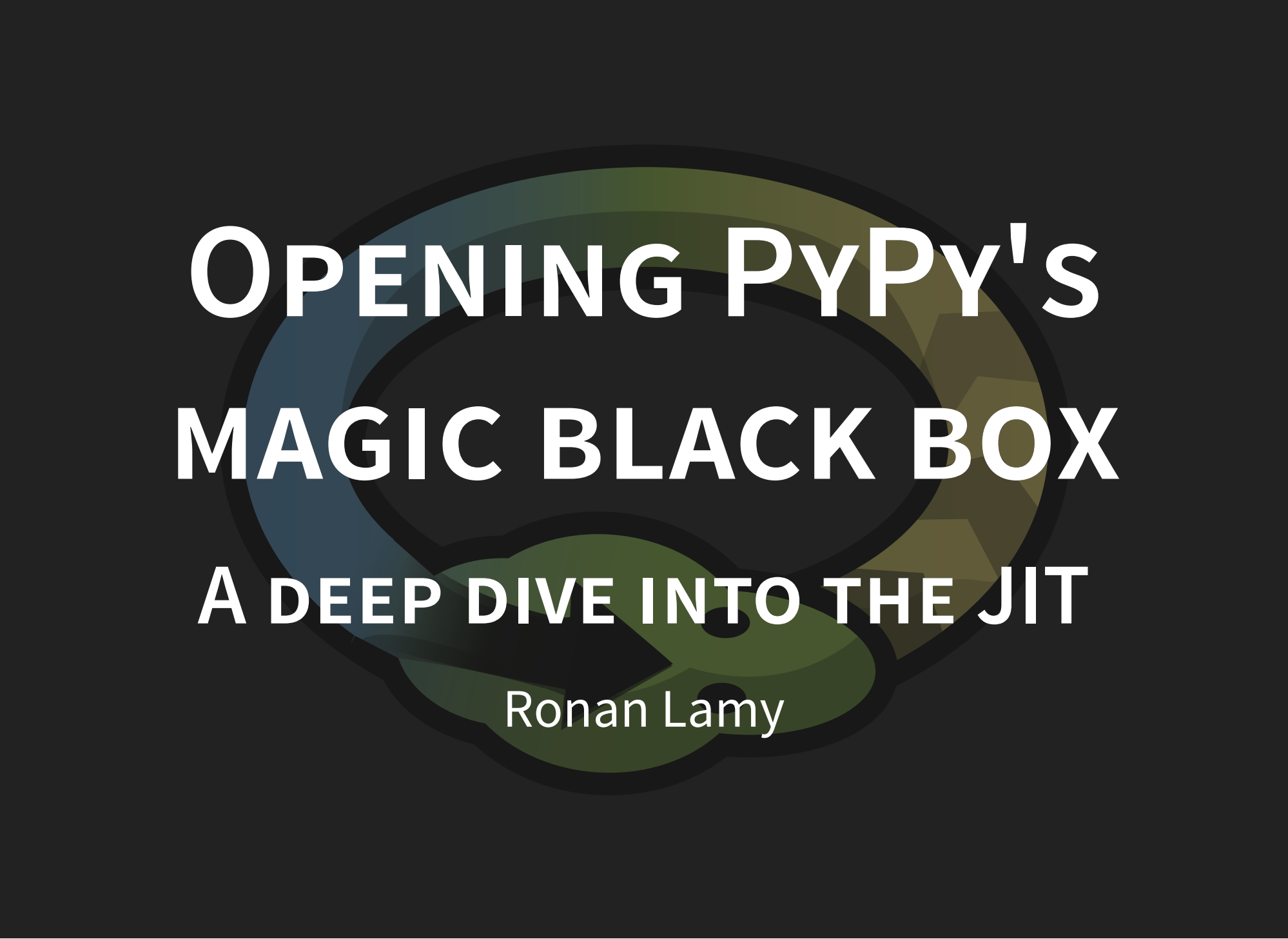


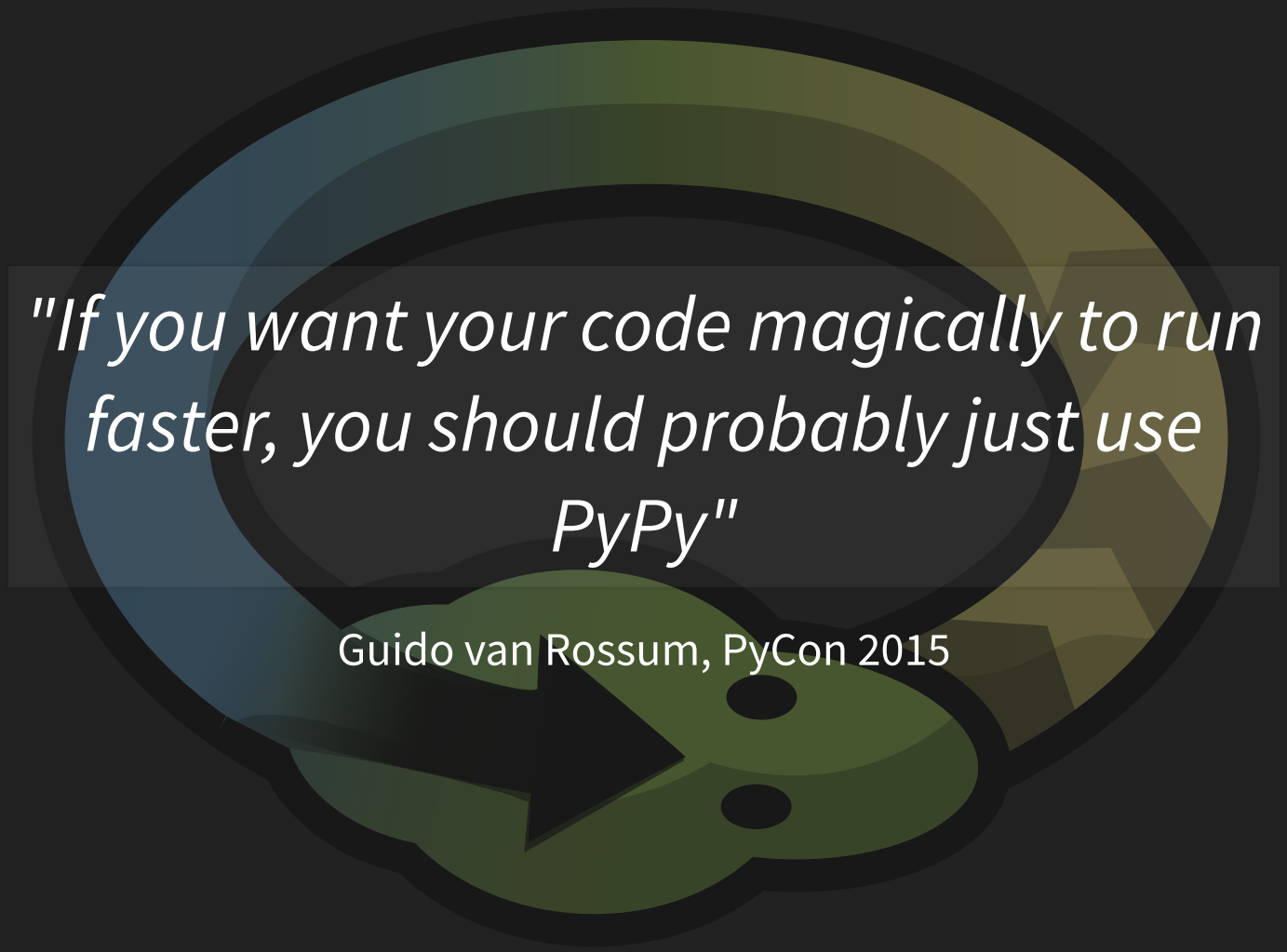
Opening PyPy's magic black box

A deep dive into the JIT

Ronan Lamy

About me

- PyPy core dev
- Python consultant and freelance developer
- Contact:
  - Ronan.Lamy@gmail.com
  - @ronanlamy



