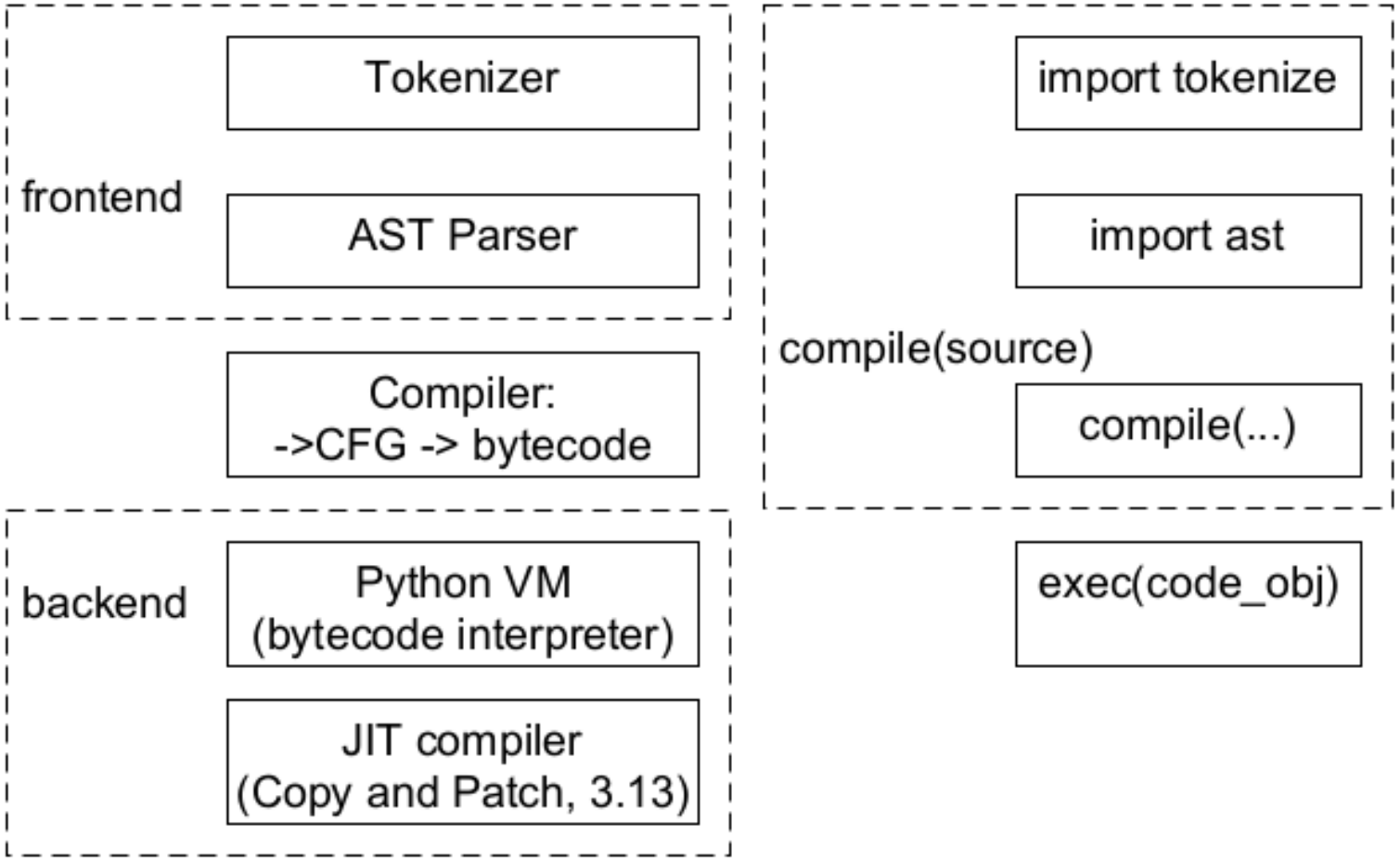
Compilers 101

Python internals and JIT compilers

Python internals & their pure Python interfaces



Python AST Parser

- Python 3.9 and before: LL(1) parser
  - Source code -> Concrete Syntax Tree (CST) -> AST

