Python's Innards: Introduction

2010/04/02 § 22 Comments

What's this?

You are currently reading **Python's Innards: Introduction** at NIL: .to write(1) ~ help:about.

META

Author: Yaniv Aknin
Comments: 22 Comments
Categories: Python's Innards

A friend once said to me: You know, to some people, C is just a bunch of macros that expand to assembly. It's been years ago (smartasses: it was also before `llvm`, ok?), but the sentence stuck with me. Do *Kernighan and Ritchie* really look at a C program and see assembly code? Does *Tim Berners-Lee* surf the Web any differently than you and me? And what on earth *did Keanu Reeves* see when he looked at all of that funky green gibberish soup, anyway? No, seriously, what the heck *did* he see there?! Uhm, back to the program. Anyway, what does Python look like in *Guido van Rossum*'s[1] eyes?

This post marks the beginning of what should develop to a **series** on Python's internals, I'm writing it since I believe that explaining something is the best way to grok it, and I'd very much like to be able to visualize more of Python's 'funky green gibberish soup' as I read Python code. On the curriculum is mainly *CPython*, mainly *py3k*, mainly *bytecode evaluation* (I'm not a big compilation fan) – but practically everything around executing Python and Python-like code (*Unladen Swallow*, *Jython*, *Cython*, etc) might turn out to be fair game in this series. For the sake of brevity and my sanity, when I say *Python*, I mean *CPython* unless noted otherwise. I also assume a POSIX-like OS or (if and where it matters) Linux, unless otherwise noted. You should read this if you want to know how Python works. You should read this if you want to contribute to CPython. You should read this to find all the mistakes I'll make and snicker at me behind me back or write snide comments. I realize it's just *your* particular way to show affection.

I gather I'll glean pretty much everything I write about from Python's source or, occasionally, other fine materials (documentation, especially **this** and **that**, certain PyCon lectures, **searching python-dev**, etc). Everything is out there, but I do hope my efforts at putting it all in one place to which you can RSS-subscribe will make your journey easier. I assume the reader knows some C, some OS theory, a bit less than some assembly (any architecture), a bit more than some Python and has reasonable UNIX fitness (i.e., feels comfortable installing something from source). Don't be afraid if you're not reasonably conversant in one (or more) of these, but I can't promise smooth sailing, either. Also, if you don't have a working toolchain to do Python development, maybe you'd like to head over **here** and do as it says on the second paragraph (and onwards, as relevant).

Let's start with something which I assume you already know, but I think is important, at least to the main way I understand… well, everything that I do understand. I look at it as if I'm looking at a machine. It's easy in Python's case, since Python relies on a Virtual Machine to do what it does (like many other interpreted languages). Be certain you understand "**Virtual Machine**" correctly in this context: think more like JVM and less like VirtualBox (very technically, they're the same, but in the real world we usually differentiate these two kinds of VMs). I find it easiest to understand "Virtual Machine" literally – it's a machine built from software. Your CPU is just a complex electronic machine which receives all input (machine code, data), it has a state (registers), and based on the input and its state it will output stuff (to RAM or a Bus), right? Well, CPython is a machine built from software components that has a state and processes instructions (different implementations may use rather different instructions). This software machine operates in the process hosting the Python interpreter. Keep this in mind; I like the **machine** metaphor (as I explain in minute details **here**).

That said, let's start with a bird's eye overview of what happens when you do this: `$ python -c 'print("Hello, world!")'`. Python's binary is executed, the standard C library initialization which pretty much any process does happens and then the main function starts executing (see its source, `./Modules/python.c: main`, which soon calls `./Modules/main.c: Py_Main`). After some mundane initialization stuff (parse arguments, see if environment variables should affect behaviour, assess the situation of the standard streams and act accordingly, etc), **./Python/pythonrun.c: Py_Initialize** is called. In many ways, this function is what 'builds' and assembles together the pieces needed to run the CPython machine and makes 'a process' into 'a process with a Python interpreter in it'. Among other things, it creates two very important Python data-structures: the **interpreter state** and **thread state**. It also creates the built-in **module sys** and the module which hosts all **builtins**. At a later post(s) we will cover all these in depth.

With these in place, Python will do one of several things based on how it was executed. Roughly, it will either execute a string (the `-c` option), execute a module as an executable (the `-m` option), or execute a file (passed explicitly on the commandline or passed by the kernel when used as an interpreter for a script) or run its **REPL** loop (this is more a special case of the file to execute being an interactive device). In the case we're currently following, it will execute a single string, since we invoked it with `-c`. To execute this single string, `./Python/pythonrun.c: PyRun_SimpleStringFlags` is called. This function creates the `__main__` **namespace**, which is 'where' our string will be executed (if you run `$ python -c 'a=1; print(a)'`, where is `a` stored? in this namespace). After the namespace is created, the string is executed in it (or rather, interpreted or *evaluated* in it). To do that, you must first transform the string into something that machine can work on.

