Python behind the scenes #1: how the CPython VM works

Introduction

Published: Mon 31 August 2020
By *Victor Skvortsov*
tags: Python behind the scenes Python CPython

Have you ever wondered what `python` does when you run one of your programs?

```
$ python script.py
```

This article opens a series which seeks to answer this very question. We'll dive into the internals of CPython, Python's most popular implementation. By doing so we'll understand the language itself at a deeper level. That is the primary goal of this series. If you're familiar with Python and comfortable reading C but have no much experience working with CPython's source code, there is a good chance you'll find this writing interesting.

What CPython is and why anyone would want to study it

Let's begin by stating some well-known facts. CPython is a Python interpreter written in C. It's one of the Python implementations, alongside with PyPy, Jython, IronPython and many others. CPython is distinguished in that it is original, most-maintained and the most popular one.

CPython implements Python, but what is Python? One may simply answer – Python is a programming language. The answer becomes much more nuanced when the same question is put properly: what defines what Python is? Python, unlike languages like C, doesn't have a formal specification. The thing that comes closest to it is the Python Language Reference which starts with the following words:

> *While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here.*

So Python is not defined by its language reference only. It would be also wrong to say that Python is defined by its reference implementation, CPython, since there are some implementation details that are not a part of the language. The garbage collector that relies on a reference counting is one example. Since there is no single source of truth, we may say that Python is defined partly by the Python Language Reference and partly by its main implementation, CPython.

Such a reasoning may seem pedantic, but I think it is crucial to clarify the key role of the subject we're going to study. You might still wonder, though, why we should study it. Besides plain curiosity, I see the following reasons:

- Having a full picture gives a deeper understanding of the language. It's much more easier to grasp some peculiarity of Python if you're aware of its implementation details.

- Implementation details matter in practice. How objects are stored, how the garbage collector works and how multiple threads are coordinated are subjects of high importance when one wants to understand the applicability of the language and its limitations, estimate the performance or detect inefficiencies.
- CPython provides Python/C API which allows to extend Python with C and embed Python inside C. To use this API effectively a programmer needs a good understanding of how CPython works.

What it takes to understand how CPython works

CPython was designed to be easy to maintain. A newcomer can certainly expect to be able to read the source code and understand what it does. However, it may take some time. By writing this series I hope to help you to shorten it.

How this series is laid out

I chose to take a top-down approach. In this part we'll explore the core concepts of the CPython virtual machine (VM). Next, we'll see how CPython compiles a program into something that the VM can execute. After that, we'll get familiar with the source code and step through the execution of a program studying main parts of the interpreter on the way. Eventually, we'll be able to pick out different aspects of the language one by one and see how they're implemented. This is by no way a strict plan but my approximate idea.

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.

The big picture

An execution of a Python program roughly consists of three stages:

1. Initialization
2. Compilation
3. Interpretation

During the initialization stage, CPython initializes data structures required to run Python. It also prepares such things as built-in types, configures and loads built-in modules, sets up the import system and does many other things. This is a very important stage that is often overlooked by the CPython's explorers because of its service nature.

Next comes the compilation stage. CPython is an interpreter, not a compiler in a sense that it doesn't produce machine code. Interpreters, however, usually translate source code into some intermediate representation before executing it. So does CPython. This translation phase does the same things a typical compiler does: parses a source code and builds an AST (Abstract Syntax Tree), generates bytecode from an AST and even performs some bytecode optimizations.

Before looking at the next stage, we need to understand what bytecode is. Bytecode is a series of instructions. Each instruction consists of two bytes: one for an opcode and one for an argument. Consider an example:

```
def g(x):
    return x + 3
```

CPython translates the body of the function `g()` to the following sequence of bytes: `[124, 0, 100, 1, 23, 0, 83, 0]`. If we run the standard `dis` module to disassemble it, here's what we'll get:

```
$ python -m dis example1.py
...
2           0 LOAD_FAST            0 (x)
            2 LOAD_CONST           1 (3)
            4 BINARY_ADD
            6 RETURN_VALUE
```

The `LOAD_FAST` opcode corresponds to the byte `124` and has the argument `0`. The `LOAD_CONST` opcode corresponds to the byte `100` and has the argument `1`. The `BINARY_ADD` and `RETURN_VALUE` instructions are always encoded as `(23, 0)` and `(83, 0)` respectively since they don't need an argument.

At the heart of CPython is a virtual machine that executes bytecode. By looking at the previous example you might guess how it works. CPython's VM is stack-based. It means that it executes instructions using the stack to store and retrieve data. The `LOAD_FAST` instruction pushes a local variable onto the stack. `LOAD_CONST` pushes a constant. `BINARY_ADD` pops two objects from the stack, adds them up and pushes the result back. Finally, `RETURN_VALUE` pops whatever is on the stack and returns the result to its caller.