- Python 3.9+: PEG grammar approach (PEP-617)
  - theory intro in 2004 by Bryan Ford
  - Gvido was inspired by TatSu PEG parser implementation
  - lib2to3 had LL(1) parser, that’s why it was removed in 3.10
- PEG grammar is unambiguous => not commutative / ordered (get first)

Generate AST Parser

- PEG-grammar: **Grammar/python.gram** -> parser.c
- Generate parser in C
  - python -m pegen -v -o parser.c **./cpython/Grammar/python.gram**
- Generate parser in Python
  - pip install pegen
  - git clone https://github.com/we-like-parsers/pegen.git ./we-like-parsers_pegen/
  - python -m pegen -v -o parser.py **./we-like-parsers_pegen/data/python.gram**
- Grammar for C (~1.4k LoC) vs grammar for Python (~2

If you need AST Parser in C++

- Create own python.gram file with C++ Standard Library (std::string etc)
  - python -m pegen -v -o parser.cpp **./your_own/python.gram**
- Maybe use StringZilla for speed up (useful for many languages)
- ‘str’ codecs design is important (‘codecs’ are in pure Python now)
- Make C++ compiler for Python using LLVM or MLIR

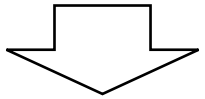
Python AST module (1/3)

- python -m ast --help
- ast.parse("def sum(a, b): return a + b")
- ast.NodeVisitor – base class to walk AST in read-only mode (picompile)
- ast.NodeTransformer – base class to walk AST to replace/remove nodes
- ast.walk() – walk without Visitor pattern

- ast.literal_eval() – light alternative to eval(), but still unsafe
- ast.unparse() – useful for decompiler (uncompyle6, pycdc), auto formatter

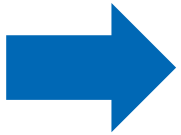
Python AST module (2/3

```python
def sum_up(a, b):
    return a + b

print(sum_up(3, 5))
```



```python
def sum_up(a: int, b: int) -> int:
    return a + b

print(sum_up(3, 5))
```

```python
import ast
with open(r"ast_input.py", "r") as f:
    content = f.read()
tree = ast.parse(content)
print(ast.dump(tree, indent=4))
```



```
Module(
    body=[
        FunctionDef(
            name='sum_up',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(arg='a'),
                    arg(arg='b')],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Return(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Name(id='b', ctx=Load()))))],
            decorator_list=[]),
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    Call(
                        func=Name(id='sum_up', ctx=Load()),
                        args=[
                            Constant(value=3),
                            Constant(value=5)],
                        keywords=[])],
                keywords=[]))],
    type_ignores=[])
```

Python AST module (3/3)

```
import ast
with open(r"ast_input.py", "r") as f:
    content = f.read()
tree = ast.parse(content)

class Folding(ast.NodeTransformer):
    def visit_BinOp(self, node):
        print(node)
        print(f"op={node.op}, left={node.left}, right={node.right}")
        print(dir(node))
        return self.generic_visit(node)

folding = Folding()
new_tree = folding.visit(tree)
```

```
<ast.BinOp object at 0x0000027151809F00>
op=<ast.Add object at 0x000002714F5549D0>, left=<ast.Constant object at 0x00000271518095A0>,
right=<ast.Constant object at 0x000002715180A050>
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__match_args__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_attributes', '_fields', 'col_offset', 'end_col_offset', 'end_lineno', 'left',
'lineno', 'op', 'right']
```

At the bottom of Python AST docs...

See also

- Green Tree Snakes, an external documentation resource, has good details on working with Python ASTs.
- ASTTokens annotates Python ASTs with the positions of tokens and text in the source code that generated them. This is helpful for tools that make source code transformations.
- leoAst.py unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.
- LibCST parses code as a Concrete Syntax Tree that looks like an ast tree and keeps all formatting details. It's useful for building automated refactoring (codemod) applications and linters.
- Parso is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). Parso is also able to list multiple syntax errors in your python file.