As I said, I'd rather not focus on the innards of Python's parser/compiler at this time. I'm not a compilation expert, I'm not entirely interested in it, and as far as I know, Python doesn't have significant Compiler-Fu beyond the basic CS compilation course. We'll do a (very) fast overview of what goes on here, and may return to it later only to inspect visible CPython behaviour (see the **global** statement, which is said to affect parsing, for instance). So, the parser/compiler stage of `PyRun_SimpleStringFlags` goes largely like this: tokenize and create a **Concrete Syntax Tree** (CST) from the code, transorm the CST into an **Abstract Syntax Tree** (AST) and finally compile the AST into a **code object** using `./Python/ast.c: PyAST_FromNode`. For now, think of the code object as a binary string of machine code that Python VM's 'machinary' can operate on – so now we're ready to do interpretation (again, *evaluation* in Python's parlance).

We have an (almost) empty `__main__`, we have a code object, we want to evaluate it. Now what? Now this line: `Python/pythonrun.c: run_mod, v = PyEval_EvalCode(co, globals, locals);` does the trick. It receives a code object and a namespace for **globals** and for **locals** (in this case, both of them will be the newly created `__main__` namespace), creates a **frame object** from these and executes it. You remember previously that I mentioned that `Py_Initialize` creates a thread state, and that we'll talk about it later? Well, back to that for a bit: each Python thread is represented by its own thread state, which (among other things) points to the stack of currently executing frames. After the frame object is created and placed at the top of the thread state stack, it (or rather, the byte code pointed by it) is evaluated, opcode by opcode, by means of the (rather lengthy) `./Python/ceval.c: PyEval_EvalFrameEx`.

`PyEval_EvalFrameEx` takes the frame, extracts opcode (and operands, if any, we'll get to that) after opcode, and executes a short piece of C code matching the opcode. Let's take a closer look at what these "opcodes" look like by disassembling a bit of compiled Python code:

```
>>> from dis import dis # ooh! a handy disassembly functior
>>> co = compile("spam = eggs - 1", "<string>", "exec")
>>> dis(co)
  1           0 LOAD_NAME                0 (eggs)
              3 LOAD_CONST               0 (1)
              6 BINARY_SUBTRACT
              7 STORE_NAME               1 (spam)
             10 LOAD_CONST               1 (None)
             13 RETURN_VALUE
>>>
```

…even without knowing much about Python's bytecode, this is reasonably readable. You "load" the name `eggs` (where do you load it from? where do you load it to? soon), and also load a constant value (`1`), then you do a "binary subtract" (what do you mean 'binary' in this context? between which operands?), and so on and so forth. As you might have guessed, the names are "loaded" from the `globals` and `locals` namespaces we've seen earlier, and they're loaded onto an operand stack (not to be confused with the stack of running frames), which is exactly where the binary subtract will pop them from, subtract one from the other, and put the result back on that stack. "Binary subtract" just means this is a subtraction opcode that has two operands (hence it is "binary", this is not to say the operands are binary numbers made of '0's and '1's).

You can go look at `PyEval_EvalFrameEx` at `./Python/ceval.c` yourself, it's not a small function by any means. For practical reasons I can't paste too much code from there in here, but I will just paste the code that runs when a `BINARY_SUBTRACT` opcode is found, I think it really illustrates things:

```
TARGET(BINARY_SUBTRACT)
    w = POP();
    v = TOP();
    x = PyNumber_Subtract(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
```

…pop something, take the top (of the operand stack), call a C function called PyNumber_Subtract() on these things, do something we still don't understand (but will in due time) called "Py_DECREF" on both, set the top of the stack to the result of the subtraction (overwriting the previous top) and then do something else we don't understand if x is not null, which is to do a "DISPATCH". So while we have some stuff we don't understand, I think it's very apparent how two numbers are subtracted in Python, at the lowest possible level. And it took us just about 1,500 words to reach here, too!

After the frame is executed and `PyRun_SimpleStringFlags` returns, the main function does some cleanup (notably, **Py_Finalize**, which we'll discuss), the standard C library deinitialization stuff is done (`atexit`, et al), and the process exits.

I hope this gives us a "good enough" overview that we can later use as scaffolding on which, in later posts, we'll hang the more specific discussions of various areas of Python. We have quite a few terms I promised to get back to: interpreter and thread state, namespaces, modules and builtins, code and frame objects as well as those DECREF and DISPATCH lines we didn't understand inside `BINARY_SUBTRACT`'s implementation. There's also a very crucial 'phantom' term that we've danced around all over this article but didn't call by name – **objects**. CPython's object system is central to understanding how it works, and I reckon we'll cover that – at length – in the **next post** of this series. Stay tuned.

---

[1] Do note that this series isn't endorsed nor affiliated with anyone unless explicitly stated. Also, note that though Python was initiated by **Guido van Rossum**, many people have **contributed** to it and to derivative projects over the years. I never checked with Guido nor with them, but I'm certain Python is no less clear to many of these contributors' eyes than it is to Guido's.