The bytecode execution happens in a giant evaluation loop that runs while there are instructions to execute. It stops to yield a value or if an error occurred.

Such a brief overview gives rise to a lot of questions:

- What do the arguments to the `LOAD_FAST` and `LOAD_CONST` opcodes mean? Are they indices? What do they index?
- Does the VM place values or references to the objects on the stack?
- How does CPython know that `x` is a local variable?
- What if an argument is too big to fit into a single byte?
- Is the instruction for adding two numbers the same as for concatenating two strings? If yes, then how does the VM differentiate between these operations?

In order to answer these and other intriguing questions we need to look at the core concepts of the CPython VM.

Code objects, function objects, frames

code object

We saw what the bytecode of a simple function looks like. But a typical Python program is more complicated. How does the VM execute a module that contains function definitions and makes function calls?

Consider the program:

```
def f(x):
    return x + 1

print(f(1))
```

What does its bytecode look like? To answer this question, let's analyze what the program does. It defines the function `f()`, calls `f()` with `1` as an argument and prints the result of the call. Whatever the function `f()` does, it's not a part of the module's bytecode. We can assure ourselves by running the disassembler.

```
$ python -m dis example2.py

1           0 LOAD_CONST               0 (<code object f at 0x10bffd1e0, file "example.py", line 1>)
            2 LOAD_CONST               1 ('f')
            4 MAKE_FUNCTION            0
            6 STORE_NAME               0 (f)

4           8 LOAD_NAME                1 (print)
           10 LOAD_NAME                0 (f)
           12 LOAD_CONST               2 (1)
           14 CALL_FUNCTION            1
           16 CALL_FUNCTION            1
```

```
            18 POP_TOP
            20 LOAD_CONST               3 (None)
            22 RETURN_VALUE
...
```

On line 1 we define the function `f()` by making the function from something called code object and binding the name `f` to it. We don't see the bytecode of the function `f()` that returns an incremented argument.

The pieces of code that are executed as a single unit like a module or a function body are called code blocks. CPython stores information about what a code block does in a structure called a code object. It contains the bytecode and such things as lists of names of variables used within the block. To run a module or to call a function means to start evaluating a corresponding code object.

function object

A function, however, is not merely a code object. It must include additional information such as the function name, docstring, default arguments and values of variables defined in the enclosing scope. This information, together with a code object, is stored inside a function object. The `MAKE_FUNCTION` instruction is used to create it. The definition of the function object structure in the CPython source code is preceded by the following comment:

> *Function objects and code objects should not be confused with each other:*
>
> *Function objects are created by the execution of the 'def' statement. They reference a code object in their __code__ attribute, which is a purely syntactic object, i.e. nothing more than a compiled version of some source code lines. There is one code object per source code "fragment", but each code object can be referenced by zero or many function objects depending only on how many times the 'def' statement in the source was executed so far.*

How can it be that several function objects reference a single code object? Here is an example:

```
def make_add_x(x):
    def add_x(y):
        return x + y
    return add_x

add_4 = make_add_x(4)
add_5 = make_add_x(5)
```

The bytecode of the `make_add_x()` function contains the `MAKE_FUNCTION` instruction. The functions `add_4()` and `add_5()` are the result of calling this instruction with the same code object as an argument. But there is one argument that differs – the value of `x`. Each function gets its own by the mechanism of cell variables that allows us to create closures like `add_4()` and `add_5()`.

Before we move to the next concept, take a look at the definitions of the code and function objects to get a better idea of what they are.

```
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) */

    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) */
        /* ... more members ... */
};


typedef struct {
    PyObject_HEAD
    PyObject *func_code;        /* A code object, the __code__ attribute */
    PyObject *func_globals;     /* A dictionary (other mappings won't do) */
    PyObject *func_defaults;    /* NULL or a tuple */
    PyObject *func_kwdefaults;  /* NULL or a dict */
    PyObject *func_closure;     /* NULL or a tuple of cell objects */
    PyObject *func_doc;         /* The __doc__ attribute, can be anything */
    PyObject *func_name;        /* The __name__ attribute, a string object */
    PyObject *func_dict;        /* The __dict__ attribute, a dict or NULL */
    PyObject *func_weakreflist; /* List of weak references */
    PyObject *func_module;      /* The __module__ attribute, can be anything */
    PyObject *func_annotations; /* Annotations, a dict or NULL */
    PyObject *func_qualname;    /* The qualified name */
    vectorcallfunc vectorcall;
} PyFunctionObject;
```