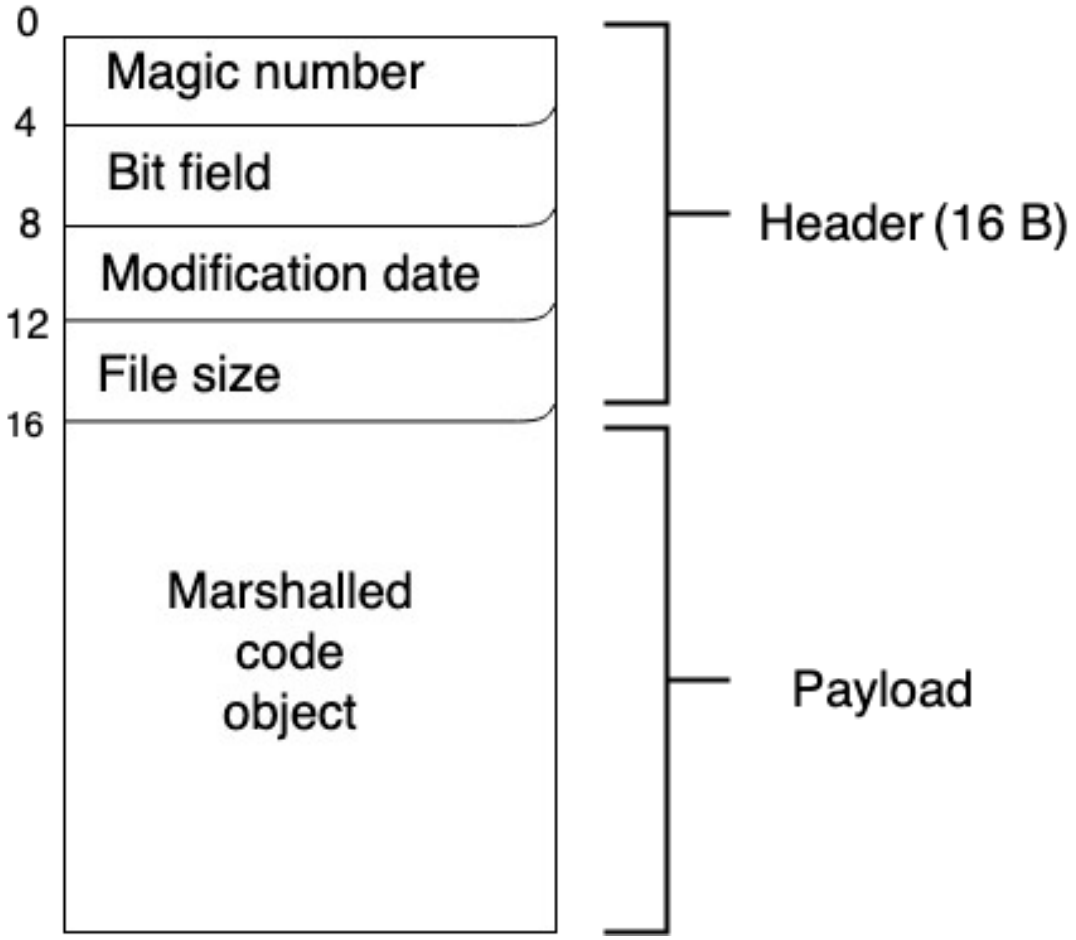
Python bytecode (IR)

- As any IR it has 3 forms:
  - Byte-code itself: ***.pyc** file content or code_object**.co_code**
  - In-memory representation: **code_object = compile(...)** or **PyCodeObject**
  - String representation: **import dis; dis.dis(code_object)**

```
>>> o = compile("def sum(a, b): return a + b", "<string>", "exec")
>>> o
<code object <module> at 0x00000216339D3730, file "<string>", line 1>
>>> dis.dis(o)
  1           0 LOAD_CONST               0 (<code object sum at 0x000
              2 LOAD_CONST               1 ('sum')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (sum)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE

Disassembly of <code object sum at 0x00000216339D
  1           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 RETURN_VALUE
```

.pyc file format & decoding

```
import marshal
import dis
with open(r"C:\Python3_11_3\Lib\__pycache__\os.cpython-311.pyc", "rb") as f:
    f.seek(16)
    bytecode = dis.Bytecode(marshal.load(f))
for instr in bytecode:
    print(instr.opname)
    print(dir(instr))
```



Bytecode interpreter (Python VM)