Related

Python's Innards: pystate
2010/05/26
In "Python's Innards"

Python's Innards: Code Objects
2010/07/03
In "Python's Innards"

Python's Innards: Objects 101
2010/05/12
In "Python's Innards"

Tagged: **internals**, **python**

§ 22 Responses to *Python's Innards: Introduction*

Eli Bendersky
2010/05/08 at 07:45

Excellent post – very well written, and the topic is fascinating. The promise of a series on Python's internals made me immediately add your blog to my RSS reader :-)

I've also written a little on Python's internals (**http://eli.thegreenplace.net/2009/11/28/python-internals-working-with-python-asts/**) and planned to write more – focusing on the compilation side, which I really like, so our material may become complementary.

Anyway, great job and keep this up.

Reply

Yaniv Aknin
2010/05/08 at 08:32

Thanks for your kind words! I'll be happy to read what you wrote/will write about the compilation phase.

Reply

karl
2010/05/09 at 00:37

For Tim's view on the Web, good news, it is accessible on the Web. :) Tim has kept for a long time a kind of diary behind the architecture and philosophy of the Web.

http://www.w3.org/DesignIssues/

Reply

Antonio
2010/05/09 at 07:41

Very cool stuff and yes, quite a few people should learn a bit from this kind of article.

I will also stay tunned for more :)

Reply

orip
2010/05/09 at 18:23

Nice :)

Reply

Yaniv Aknin
2010/05/09 at 18:29

Thank you, oh venerable Python-Fu master :)

Reply

Dan Powell
2010/05/10 at 01:10

This is the best overview of python internals I've seen.

Reply

Alex Conrad
2010/05/10 at 13:09

Great and fascinating post! Thanks!

Reply

Tobias
2010/05/10 at 18:49

Great post, but since the ret of the series will be just as long, please stop using low-contrast grey type.

Reply

Yaniv Aknin
2010/05/11 at 01:35

Thanks for the input!

I'm not really a CSS expert and I'm not sure my taste in design is always to my taste [sic], but I've hacked at the CSS a bit to make text, hyperlinks and headings twice as dark.

If you (or anyone) finds any visual bugs because of this, I'll be happy to know.

Reply

Tobias
2010/05/11 at 12:28

Much easier to read. Thanks!

Reply

Nick Coghlan
2010/05/26 at 03:57

Excellent start, and I really do recommend that manuscript I linked on python-dev.

You're including a lot more implementation-specific detail than I did (which is obviously part of the point), but I think many of the core concepts will be common (and important).

Reply

Snake
2010/06/29 at 14:48

would be so nice to have a print view of your postings!

Reply

Yaniv Aknin
2010/06/29 at 15:45

Hmm… You're the second person to complain, but I'm not sure what to fix. The posts print nicely for me (Chrome, Ubuntu, to PDF). I'm not sure I can help, since the blog is hosted, but can you please describe exactly what are you expecting from a print which you currently don't get?

Reply

Mike
2010/12/08 at 20:23

PyString_FromString("very cool :-)");

Reply

transfinite
2011/01/25 at 00:10

Great, thanks for this, it's just the material I wanted at the depth & pace I've been looking for.

I've taken the liberty of converting it to Kindle format at **http://www.zinepal.com/user/benm** , hope that you don't object.

Reply

Tweets that mention Python's Innards: Introduction « NIL: .to write(1) ~ help:about -- Topsy.com
2011/01/25 at 01:59

[…] This post was mentioned on Twitter by Andrew Gleave, Frank Worsley and Ben Moran, Christophe Lalanne. Christophe Lalanne said: RT @benm: Here is a nice series of blogs by Yaniv Aknin on the innards of Python **http://bit.ly/hmEJvy** , as eBooks for Kindle: **http://b** … […]

Reply

Good to Great Python reads | Pythoners' Home
2011/11/16 at 06:50

[…] Python's Innards: Introduction […]

Reply

Interesante listado de enlaces sobre Python | CyberHades
2011/11/16 at 14:27

[…] Python's Innards: Introduction […]

Reply

Python (programming language): What is the best resource for understanding what goes on "under the hood" in the interpretation of Python programs? - Quora
2012/10/26 at 21:54

[…] work at the interpreter level, please read through the whole "python innards" serieshttps://tech.blog.aknin.name/2010…Embed QuoteComment Loading… • Share • Embed • Just now Ankur Gupta, […]

Reply

Python (programming language): What is the best resource for understanding what goes on "under the hood" in the interpretation of Python programs? - Quora
2012/10/27 at 07:02

[…] work at the interpreter level, please read through the whole "python innards" serieshttps://tech.blog.aknin.name/2010…Embed QuoteComment Loading… • Share • Embed • 9h ago Ankur Gupta, […]

Reply

Python's Innards: for my wife « NIL: .to write(1) ~ help:about
2013/02/02 at 18:43

[…] something to put her to sleep. Since she's not quite a hacker, I figured some discussion of what I usually write about may do the trick (okay, maybe 'not quite a hacker' is an understatement, she's an […]

Reply

Leave a Reply

Enter your comment here...

« Contributing to Python   Labour updated »