Python behind the scenes #4: how Python bytecode is executed

Published: Fri 30 October 2020
By *Victor Skvortsov*
tags: Python behind the scenes Python CPython

We started this series with an overview of the CPython VM. We learned that to run a Python program, CPython first compiles it to bytecode, and we studied how the compiler works in part two. Last time we stepped through the CPython source code starting with the `main()` function until we reached the evaluation loop, a place where Python bytecode gets executed. The main reason why we spent time studying these things was to prepare for the discussion that we start today. The goal of this discussion is to understand how CPython does what we tell it to do, that is, how it executes the bytecode to which the code we write compiles.

Note: In this post I'm referring to CPython 3.9. Some implementation details will certainly change as CPython evolves. I'll try to keep track of important changes and add update notes.

Starting point

Let's briefly recall what we learned in the previous parts. We tell CPython what to do by writing Python code. The CPython VM, however, understands only Python bytecode. This is the job of the compiler to translate Python code to bytecode. The compiler stores bytecode in a code object, which is a structure that fully describes what a code block, like a module or a function, does. To execute a code object, CPython first creates a state of execution for it called a frame object. Then it passes a frame object to a frame evaluation function to perform the actual computation. The default frame evaluation function is `_PyEval_EvalFrameDefault()` defined in Python/ceval.c. This function implements the core of the CPython VM. Namely, it implements the logic for the execution of Python bytecode. So, this function is what we're going to study today.

To understand how `_PyEval_EvalFrameDefault()` works, it is crucial to have an idea of what its input, a frame object, is. A frame object is a Python object defined by the following C struct:

```
// typedef struct _frame PyFrameObject; in other place
struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;      /* previous frame, or NULL */
    PyCodeObject *f_code;       /* code segment */
    PyObject *f_builtins;       /* builtin symbol table (PyDictObject) */
    PyObject *f_globals;        /* global symbol table (PyDictObject) */
    PyObject *f_locals;         /* local symbol table (any mapping) */
    PyObject **f_valuestack;    /* points after the last local */
    /* Next free slot in f_valuestack.  Frame creation sets to f_valuestack.
       Frame evaluation usually NULLs it, but a frame that yields sets it
       to the current stack top. */
    PyObject **f_stacktop;
    PyObject *f_trace;          /* Trace function */
    char f_trace_lines;         /* Emit per-line trace events? */
```

```
    char f_trace_opcodes;       /* Emit per-opcode trace events? */

    /* Borrowed reference to a generator, or NULL */
    PyObject *f_gen;

    int f_lasti;                /* Last instruction if called */
    int f_lineno;               /* Current line number */
    int f_iblock;               /* index in f_blockstack */
    char f_executing;           /* whether the frame is still executing */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    PyObject *f_localsplus[1];  /* locals+stack, dynamically sized */
};
```

The `f_code` field of a frame object points to a code object. A code object is also a Python object. Here's its definition:

```
struct PyCodeObject {
    PyObject_HEAD
    int co_argcount;            /* #arguments, except *args */
    int co_posonlyargcount;     /* #positional only arguments */
    int co_kwonlyargcount;      /* #keyword only arguments */
    int co_nlocals;             /* #local variables */
    int co_stacksize;           /* #entries needed for evaluation stack */
    int co_flags;               /* CO_..., see below */
    int co_firstlineno;         /* first source line number */
    PyObject *co_code;          /* instruction opcodes */
    PyObject *co_consts;        /* list (constants used) */
    PyObject *co_names;         /* list of strings (names used) */
    PyObject *co_varnames;      /* tuple of strings (local variable names) */
    PyObject *co_freevars;      /* tuple of strings (free variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable names) */
    /* The rest aren't used in either hash or comparisons, except for co_name,
       used in both. This is done to preserve the name and line number
       for tracebacks and debuggers; otherwise, constant de-duplication
       would collapse identical functions/lambdas defined on different lines.
    */
    Py_ssize_t *co_cell2arg;    /* Maps cell vars which are arguments. */
    PyObject *co_filename;      /* unicode (where it was loaded from) */
    PyObject *co_name;          /* unicode (name, for reference) */
    PyObject *co_lnotab;        /* string (encoding addr<->lineno mapping) See
                                   Objects/lnotab_notes.txt for details. */
    void *co_zombieframe;       /* for optimization only (see frameobject.c) */
    PyObject *co_weakreflist;   /* to support weakrefs to code objects */
    /* Scratch space for extra data relating to the code object.
       Type is a void* to keep the format private in codeobject.c to force
       people to go through the proper APIs. */
    void *co_extra;

    /* Per opcodes just-in-time cache
     *
     * To reduce cache size, we use indirect mapping from opcode index to
     * cache object:
     *   cache = co_opcache[co_opcache_map[next_instr - first_instr] - 1]
     */

    // co_opcache_map is indexed by (next_instr - first_instr).
    //  * 0 means there is no cache for this opcode.
    //  * n > 0 means there is cache in co_opcache[n-1].
    unsigned char *co_opcache_map;
    _PyOpcache *co_opcache;
    int co_opcache_flag;  // used to determine when create a cache.
    unsigned char co_opcache_size;  // length of co_opcache.
};
```