- Bytecode docs: https://devguide.python.org/internals/interpreter/ (3.11)
- 2 kinds of VM:
  - Stack-based VM (most popular, including Python VM)
    - More portable, easier implementation, more compact in memory (doesn't need addresses)
  - Register-based VM (https://github.com/iritkatriel/cpython/tree/regmachine2)
    - Less portable, need more memory (need addresses), should be faster (need other compiler)
- ceval.c, `_PyEval_EvalFrameDefault(...)` – can be replaced using `_PyInterpreterState_SetEvalFrameFunc(...)`
  - `#include "generated_cases.c.h"` (tier 1 / main interpreter)
  - `#include "executor_cases.c.h"` (tier 2 interpreter! new in 3.13, use micro-ops)

Bytecode specializations (Python 3.11, PEP-659)

- Bytecode ops are too abstract initially
- If “hot” code is detected (saturating counter), specialization is done:
  - Example: LOAD_GLOBAL -> LOAD_GLOBAL_BUILTIN / LOAD_GLOBAL_MODULE
  - Example: BINARY_ADD -> BINARY_ADD_INT / BINARY_ADD_FLT
- If input is changed few times (counter too), de-optimization happens
  - back to LOAD_GLOBAL
  - back to BINARY_ADD

Source code of Compiler and Interpreter(s)

- Python/ast.c, asdl.c, Parser/*.c (AST parser)
- Python/ast_opt.c (constant folding + few other simple optimizations)
- Python/compile.c (compile AST to CFG and then to bytecode)
  - #include <pycore_flowgraph.h>
  - #include <pycore_symtable.h> (symtable.c; import symtable)
- Python/ceval.c (bytecode interpreter, or Python VM)
  - #include “generated_cases.c.h” (tier 1 / main interpreter): case <opcode>: { <some logic>; }
  - #include “executor_cases.c.h” (new in 3.13; tier 2 interpreter): case <micro-op>: { <some logic>; }
- Python/jit.c (light JIT compiler: Copy and Patch approach)
  - #include “jit_stencils.h” (pieces of machine code generated by clang at build time)

Source code of built-in types and modules

- Objects/
  - dictobject.c, listobject.c, unicodeobject.c, ...
- Modules/
  - _json.c, socketmodule.c, timemodule.c, ...
- Python/
  - sysmodule.c, _warnings.c, bltinmodule.c, ...

- Lib/*.py

Python C API tiers and layers (1463 functions)

- Limited C API (== Stable ABI): ~900 functions
- Unstable C API see https://docs.python.org/3.13/c-api/stable.html

- C API docs index (very useful)
  - **Abstract Object Layer** (connected with PyTypeObject: virtual method table)
  - Concrete Object Layer
- How many lines of C needed to execute a + b in Python?
  - https://habr.com/ru/articles/780386/ (in Russian)
  - https://codeconfessions.substack.com/p/cpython-dynamic-dispatch-internals (orig)

Boxing and unboxing

- PyObject* to static type
  - `PyAPI_FUNC(long) PyLong_AsLong(PyObject *obj)`
- Static type to PyObject*
  - `PyAPI_FUNC(PyObject *) PyLong_FromLong(long ival)`

```
def outer_func(N: int) -> int:
    res: int = 0
    for i in range(N*N):
            res += inner_func(i)
    return res
```

3rd-party JIT compilers for Python functions

```
from numba import jit

@jit
def func(N):
    sum = 0
    for i in range(N):
            sum += i
    return sum

>>> func(100)
4950
>>> func
CPUDispatcher(<function func at 0x000002715247CF40>)
```

- @jit(cache=True, nopython=True) is recommended (cache=True saves ~1 sec.)

