Who's Afraid of the Big Bad ceval Loop?

Me

Moshe Zadka – https://cobordism.com

2020

What is not the eval loop?

- Parser
- Byte compiler
- Object semantics

The CPython interpreter, as its name implies, is writen in C. It contains several parts, which we will not delve into today. It has the parser and byte compiler, which produce Python byte code. It also has the implementation of native objects, such as integers as strings, as C "extension classes".

The eval loop does not do any parsing or byte-code compilation. Those steps must have been done before: usually, Python will do them for us (although we can force it to execute on a sequence of byte codes). It also does not have any of Python's *object* semantics: it does not know that one integer divided by another must sometimes result in a floating point number, or that strings are immutable.

The semantics of Python's *objects* are part of the class definitions: this is where integer division or string modification (resulting in an error) are implemented.

What is the eval loop?

```
for (;;) {
    /* ... */
    /* This is a lie */
    opcode = (*next_instr) & 0xff;
    oparg = (*next_instr) >> 8;
    next_instr++;
    /* ... */
    switch (opcode) {
        /* ... */
    }
}
```

In the middle between those parts, something must *actually execute* the byte-code. When we have a sequence of byte codes, something must inspect the code, decide which operation to do, and do it.

As this description might imply, this is one of the most performance critical parts in the CPython interpreter. The eval loop has seen many generations of iterations, and has evolved together with CPU architecture. This is the reason for the white lie above: the actual code is a few layers of macros, including an ifdef based on the machine architecture.

On lying

Everybody lies.

The code inside CPython is the result of, literally, decades of optimization. You are, hopefully, grateful for the optimizations every time you run your Python code and it finishes quickly. But those optimizations do not come cheap: among other things, they require non-trivial coding techniques. These often obscure the actual algorithm, and make the code hard to read.

This is a great trade-off for an interpreter! But it is not a great trade-off for a talk: my goal is for you to understand this, not to be able to run it in your head faster.

So I have decided to take a somewhat extreme solution: I intend to lie through my teeth. Most of the code examples I will show to you do not appear in the interpreter. However, they are more-or-less equivalent, and get the point across. They are, for the most part, the first draft you would write if you were to reimplement CPython, before starting to profile and optimize.

Code objects

```
>>> (lambda x:None).__code__
<code object <lambda> at 0x7fb7b67725d0, file "<stdin>", line 1>
```

Code objects are usually found in functions, as the `__code__` attribute.

Python bytecode

```
>>> (lambda a, b: a + b
... ).__code__.co_code
b'|\x00|\x01\x17\x00S\x00'
>>> dis.dis(_)
              0 LOAD_FAST                0 (0)
              2 LOAD_FAST                1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

Python bytecode are... bytes. They come in sequence of two: a code, and then an argument. For codes that do not need an argument, the argument will be NIL, the 0 byte.

Frame

Where the eval happens

Python bytecode always happens in the context of a *frame*. A frame gives context to code. Calling a function will *create* a frame. Naturally, distinct calls to the function will generate distinct frames. This is how recursion can work.

Frames are explicit, though not documented, in Python. The current frame can always be retrieved by `sys._getframe()`. While it should never be done in "real" code, it is a great exercise to play with it. It also serves as a useful thing in debuggers and other coding tools.

Frame contents

Code and context

Since code evaluation always happen in the context of a *frame*, the frame holds the code. It also holds several links to namespaces: the locals, the globals, and the builtins. Finally, frames are *nested*: a frame will have a link to the "outer", or "back", frame.

Frame inspection

```
>>> def foo(): bar()
...
>>> def bar():
...     global frm;frm = sys._getframe()
...
>>> frm.f_code.co_name
'bar'
>>> frm.f_back.f_code.co_name
'foo'
>>> frm.f_back.f_back.f_code.co_name
'<module>'
```

The frames have a link to the "back frame", where a "return" statement or a "yield" statement would go. This explicit structure of frames is an essential part of allowing generators and other more complicated control structures. They also have a link to the code object. Frames also have an `f_lasti` property, which contains an index to the last instruction. This is the variable that the eval loop increments.

Stack machine

```
PyObject **stack_pointer;
/* ... */
#define BASIC_PUSH(v)     (*stack_pointer++ = (v))
#define BASIC_POP()       (*--stack_pointer)
```

The basic semantics of Python's byte code are those of a stack machine. The operations all operate on a stack: it can be pushed or popped, and operations happen on the stack. The stack space is part of the frame. This means that a frame suspended, for example when yielding, still has the stack.

Stack operations

```
#define TOP()             (stack_pointer[-1])
#define PEEK(n)           (stack_pointer[-(n)])
#define SET_TOP(v)        (stack_pointer[-1] = (v))
```

The Python implementation defines some macros for highly efficient stack operations. This is important, since much of what Python does involves manipulatig the stack.

The Loop arguments