The most important field of a code object is `co_code`. It's a pointer to a Python bytes object representing the bytecode. The bytecode is a sequence of two-byte instructions: one byte for an opcode and one byte for an argument.

Don't worry if some members of the above structures are still a mystery to you. We'll see what they are used for as we move forward in our attempt to understand how the CPython VM executes the bytecode.

Overview of the evaluation loop

The problem of executing Python bytecode may seem a no-brainer to you. Indeed, all the VM has to do is to iterate over the instructions and to act according to them. And this is what essentially `_PyEval_EvalFrameDefault()` does. It contains an infinite `for (;;)` loop that we refer to as the evaluation loop. Inside that loop there is a giant `switch` statement over all possible opcodes. Each opcode has a corresponding `case` block containing the code for executing that opcode. The bytecode is represented by an array of 16-bit unsigned integers, one integer per instruction. The VM keeps track of the next instruction to be executed using the `next_instr` variable, which is a pointer to the array of instructions. At the start of each iteration of the evaluation loop, the VM calculates the next opcode and its argument by taking the least significant and the most significant byte of the next instruction respectively and increments `next_instr`. The `_PyEval_EvalFrameDefault()` function is nearly 3000 lines long, but its essence can be captured by the following simplified version:

```
PyObject*
_PyEval_EvalFrameDefault(PyThreadState *tstate, PyFrameObject *f, int throwflag)
{
    // ... declarations and initialization of local variables
    // ... macros definitions
    // ... call depth handling
    // ... code for tracing and profiling

    for (;;) {
        // ... check if the bytecode execution must be suspended,
        // e.g. other thread requested the GIL

        // NEXTOPARG() macro
        _Py_CODEUNIT word = *next_instr; // _Py_CODEUNIT is a typedef for uint16_t
        opcode = _Py_OPCODE(word);
        oparg = _Py_OPARG(word);
        next_instr++;

        switch (opcode) {
            case TARGET(NOP) {
                FAST_DISPATCH(); // more on this later
            }

            case TARGET(LOAD_FAST) {
                // ... code for loading local variable
            }

            // ... 117 more cases for every possible opcode
        }

        // ... error handling
    }

    // ... termination
}
```

To get a more realistic picture, let's discuss some of the omitted pieces in more detail.

reasons to suspend the loop

From time to time, the currently running thread stops executing the bytecode to do something else or to do nothing. This can happen due to one of the four reasons:

- There are signals to handle. When you register a function as a signal handler using `signal.signal()`, CPython stores this function in the array of handlers. The function that will actually be called when a thread receives a signal is `signal_handler()` (it's passed to the `sigaction()` library function on Unix-like systems). When called, `signal_handler()` sets a boolean variable telling that the function in the array of handlers corresponding to the received signal has to be called. Periodically, the main thread of the main interpreter calls the tripped handlers.
- There are pending calls to call. Pending calls is a mechanism that allows to schedule a function to be executed from the main thread. This mechanism is exposed by the Python/C API via the `Py_AddPendingCall()` function.
- The asynchronous exception is raised. The asynchronous exception is an exception set in one thread from another. This can be done using the `PyThreadState_SetAsyncExc()` function provided by the Python/C API.
- The currently running thread is requested to drop the GIL. When it sees such a request, it drops the GIL and waits until it acquires the GIL again.