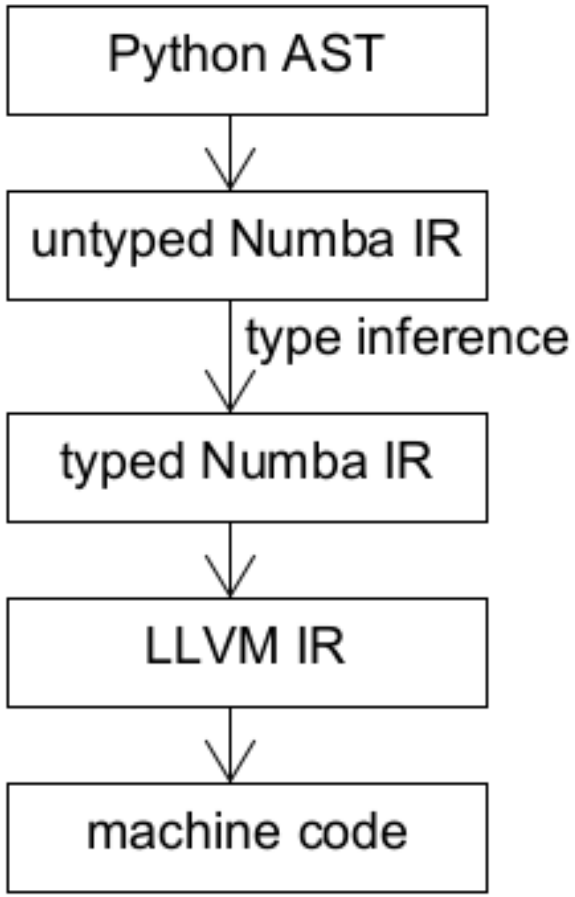
3rd-party JIT compilers for Python functions

- Numba
  - pip install llvmlite; -> Numba IR -> LLVM IR
  - TBB support, Intel only GPU support, special replacements for Numpy functions
- Taichi
  - GPU support; focus on graphics
  - more API
- 4 мушкетера: Python, Cython, Numba и Taichi (PyCon Weekend 2022)

3rd-party JIT compilers for Python

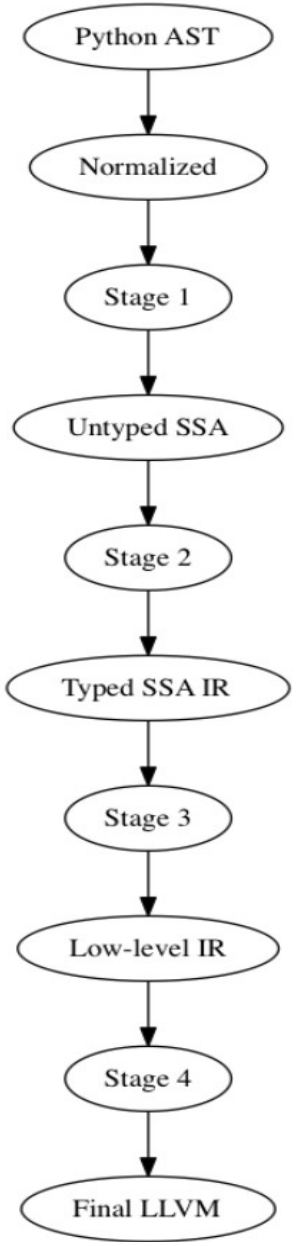
- PyPy is a Python implemented in RPython as a JIT compiler
  - Trace based GC (mark-and-sweep with modifications)
  - ctypes are very slow with C extensions (need to rewrite code to CFFI)
  - ~40% faster than CPython 3.11 in average (in pure Python)
- Pyjion: JIT compiler running on .NET VM (project is closed)
- Cinder: JIT compiler forked from CPython 3.9 (serves Nelzyagram*)
  - HIR (bytecode + expl. refcounting) -> LIR -> JIT compiler (own C++ implementation)
  - Full compatibility is not guaranteed and not checked (?)

Typical compilation pipeline (Numba)



- Env var NUMBA_DUMP_CFG=1
- NUMBA_DUMP_IR=1
- NUMBA_DUMP_ANNOTATION=1
- NUMBA_DUMP_LLVM=1
- NUMBA_DUMP_OPTIMIZED=1
- NUMBA_DUMP_ASSEMBLY=1

- search for “Numba Compiler Architecture”



Type inference for Python

- mypy + mypyC (transpiler to C)
- pytype (type hints are not necessary; by Google)
- pyright (type checker for VSCode; by Microsoft; written in TypeScript)
- JohnnyPeng18/HiTyper implements type dependency graph + some ML

- Type inference is not decidable (algorithmically unresolvable) – 2022
  - https://arxiv.org/pdf/2208.14755.pdf

JIT compiler (Copy and Patch approach)

