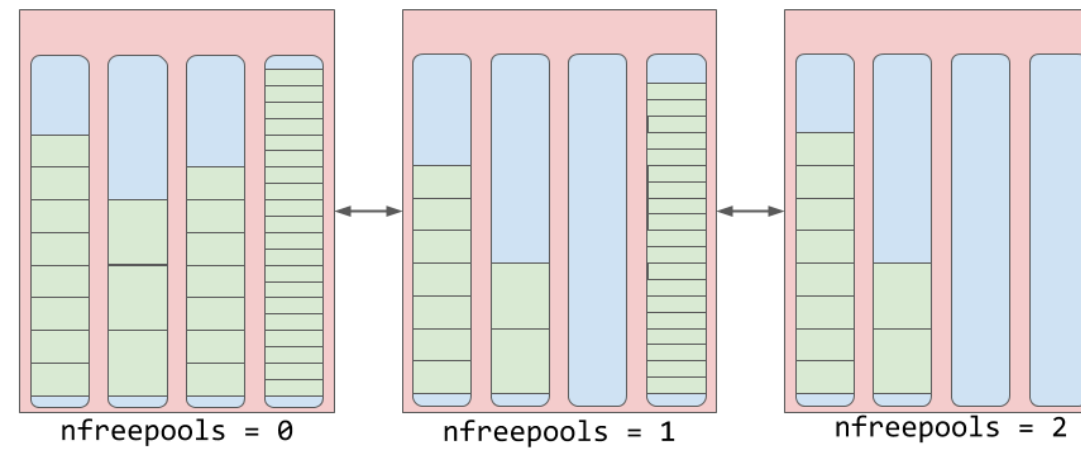
# Управление памятью. румalloc (5)



- Арены организованы в виде двойного связного списка `usable_arenas`. Этот список отсортирован по количеству доступных свободных пулов. Чем меньше свободных пулов, тем ближе к началу списка
- Это означает, что наиболее заполненный пул, будет выбран первым для размещения новых данных
- В действительности блоки при переходе в состояние `free` не освобождают память ОС. Python процесс по прежнему удерживает их для дальнейшего размещения новых данных
- Арена единственный компонент, который может действительно может освободить память. Поэтому аренам, которые ближе к тому, чтобы стать пустыми, дается шанс освободить в будущем память Python программы

# Управление памятью. pymalloc (7)

usable\_arenas



- Сборщик мусора необходим для освобождения памяти кучи от объектов, которые больше не используются, но при этом занимают некоторый объём памяти
- Для этой цели в CPython применяются две техники:
  - Подсчет ссылок (Reference counting)
  - Сборщик мусора с поколениями (Generational garbage collector)



- Подсчет ссылок на объекты – простая техника, в которой объекты освобождаются, когда на них нет ссылок в программе. Если на объект нет ссылок, то считается, что он больше не нужен
- Каждая переменная в Python есть ссылка (указатель) на объект
- Для отслеживания ссылок каждый объект имеет дополнительное поле, называемое счетчик ссылок, который увеличивается или уменьшается, когда создается или удаляется указатель на объект, соответственно
- Если счетчик достиг нулевого значения, то CPython вызывает специальную функцию, отвечающую за удаление объекта из памяти

# Сборщик мусора с поколениями (1)

- Техника подсчета ссылок имеет фундаментальную проблему – она не может обнаружить циклические ссылки
- Циклические ссылки возникают, когда один или более объектов ссылаются друг на друга
- Зацикливание может произойти только в объектах-контейнерах (которые могут содержать другие объекты), таких как списки, кортежи, словари, классы
- Алгоритм сборщика мусора с поколениями не отслеживает неизменяемые типы. Исключением являются кортежи
- Кортежи и словари, содержащие только неизменяемые объекты, при определенных условиях могут быть исключены из списка отслеживаемых объектов
- Техника подсчета ссылок используется только в случае отсутствия циклических ссылок

## Сборщик мусора с поколениями (2)

- В отличие от техники подсчета ссылок, которая работает в реальном времени, сборщик мусора для объектов с циклическими ссылками запускается периодически
- Чтобы сократить частоту вызова сборщика и как следствия пауз в CPython используется различные эвристики
- Сборщик мусора разделяет объекты-контейнеры на три поколения. Каждый новый объект начинает с первого поколения. Если объект выживает итерацию сборки мусора, он перемещается в следующее поколение. Сборка для ранних поколений выполняется более часто, чем для более поздних поколений
- Это связано с тем, что большинство вновь созданных объектов прекращают свое существование в течение короткого промежутка времени
- Это улучшает производительность сборщика мусора и сокращает паузы в работе

# Глобальная блокировка интерпретатора (GIL)

- Python интерпретатор не является полностью потокобезопасным (thread-safe). Для поддержки многопоточности Python программ, используется глобальная блокировка, называемая глобальная блокировка интерпретатора (Global Interpreter Lock – GIL). Перед доступом к Python объекту, текущий поток должен удерживать GIL. Без блокировки даже простейшие операции могут стать причиной проблем в многопоточной программе
- Пример: в случае, когда два потока одновременно увеличивают счетчик ссылок одного и того же объекта, может получиться так, что счетчик увеличится только на один вместо двух
- Поэтому только поток, который удерживает GIL, может производить операции над Python объектами или вызывать Python/C API функции
- Чтобы поддерживать подобие параллельности выполнения, интерпретатор переключает потоки с определенным интервалом. Блокировка также снимается перед блокирующими операциями ввода-вывода, такими как чтение или запись файла, так что другие потоки могут выполняться в промежутке
- Python интерпретатор хранит сопутствующую информацию о потоке в структуре данных PyThreadState. Глобальная переменная указывает на текущее состояние PyThreadState

- Python/C API Reference Manual / <https://docs.python.org/3.7/c-api/index.html>
- Python/C API Reference Manual. Introduction / <https://docs.python.org/3.7/c-api/intro.html>
- The Python Language Reference. Data model / <https://docs.python.org/3.7/reference/datamodel.html>
- Understanding Data Types in Python / <https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html>
- Design of CPython's Compiler / <https://devguide.python.org/compiler/>
- Understanding Python Bytecode / <https://towardsdatascience.com/understanding-python-bytecode-e7edaae8734d>
- Python behind the scenes: how the CPython VM works / <https://tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/>
- Memory Management / <https://docs.python.org/3.7/c-api/memory.html>
- Memory Management in Python / <https://realpython.com/python-memory-management/>
- Memory Management. Source Code / <https://github.com/python/cpython/blob/3.7/Objects/obmalloc.c>
- Design of CPython's Garbage Collector / [https://devguide.python.org/garbage\\_collector/](https://devguide.python.org/garbage_collector/)
- Garbage Collection for Python / <http://www.arctrix.com/nas/python/gc/>
- GC Module. Source Code / <https://github.com/python/cpython/blob/v3.7.12/Modules/gcmodule.c>