CPython has indicators for each of these events. The variable indicating that there are handlers to call is a member of `runtime->ceval`, which is a `_ceval_runtime_state` struct:

```
struct _ceval_runtime_state {
    /* Request for checking signals. It is shared by all interpreters (see
       bpo-40513). Any thread of any interpreter can receive a signal, but only
       the main thread of the main interpreter can handle signals: see
       _Py_ThreadCanHandleSignals(). */
    _Py_atomic_int signals_pending;
    struct _gil_runtime_state gil;
};
```

Other indicators are members of `interp->ceval`, which is a `_ceval_state` struct:

```
struct _ceval_state {
    int recursion_limit;
    /* Records whether tracing is on for any thread.  Counts the number
       of threads for which tstate->c_tracefunc is non-NULL, so if the
       value is 0, we know we don't have to check this thread's
       c_tracefunc.  This speeds up the if statement in
       _PyEval_EvalFrameDefault() after fast_next_opcode. */
    int tracing_possible;
    /* This single variable consolidates all requests to break out of
       the fast path in the eval loop. */
    _Py_atomic_int eval_breaker;
    /* Request for dropping the GIL */
    _Py_atomic_int gil_drop_request;
    struct _pending_calls pending;
};
```

The result of ORing all indicators together is stored in the `eval_breaker` variable. It tells whether there is any reason for the currently running thread to stop its normal bytecode execution. Each iteration of the evaluation loop starts with the check whether `eval_breaker` is true. If it is true, the thread checks the indicators to determine what exactly it is asked to do, does that and continues to execute the bytecode.

computed GOTOs

The code for the evaluation loop is full of macros such as `TARGET()` and `DISPATCH()`. These are not the means to make the code more compact. They expand to different code depending on whether the certain optimization, referred to as "computed GOTOs" (a.k.a. "threaded code"), is used. They goal of this optimization is to speed up the bytecode

execution by writing code in such a way, so that a CPU can use its branch prediction mechanism to predict the next opcode.

After executing any given instruction, the VM does one of the three things:

- It returns from the evaluation function. This happens when the VM executes `RETURN_VALUE`, `YIELD_VALUE` or `YIELD_FROM` instruction.
- It handles the error and either continues the execution or returns from the evaluation function with the exception set. The error can occur when, for example, the VM executes the `BINARY_ADD` instruction and the objects to be added do not implement `__add__` and `__radd__` methods.
- It continues the execution. How to make the VM execute the next instruction? The simplest solution would be to end each non-returning `case` block with the `continue` statement. The real solution, though, is a little bit more complicated.

To see the problem with the simple `continue` statement, we need to understand what `switch` compiles to. An opcode is an integer between 0 and 255. Because the range is dense, the compiler can create a jump table that stores addresses of the `case` blocks and use opcodes as indices into that table. The modern compilers indeed do that, so the dispatching of cases is effectively implemented as a single indirect jump. This is an efficient way to implement `switch`. However, placing `switch` inside the loop and adding `continue` statements creates two inefficiencies:

- The `continue` statement in the end of a `case` block adds another jump. Thus, to execute an opcode, the VM has to jump twice: to the start of the loop and then to the next `case` block.
- Since all opcodes are dispatched by a single jump, a CPU has a little chance of predicting the next opcode. The best it can do is to choose the last opcode or, possibly, the most frequent one.

The idea of the optimization is to place a separate dispatch jump in the end of each non-returning `case` block. First, it saves a jump. Second, a CPU can predict the next opcode as the most probable opcode following the current one.

The optimization can be enabled or disabled. It depends on whether the compiler supports the GCC C extension called "labels as values" or not. The effect of enabling the optimization is that the certain macros, such as `TARGET()`, `DISPATCH()` and `FAST_DISPATCH()`, expand in different way. These macros are used extensively throughout the code of the evaluation loop. Every case expression has a form `TARGET(op)`, where `op` is a macro for the integer literal representing an opcode. And every non-returning `case` block ends with `DISPATCH()` or `FAST_DISPATCH()` macro. Let's first look at what these macros expand to when the optimization is disabled:

```
for (;;) {
    // ... check if the bytecode execution must be suspended

fast_next_opcode:
    // NEXTOPARG() macro
    _Py_CODEUNIT word = *next_instr;
    opcode = _Py_OPCODE(word);
    oparg = _Py_OPARG(word);
    next_instr++;

    switch (opcode) {
        // TARGET(NOP) expands to NOP
        case NOP: {
            goto fast_next_opcode; // FAST_DISPATCH() macro
        }

        // ...

        case BINARY_MULTIPLY: {
            // ... code for binary multiplication
            continue; // DISPATCH() macro
```

```
            }

            // ...
        }

        // ... error handling
    }
```

The `FAST_DISPATCH()` macro is used for some opcode when it's undesirable to suspend the evaluation loop after executing that opcode. Otherwise, the implementation is very straightforward.

If the compiler supports "labels as values" extension, we can use the unary `&&` operator on a label to get its address. It has a value of type `void *`, so we can store it in a pointer:

```
void *ptr = &&my_label;
```

We can then go to the label by dereferencing the pointer:

```
goto *ptr;
```

This extension allows to implement a jump table in C as an array of label pointers. And that's what CPython does:

```
static void *opcode_targets[256] = {
    &&_unknown_opcode,
    &&TARGET_POP_TOP,
    &&TARGET_ROT_TWO,
    &&TARGET_ROT_THREE,
    &&TARGET_DUP_TOP,
    &&TARGET_DUP_TOP_TWO,
    &&TARGET_ROT_FOUR,
    &&_unknown_opcode,
    &&_unknown_opcode,
    &&TARGET_NOP,
    &&TARGET_UNARY_POSITIVE,
    &&TARGET_UNARY_NEGATIVE,
    &&TARGET_UNARY_NOT,
    // ... quite a few more
};
```

Here's what the optimized version of the evaluation loop looks like:

```
for (;;) {
    // ... check if the bytecode execution must be suspended

fast_next_opcode:
    // NEXTOPARG() macro
    _Py_CODEUNIT word = *next_instr;
    opcode = _Py_OPCODE(word);
    oparg = _Py_OPARG(word);
    next_instr++;

    switch (opcode) {
        // TARGET(NOP) expands to NOP: TARGET_NOP:
        // TARGET_NOP is a label
        case NOP: TARGET_NOP: {
            // FAST_DISPATCH() macro
            // when tracing is disabled
            f->f_lasti = INSTR_OFFSET();
            NEXTOPARG();
            goto *opcode_targets[opcode];
```

```
            }

            // ...

            case BINARY_MULTIPLY: TARGET_BINARY_MULTIPLY: {
                // ... code for binary multiplication
                // DISPATCH() macro
                if (!_Py_atomic_load_relaxed(eval_breaker)) {
                  FAST_DISPATCH();
                }
                continue;
            }

            // ...
        }

        // ... error handling
    }
```

The extension is supported by the GCC and Clang compilers. So, when you run `python`, you probably have the optimization enabled. The question, of course, is how it affects the performance. Here, I'll rely on the comment from the source code:

> *At the time of this writing, the "threaded code" version is up to 15-20% faster than the normal "switch" version, depending on the compiler and the CPU architecture.*

This section should give us an idea of how the CPython VM goes from one instruction to the next and what it may do in between. The next logical step is to study in more depth how the VM executes a single instruction. CPython 3.9 has 119 different opcodes. Of course, we're not going to study the implementation of each opcode in this post. Instead, we'll focus on the general principles that the VM uses to execute them.

Value stack

The most important and, fortunately, very simple fact about the CPython VM is that it's stack-based. This means that to compute things, the VM pops (or peeks) values from the stack, performs the computation on them and pushes the result back. Here's some examples:

- The `UNARY_NEGATIVE` opcode pops value from the stack, negates it and pushes the result.
- The `GET_ITER` opcode pops value from the stack, calls `iter()` on it and pushes the result.
- The `BINARY_ADD` opcode pops value from the stack, peeks another value from the top, adds the first value to the second and replaces the top value with the result.

The value stack resides in a frame object. It's implemented as a part of the array called `f_localsplus`. The array is split into several parts to store different things, but only the last part is used for the value stack. The start of this part is the bottom of the stack. The `f_valuestack` field of a frame object points to it. To locate the top of the stack, CPython keeps the `stack_pointer` local variable, which points to the next slot after the top of the stack. The elements of the `f_localsplus` array are pointers to Python objects, and pointers to Python objects is what the CPython VM actually works with.

Error handling and block stack

Not all computations performed by the VM are successful. Suppose we try to add a number to a string like `1 + '41'`. The compiler produces the `BINARY_ADD` opcode to add two objects. When the VM executes this opcode, it calls `PyNumber_Add()` to calculate the result:

```
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    // ... special case of string addition
    sum = PyNumber_Add(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

What's important for us now is not how `PyNumber_Add()` is implemented, but that the call to it results in an error. The error means two things:

- `PyNumber_Add()` returns `NULL`.
- `PyNumber_Add()` sets the current exception to the `TypeError` exception. This involves setting `tstate->curexc_type`, `tstate->curexc_value` and `tstate->curexc_traceback`.

`NULL` is an indicator for an error. The VM sees it and goes to the `error` label at the end of the evaluation loop. What happens next depends on whether we've set up any exception handlers or not. If we haven't, the VM reaches the `break` statement and the evaluation function returns `NULL` with the exception set on the thread state. CPython prints the details of the exception and exits. We get the expected result:

```
$ python -c "1 + '42'"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

But suppose that we place the same code inside the `try` clause of the `try-finally` statement. In this case, the code inside the `finally` clause is executed as well:

```
$ python -q
>>> try:
...     1 + '41'
... finally:
...     print('Hey!')
...
Hey!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

How can the VM continue the execution after the error has occurred? Let's look at the bytecode produced by the compiler for the `try-finally` statement:

```
$ python -m dis try-finally.py

  1           0 SETUP_FINALLY           20 (to 22)

  2           2 LOAD_CONST               0 (1)
              4 LOAD_CONST               1 ('41')
              6 BINARY_ADD
              8 POP_TOP
             10 POP_BLOCK
```

```
  4          12 LOAD_NAME                0 (print)
             14 LOAD_CONST               2 ('Hey!')
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 JUMP_FORWARD            10 (to 32)
        >>   22 LOAD_NAME                0 (print)
             24 LOAD_CONST               2 ('Hey!')
             26 CALL_FUNCTION            1
             28 POP_TOP
             30 RERAISE
        >>   32 LOAD_CONST               3 (None)
             34 RETURN_VALUE
```

Note the `SETUP_FINALLY` and `POP_BLOCK` opcodes. The first one sets up the exception handler and the second one removes it. If an error occurs while the VM executes the instructions between them, the execution continues with the instruction at offset 22, which is the start of the `finally` clause. Otherwise, the `finally` clause is executed after the `try` clause. In both cases, the bytecode for the `finally` clause is almost identical. The only difference is that the handler re-raises the exception set in the `try` clause.

An exception handler is implemented as a simple C struct called block:

```
typedef struct {
    int b_type;                 /* what kind of block this is */
    int b_handler;              /* where to jump to find handler */
    int b_level;                /* value stack level to pop to */
} PyTryBlock;
```

The VM keeps blocks in the block stack. To setup an exception handler means to push a new block onto the block stack. This is what opcodes like `SETUP_FINALLY` do. The `error` label points to a piece of code that tries to handle an error using blocks on the block stack. The VM unwinds the block stack until it finds the top-most block of type `SETUP_FINALLY`. It restores the level of the value stack to the level specified by the `b_level` field of the block and continues to execute the bytecode with the instruction at offset `b_handler`. This is basically how CPython implements statements like `try-except`, `try-finally` and `with`.

There is one more thing to say about exception handling. Think of what happens when an error occurs while the VM handles an exception:

```
$ python -q
>>> try:
...     1 + '41'
... except:
...     1/0
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ZeroDivisionError: division by zero
```

As expected, CPython prints the original exception. To implement such behavior, when CPython handles an exception using a `SETUP_FINALLY` block, it sets up another block of type `EXCEPT_HANDLER`. If an error occurs when a block of this type is on the block stack, the VM gets the original exception from the value stack and sets it as the current one. CPython used to have different kinds of blocks but now it's only `SETUP_FINALLY` and `EXCEPT_HANDLER`.

The block stack is implemented as the `f_blockstack` array in a frame object. The size of the array is statically defined to 20. So, if you nest more than 20 `try` clauses, you get `SyntaxError: too many statically nested blocks`.

Summary

Today we've learned that the CPython VM executes bytecode instructions one by one in an infinite loop. The loop contains a `switch` statement over all possible opcodes. Each opcode is executed in the corresponding `case` block. The evaluation function runs in a thread and sometimes that thread suspends the loop to do something else. For example, a thread may need to release the GIL, so that other thread can take it and continue to execute its bytecode. To speed up the bytecode execution, CPython employs an optimization that allows to make use of the CPU's branch prediction mechanism. A comment says that it makes CPython 15-20% faster.

We've also looked at two data structures crucial for the bytecode execution:

- the value stack that the VM uses to compute things; and
- the block stack that the VM uses to handle exceptions.

The most important conclusion from the post is this: if you want to study the implementation of some aspect of Python, the evaluation loop is a perfect place to start. Want to know what happens when you write `x + y`? Take a look at the code for the `BINARY_ADD` opcode. Want to know how the `with` statement is implemented? See `SETUP_WITH`. Interested in the exact semantics of a function call? The `CALL_FUNCTION` opcode is what you're looking for. We'll apply this method next time when study how variables are implemented in CPython.

If you have any questions, comments or suggestions, feel free to contact me at victor@tenthousandmeters.com

Update from 10 November 2020: I was pointed out in the comments on HN that the `UNARY_NEGATIVE` and `GET_ITER` opcodes peek the value on top of the stack and replace it with the result instead of popping the value and pushing the result. This is indeed so. Semantically, the two approaches are equivalent. The only difference is that the pop/push approach first decrements and then increments `stack_pointer`. CPython avoids these redundant operations.