- Brandt Bucher – A JIT Compiler for CPython (December, 2023)
  - The first idea publication at 2021
  - The first use for Lua (35% slower than LuaJIT) at 2023
  - Python JIT adds 2% - 9% performance improvement
  - LLVM (clang) is used at build time only. The distribution doesn’t have LLVM libs.
  - Python 3.13 may include it (configuration option: “-enable-experimental-jit”)

- Machine code (as bytes) is copied from C header to memory (Copy)
- Bytecode arguments are patched with data (Patch)

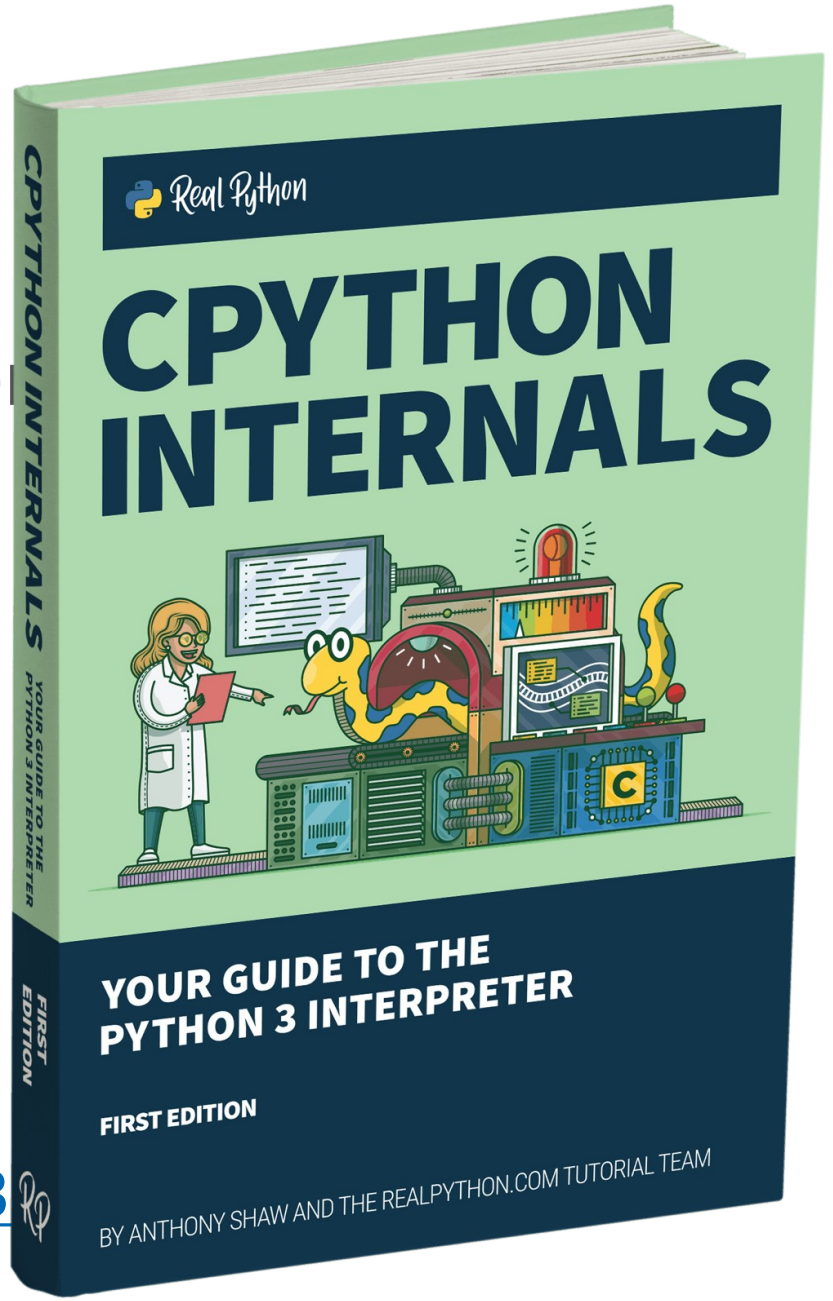
Global Interpreter Lock (GIL), no-GIL PEP-703