frame object

When the VM executes a code object, it has to keep track of the values of variables and the constantly changing value stack. It also needs to remember where it stopped executing the current code object to execute another and where to go on return. CPython stores this information inside a frame object, or simply a frame. A frame provides a state in which a code object can be executed. Since we're getting more accustomed with the source code, I leave the definition of the frame object here as well:

```
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 */

    PyObject **f_stacktop;          /* Next free slot in f_valuestack.  ... */
    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 first frame is created to execute the module's code object. CPython creates a new frame whenever it needs to execute another code object. Each frame has a reference to the previous frame. Thus, frames form a stack of frames, also known as the call stack, with the current frame sitting on top. When a function is called, a new frame is pushed onto the stack. On return from the currently executing frame, CPython continues the execution of the previous frame by remembering its last processed instruction. In some sense, the CPython VM does nothing but constructs and executes the frames. However, as we'll soon see, this summary, to put it mildly, hides some details.

Threads, interpreters, runtime

We've already looked at the three important concepts:

- a code object
- a function object; and
- a frame object.

CPython has three more:

- a thread state
- an interpreter state; and
- a runtime state.

thread state

A thread state is a data structure that contains thread-specific data including the call stack, the exception state and the debugging settings. It should not be confused with an OS thread. They're closely connected, though. Consider what happens when you use the standard `treading` module to run a function in a separate thread:

```
from threading import Thread

def f():
    """Perform an I/O-bound task"""
    pass

t = Thread(target=f)
t.start()
t.join()
```

`t.start()` actually creates a new OS thread by calling the OS function (`pthread_create()` on UNIX-like systems and `_beginthreadex()` on Windows). The newly created thread invokes the function from the `_thread` module that is responsible for calling the target. This function receives not only the target and the target's arguments but also a new thread state to be used within a new OS thread. An OS thread enters the evaluation loop with its own thread state, thus always having it at hand.

We may remember here the famous GIL (Global Interpreter Lock) that prevents multiple threads to be in the evaluation loop at the same time. The major reason for that is to protect the state of CPython from corruption without introducing more fine-grained locks. The Python/C API Reference explains the GIL clearly:

> *The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.*

To manage multiple threads, there needs to be a higher-level data structure than a thread state.

interpreter and runtime states

In fact, there are two of them: an interpreter state and the runtime state. The need for both may not seem immediately obvious. However, an execution of any program has at least one instance of each and there are good reasons for that.

An interpreter state is a group of threads along with the data specific to this group. Threads share such things as loaded modules (`sys.modules`), builtins (`builtins.__dict__`) and the import system (`importlib`).

The runtime state is a global variable. It stores data specific to a process. This includes the state of CPython (e.g. is it initialized or not?) and the GIL mechanism.

Usually, all threads of a process belong to the same interpreter. There are, however, rare cases when one may want to create a subinterpreter to isolate a group of threads. mod_wsgi, which uses distinct interpreters to run WSGI applications, is one example. The most obvious effect of isolation is that each group of threads gets its own version of all modules including `__main__`, which is a global namespace.

CPython doesn't provide an easy way to create new interpreters analogous to the `threading` module. This feature is supported only via Python/C API, but this may change someday.

Architecture summary

Let's make a quick summary of the CPython's architecture to see how everything fits together. The interpreter can be viewed as a layered structure. The following sums up what the layers are:

1. Runtime: the global state of a process; this includes the GIL and the memory allocation mechanism.
2. Interpreter: a group of threads and some data they share such as imported modules.
3. Thread: data specific to a single OS thread; this includes the call stack.
4. Frame: an element of the call stack; a frame contains a code object and provides a state to execute it.
5. Evaluation loop: a place where a frame object gets executed.

The layers are represented by the corresponding data structures, which we've already seen. In some cases they are not equivalent, though. For example, the mechanism of memory allocation is implemented using global variables. It's not a part of the runtime state but certainly a part of the runtime layer.

Conclusion

In this part we've outlined what `python` does to execute a Python program. We've seen that it works in three stages:

1. initializes CPython
2. compiles the source code to the module's code object; and
3. executes the bytecode of the code object.

The part of the interpreter that is responsible for the bytecode execution is called a virtual machine. The CPython VM has several particularly important concepts: code objects, frame objects, thread states, interpreter states and the runtime. These data structures form the core of the CPython's architecture.

We haven't covered a lot of things. We avoided digging into the source code. The initialization and compilation stages were completely out of our scope. Instead, we started with the broad overview of the VM. In this way, I think, we can

better see the responsibilities of each stage. Now we know what CPython compiles source code to – to the code object. Next time we'll see how it does that.

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

Update from 4 September 2020: I've made a list of resources that I've used to learn about CPython internals.