```
PyObject *_PyEval_EvalFrameDefault(
    PyFrameObject *f,
    int throwflag)
```

These is the function definition of the Python eval. It accepts a frame object as an input: as we saw earlier, the frame object has all the important things: the code, the stack, the instruction pointer, etc.

The Loop

```
    /* ... */
    next_instr = f->f_code->co_code + f->f_lasti;
    for (;;) {
        /* ... */
    fast_next_opcode:
        opcode = (*next_instr) & 0xff;
        oparg = (*next_instr) >> 8;
        next_instr++;
        switch (opcode) {
            /* ... */
        }
    }
    error:
        /* ... */
```

This is the Python eval loop. We initialize the insturction pointer to the current place based on the frame (in case it needs to be resumed). The lies are because I simplified the code, and made it less efficient, but equivalent. Python is highly optimized, but that can obscure the semantics.

The Switch

```
switch (opcode) {
        /* ... */
        case TARGET(LOAD_FAST): {
            PyObject *value = GETLOCAL(oparg);
            if (value == NULL) {
                /* ... */
                goto error;
            }
            /* ... */
            goto fast_next_opcode;
        }
        /* ... */
        case TARGET(UNARY_NEGATIVE): {
            /* ... */
```

```
            continue;
        }
        /* ... */
}
```

Every instruction handle finishes with one of three: either it goes to an error, or it does a "fast" or "slow" dispatch. "Fast" dispatches are those that skip various checks: for example, thread switches cannot happen between a "fast" dispatch and the one that follows.

Eval Breaker

```
ceval.eval_breaker =
    ceval.gil_drop_request |
    ceval.signals_pending |
    ceval.pending.calls_to_do) |
    ceval.pending.async_exc;
```

These are lies! The actual code is slightly more complicated since it has to do this computation while carefully considering signals and threads, so it uses all kinds of thread-safe primitives. However, the actual computation is the same.

Eval Breaker

```
if (eval_breaker) {
    if (ceval.signals_pending)
        handle_signals(runtime);
    if (ceval.pending.calls_to_do)
        make_pending_calls(runtime);
    if (ceval.gil_drop_request) {
        drop_gil(ceval, tstate);
        /* Other threads may run now */
        take_gil(ceval, tstate);
    }
    if (tstate->async_exc != NULL) {
        _PyErr_SetNone(tstate, tstate->async_exc);
        goto error;
    }
}
```

As always, these are a bunch of dirty lies! The actual code has a bunch of stuff for thread-safety, and error handling. It also skips eval-breakage in the presence of some opcodes that would make handling signals cause issues: we delay until we get to a nice, regular opcode.

Signal handling

```
void trip_signal(int sig_num) {
     /* ... */
    PyRuntimeState *runtime = &_PyRuntime;
    PyThreadState *tstate = _PyRuntimeState_GetThreadState(runtime);
    runtime->ceval.signals_pending = 1
    runtime->ceval.eval_breaker = 1
    /* ... */
}
```

This is a simplified version of Python's signal handler. It sets the signals pending and eval breaker flags, so that eventually, the loop will handle the signals.

Signals are called in a "safe" place Python-wise, but still somewhat arbitrary: in theory, we could be inside the evaluation of Python code inside an `__del__` that was called because a list index was set.

Global Interpreter Lock

- Python bytecode evaluation

The global interpreter lock, on which many reams of ink have been spilled, makes sure that Python can only evaluate Python bytecode on only one thread. This means that the beginning of the ceval loop, somewhere at the top, makes sure we have the lock.

However, as we saw, every few calls, we do "eval breaker" and do a drop/take of the GIL. This means that if another thread is waiting in its "take" GIL then the drop will give it a chance to capture it.

GIL: Acquisition

```
MUTEX_LOCK(gil->mutex);
while (gil.locked)
    COND_TIMED_WAIT(gil->cond, gil->mutex, interval, timed_out);
    ceval.gil_drop_request = 1;
    ceval.eval_breaker = 1;
}
MUTEX_UNLOCK(gil->mutex);
MUTEX_LOCK(gil->switch_mutex);
gil.locked = 1
MUTEX_UNLOCK(gil->switch_mutex);
```

This loop is the logic for acquiring the GIL. We lock the mutex, and then loop waiting for the condition variable to fire, and the GIL to become unlocked. Once it is unlocked, we lock the GIL ourselves, and we are done.

GIL: Release

```
MUTEX_LOCK(gil->mutex);
gil.locked = 0;
COND_SIGNAL(gil->cond);
MUTEX_UNLOCK(gil->mutex);
```

Releasing the GIL is much more straightforward: we do not have to wait, we just set it to unlocked, and fire the condition variable.

Takeaways

- Python is not magic
- ...although it is definitely sufficiently advanced.

CPython, at the end of the day, is just a program written in C. It is possible to figure what is going on there. That said, remember that almost all code examples shown here were bold-faced lies, and the actual code pulls a lot of tricks to eke out performance.