- Threads in Python are not parallel! GIL cannot be disabled in pure Python.
- GIL can be disabled in C code / C extension.
- GILectomy (Larry Hastings, Python 3.6 fork) – **failed**
- No-GIL (Sam Gross, MSFT, 2020-now, started as Python 3.9 fork)
  - Keynote talk at EuroPython 2022 (79x speed up for 80 cores)
  - PEP-703 accepted, Sam Gross approved as Python Core Developer in Feb 2024.
  - Biased RC (slowdown: 10% vs 60% w/ atomics), deferred RC (no RC for VM stack).
  - Immortal objects (None, True, ...): no RC for many constants, interned strings etc.
  - Thread-safe allocator “mimalloc” (born in 2019 at Microsoft)
  - Critical section C API

Recommended reading

- Anthony Shaw (Microsoft) “CPython Inte
  - (the 2nd edition covers versions till ~3.10)

- Python Dev Guide: CPython’s Internals
  - https://devguide.python.org/internals/
  - (more compact)
- About cell and free variables in closure
  - https://stackoverflow.com/a/23830790/36483



Recommended watching & conferences/meetups

- My Youtube playlist (Python, MLIR + LLVM, Mojo, Julia, Kotlin Native):
  - https://www.youtube.com/playlist?list=PLeZ4oq0ctHI2uO2SXcQj_DE3jHonxOJRk

- Лучший курс по Python (by Nikita Sobolev, in progress...)

- PyCon US: https://www.youtube.com/@PyConUS
- EuroPython: https://www.youtube.com/@EuroPythonConference
- PyCon Russia, Moscow Python, PiterPy, EkbPy, @Pytho_NN, Pytup, ...

Next time...

- MLIR practice in 300 lines of code
- Mojo🔥 & clangIR internals
- Intro to exception handling
- LLVM JIT engine (ORCv2) usage

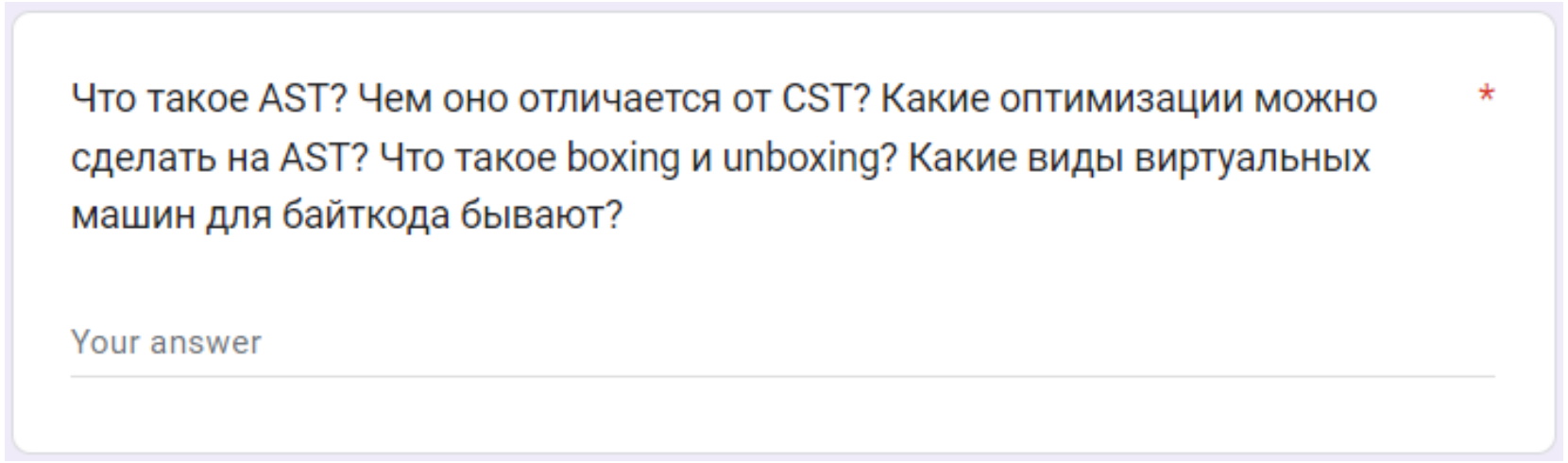
No lab assignment

- Where to ask questions
  - Telegram: @vasily_v_ryabov (questions)

Test

https://forms.gle/eNpQzdsvN99S7byY6

Submission time: **10 minutes**



Backup: