6.009: Fundamentals of Programming

Week 12 Lecture: Digging Into the Python Interpreter

- Understanding Where We Are in the Universe
- Explaining Some Behaviors of Python by Investigating the Source Code
- Playing with Python Bytecode for Fun ~~and Profit~~

Adam Hartz
hz@mit.edu

10 May 2021

The Power of Abstraction

Any system can be analyzed at many different layers of abstration.

The primitives at any layer of abstraction are in fact complicated combinations of primitives at a different layer of abstration.

The Power of Abstraction

Any system can be analyzed at many different layers of abstration.

The primitives at any layer of abstraction are in fact complicated combinations of primitives at a different layer of abstration.

Example (operations in Python):

- Primitives: `+`, `*`, `==`, `!=`, ...
- Combination: `if`, `while`, `f(g(x))`, ...
- Abstraction: `def`

The Python Interpreter

Today, we'll explore the Python interpreter just a little bit, using the source code of the interpreter, as well as some built-in modules, to understand a few things we've alluded to in 6.009.

The Python Interpreter

Today, we'll explore the Python interpreter just a little bit, using the source code of the interpreter, as well as some built-in modules, to understand a few things we've alluded to in 6.009.

Why do we care?

The Python Interpreter

Today, we'll explore the Python interpreter just a little bit, using the source code of the interpreter, as well as some built-in modules, to understand a few things we've alluded to in 6.009.

Why do we care?

- Maybe we find it cool in and of itself :)
- Maybe we're planning on contributing code to the Python interpreter
- Maybe we're interested in writing more complex interpreters of our own
- Maybe we can use this to explain some of Python's behaviors

The Python Interpreter

Today, we'll explore the Python interpreter just a little bit, using the source code of the interpreter to understand a few things we've alluded to in 6.009.

For example, we mentioned early on that running containment checks on a list takes time that grows like the length of the list. But from our perspective, we just saw the `in` operator. How can we understand what is happening there?

Let's look and see :)

Digging Into Python a Bit More

The CPython interpreter generally works by converting code first to an abstract syntax tree (like we did in labs 9/10), but then CPython takes an additional step, converting that AST to an intermediate representation called *bytecode*.

$$\text{source string} \rightarrow \text{abstract syntax tree} \rightarrow \text{"bytecode"} \rightarrow \text{result}$$

In some sense, the thing that is actually interpreted is the bytecode. That representation is the focus of the rest of today.

The Abstraction: A "Stack Machine"

Bytecode can be thought of as a sequence of low-level instructions for a "virtual machine." This machine consists of several abstractions:

- the **heap**, where **objects** are stored
- a **data stack**, which contains references to objects as the interpreter operates on them
- a **call stack**, where **stack frames** are held

The Abstraction: A "Stack Machine"

Bytecode can be thought of as a sequence of low-level instructions for a "virtual machine." The interpreter simulates this machine, and the various instructions affect the state of the machine (the data stack, the call stack, and the heap).

Several kinds of instructions ("opcodes") exist, for example:

- Data stack manipulation (pop an item off the stack, swap the top two elements, duplicate the top element, ...)
- Flow control (jump to different spots in the code)
- Arithmetic (add two numbers, comparisons, etc)
- Python-specific (make a dictionary, make a list, call a function, etc)

All Python code is first converted to bytecode before being interpreted, and interpretation includes simulating this machine.

First-Class Objects

Interestingly, Python's representation of frames, and of bytecode, are first-class objects that are accessible at runtime.

Important data types we'll be working with:

- `code` objects represent Python bytecode (including the raw "machine code," necessary constants, etc)
- `function` objects combine code and information about the context in which the function was defined
- `frame` objects keep track of current execution (variable bindings, code, interpreter's position in the code)

Our First Python `code` Object

Interestingly, Python's representation of frames, and of bytecode, are first-class objects that are accessible at runtime.

Let's take a look at Python's internal representation of the following function:

```
def double(x):
    return x*2
```

The `dis` Module

Interestingly, Python's representation of frames, and of bytecode, are first-class objects that are accessible at runtime.

The Python standard library includes a module for analyzing bytecode by disassembling it:`dis`

I'll use several pieces of the `dis` module throughout this lecture:

- `dis.disco(code_object)`: print a (relatively) human-readable disassembly of the given code object
- `dis.get_instructions(code_object)`: returns an iterator over the code object, in a useful representation
- `dis.opmap` and `dis.opname` are dictionaries mapping opcodes to human-readable names, and *vice versa*

Let's take another look at `double`

Why Do This?

In large part, this structure helps manage the complexity of the interpreter.

The CPython interpreter is a big piece of software, but the core of the *evaluation/interpretation* step, in some sense, is a single large switch statement inside of a loop in `ceval.c`.

Why Do We Care?

One reason to care about this representation is that it lets us see some details of Python's operation that might otherwise be difficult to see.

For example:

- `elif` vs. nested conditionals
- short-circuiting of `and` and `or`
- chained comparison operations (`x < y < z`, vs `x < y and y < z`)

Check Yourself!

One more example before moving on. Below is the result of calling `dis.disco` on some function:

```
  2           0 LOAD_FAST                0 (x1)
              2 LOAD_FAST                0 (x1)
              4 BINARY_MULTIPLY
              6 LOAD_FAST                1 (x2)
              8 LOAD_FAST                1 (x2)
             10 BINARY_MULTIPLY
             12 BINARY_ADD
             14 LOAD_CONST               1 (0.5)
             16 BINARY_POWER
             18 RETURN_VALUE
```

What does this function do?

Let's Write Some Python!

It is also possible, of course, to create code objects of our own!

Let's write an `absolute_value` function.

The Rest of Today

Examples of analyzing/manipulating bytecode to enable interesting new behaviors that would be difficult/impossible with regular Python syntax.

Feel free to try this at home, but maybe not in production software.

- Python bytecode is poorly documented (if at all)
- The bytecode details differ between Python implementations and Python versions (I'm using CPython, v3.9.2)
- These kinds of hacks can be difficult to debug (example: in `absolute_value`, change one opcode)

Fun Hack #1: Detecting Whether Return Value is Used

First example hack: determining if/how a function's return value is going to be used, from within that function.

Fun Hack #2: Goto

Some languages have support for "jumps" to arbitrary locations in the source code (can be used as a replacement for looping structures, etc).

Let's see if we can make Python support this :)

Fun Hack #3: Tail Call Elimination

Tail Call Elimination is an optimization (intentionally left out of Python) that allows tail-recursive function calls to reuse existing stack frames (rather than allocating new ones).

A tail call is a call to a function that happens as the last action in a function. If this call is a recursive call, the function is said to be tail-recursive.

```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)

def fib(n, current=0, next_=1):
    if n == 0:
        return current
    return fib(n-1, next_, current+next_)
```

The Real Message

The power of **abstraction**!