"If you want your code magically to run faster, you should probably just use PyPy"

Guido van Rossum, PyCon 2015

PyPy status

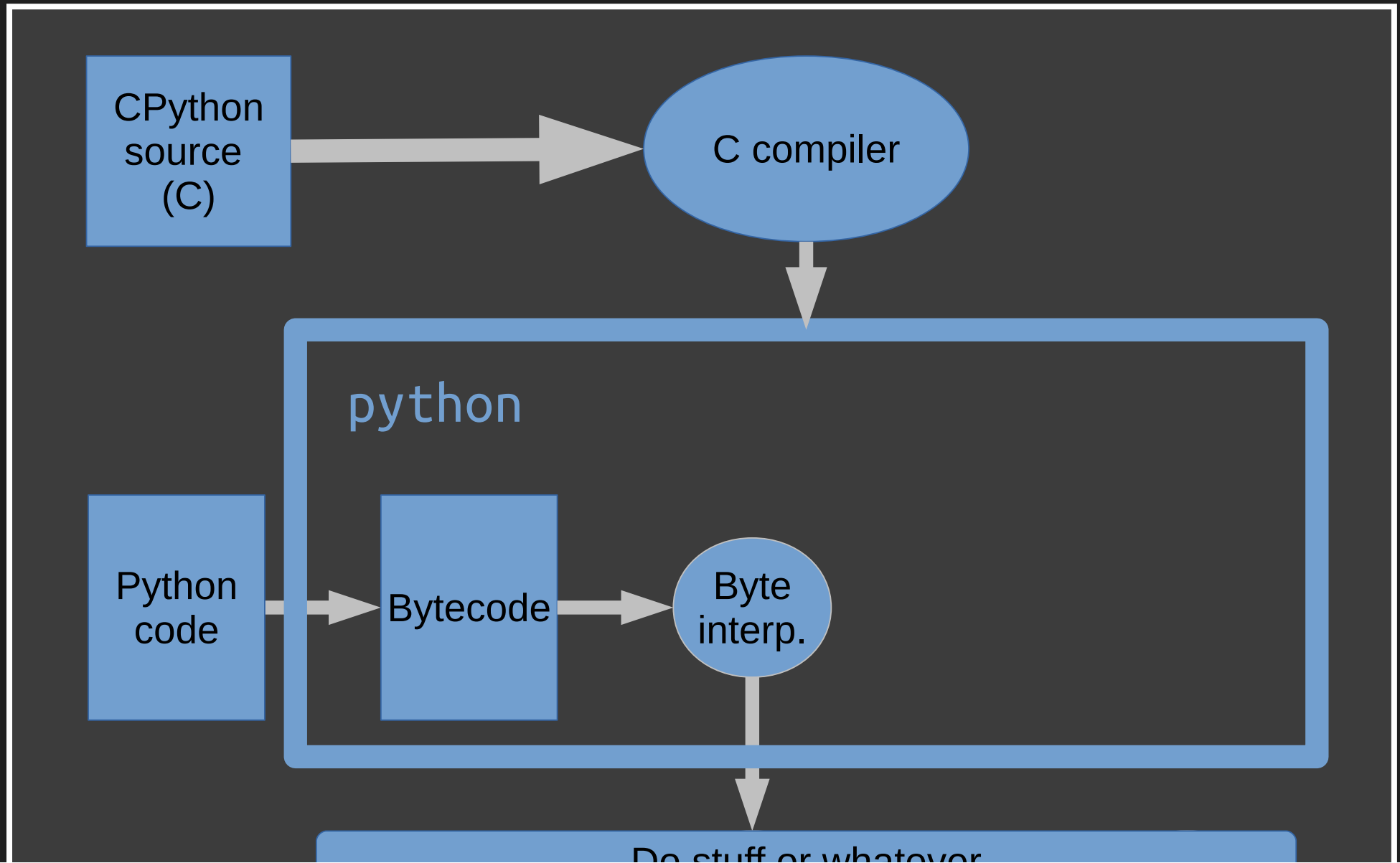
- Fast and compliant implementation of Python
- Full support for 2.7
- Beta support for 3.6 (full release soonish)
- Fast! (1x to 100x faster than CPython)
- cffi: fast and convenient interface to C code

C extension support

- numpy, scipy, pandas, scikit-learn, lxml, ...
- Cython + most extensions written in Cython
- 'pip install' works
- Wheels available at

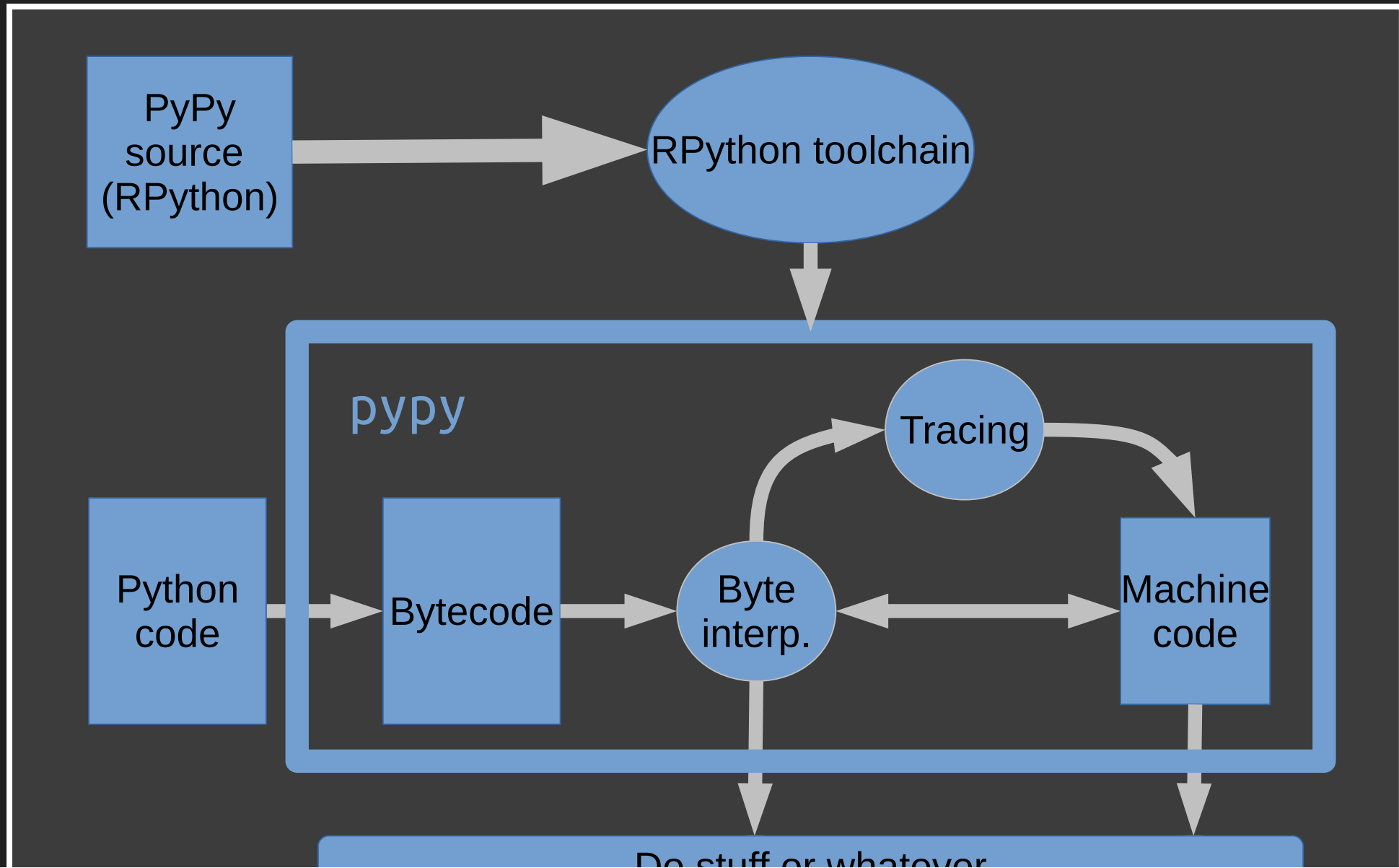
```
https://github.com/antocuni/pypy-wheels
```

CPython architecture



Do stuff or whatever

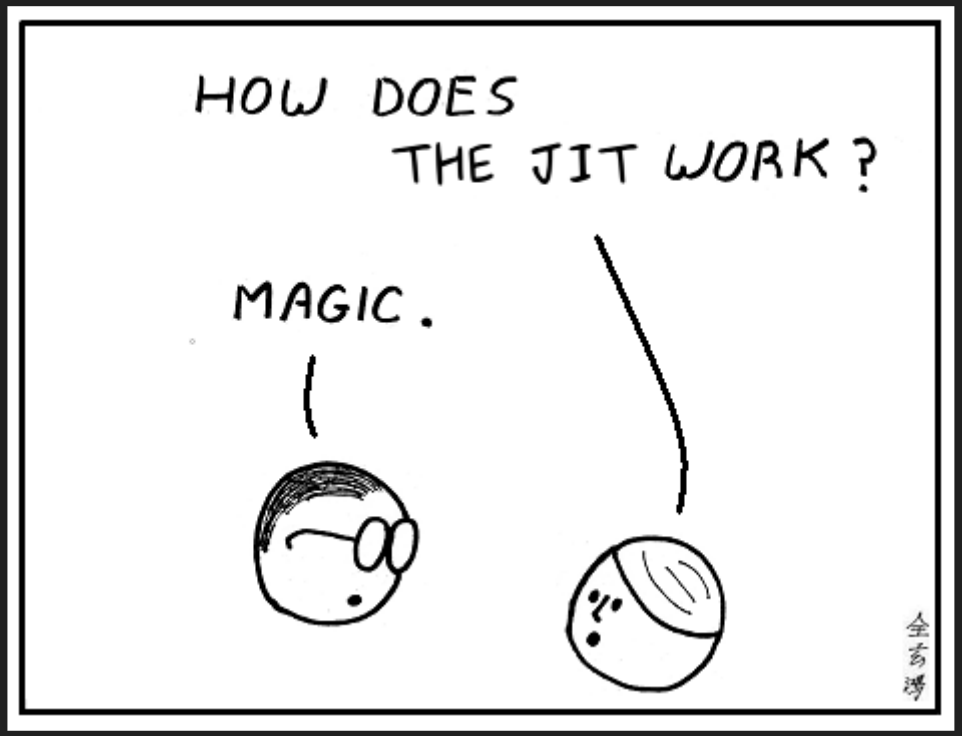
PyPy architecture



Do stuff or whatever

RPython translation stages

- RPython code
- `import`
  - Python objects (functions, classes, ...)
- Bytecode analysis, type inference
  - Typed control flow graphs
- Add GC and JIT
- Generate C code
- `gcc`
- Compiled executable


HOW DOES THE JIT WORK?
MAGIC.
?!
金玄湧

Maciej Fijałkowski @fijall · Jul 4

If you spend enough time with it, any magic is just careful and clever putting bits together.

1 1 6

Example

```
class Quantity:
    def __init__(self, value, unit):
        self.value = value
        self.unit = unit

    def __add__(self, other):
        if isinstance(other, Quantity):
            if other.unit != self.unit:
                raise ValueError("units must match")
            else:
                return Quantity(self.value + other.value, self.unit)
        else:
            return NotImplemented

    def __str__(self):
        return f"{self.value} {self.unit}"

def compute(n):
    total = Quantity(0, 'm')
    increment = Quantity(1., 'm')
```

Demo

INPLACE_ADD (simplified)

```
def INPLACE_ADD(self, *ignored):
    w_2 = self.popvalue()
    w_1 = self.popvalue()
    w_result = self.space.inplace_add(w_1, w_2)
    self.pushvalue(w_result)

def inplace_add(space, w_lhs, w_rhs):
    w_impl = space.lookup(w_lhs, '__iadd__')
    if w_impl is not None:
        # cpython bug-to-bug compatibility:
        if (space.type(w_lhs).flag_sequence_bug_compat
                        and not space.type(w_rhs).flag_sequence_bug_compat):
            w_res = _invoke_binop(space, space.lookup(w_rhs, '__radd__'),
                                  w_rhs, w_lhs)
            if w_res is not None:
                return w_res
        w_res = space.get_and_call_function(w_impl, w_lhs, w_rhs)
        if _check_notimplemented(space, w_res):
            return w_res
    return space.add(w_lhs, w_rhs)
```

Tracing JIT principles

- Pareto principle
  - 80% of the time is spent in 20% of the code
- Most branches are very imbalanced

Tracing JIT principles

- Pareto principle
  - 80% of the time is spent in 20% of the code
- Most branches are very imbalanced
- Compile only hot loops
- Optimise for the fast path
- Take advantage of run-time information

Tracing JIT principles

- Pareto principle
  - 80% of the time is spent in 20% of the code
- Most branches are very imbalanced
- Compile only hot loops
- Optimise for the fast path
- Take advantage of run-time information
- Trace = record one iteration of the loop
- Optimise trace and add guards
- Trace the interpreter, not user code

Jitcodes

- RPython code contains JIT hints
  - JIT drivers
  - `@dont_look_inside`
  - `@elidable`
  - quasi-immutables
- Toolchain creates flowgraphs
- Flowgraphs serialised to JIT-friendly IR: jitcode
- jitcodes stored in binary

Tracing

- The Python interpreter runs on top of a tracing interpreter: the meta-interpreter
- Meta-interpreter executes jitcodes
- and records operations in SSA form.
- Inlines function calls, flattens loops, ...
- Program values labeled as constants or variables
- Tracing ends when loop is closed

Guards

- Guards ~ JIT-level assertions
- On failure, must resume normal execution
- Checked at runtime
- Examples: conditional branches, overflow, exceptions, ...
- If a guard fails often, compile a "bridge"

Guards

- Guards ~ JIT-level assertions
- On failure, must resume normal execution
- Checked at runtime
- Examples: conditional branches, overflow, exceptions, ...
- If a guard fails often, compile a "bridge"
- Out-of-line guards: invalidate the whole trace
  - Zero run-time cost!

VMPROF

- Statistical profiler for CPython and PyPy
- Visualise JIT traces
- vmprof client records profile and JIT information
- Server renders logs

Demo

Optimisations

- Strength reduction
- intbounds
- Constant-folding
- strings
- Remove extra guards
- Virtuals and virtualisables

Unrolling

- Compute invariants
- First iteration: preamble
- Second iteration: tight loop

Backends

- x86, x86_64, PowerPC, S390x, ARMv7, ARM64 (in progress)
- GC has to be informed of dynamic allocations
- Linear register allocator
- Hand-written assembly for each operation

```
def genop_float_add(self, op, arglocs, result_loc):
    self.mc.ADDSD(arglocs[0], arglocs[1])
```

Demo

Summary

- Be wary of microbenchmarks!
- RPython toolchain has a generic JIT framework
- PyPy interpreter exploits JIT hints
- Abstractions for free!

Contact

- IRC: #pypy on Freenode IRC
- http://pypy.org
- pypy-dev @ python.org

- PyPy help desk Friday morning
- Sprint Saturday and Sunday

- Questions?

The end