Exploring Python dis

Justus Perlwitz

2015-10-17

Contents

As you might know, CPython, the most commonly used implementation of Python, uses a stack based virtual machine to run Python scripts. This means that Python script ASTs are never directly executed. Instead, CPython compiles `.py` script files into byte code files first (`.pyc` files). This allows for some optimizations during the byte code compilation step and shipping of byte code files to end customers instead of raw scripts. This is useful to make reverse engineering hard. Ableton Live includes a CPython interpreter for its MIDI hardware integration. It does not let users look at the actual hardware integration scripts though. Instead, all hardware integration scripts can only be found in their compiled `.pyc` form.

So what exactly can be found in CPython byte code? Byte code contains the stack instructions for CPython's virtual machine. Enter the Python disassembly library dis. It is included in the CPython standard library and allows developers to view and understand the internal bytecode representation of code objects in CPython.

Other implementations of Python typically do not use an internal byte code representation. Therefore, the following is highly CPython specific.

First, let's get familiar with code objects. Code objects can be created from any executable series of Python statements. For example, if we enter the following in a REPL

```
def show_weather():
    print("The weather is great!")
```

we not only define a function `show_weather`, but a CPython code object as well. But first, we need to check the output of our newly defined function.

```
show_weather()
```

```
The weather is great!
```

Now that the we know that the weather is great, we should take a look at the code object. A CPython code object can be accessed through the `__code__` attribute of a function.

Before even using the `dis` library, we can actually peek at the byte code of our `show_weather` function.

```
code_object = show_weather.__code__ # __code__ contains the function internals
code = code_object.co_code
print(code)
```

```
b't\x00\x00d\x01\x00\x83\x01\x00\x01d\x00\x00S'
```

That does not look very helpful. Thankfully, the `dis` module has an `opname` list, that maps byte code instructions (opcodes) to human readable names, similar to assembler instruction mnemonics such as MOV, ADD, etc.

We need to keep in mind though, that we cannot just run all bytes through the `opname` list, as some instructions can require attributes. Attributes usually are numbers that refer to local and global names. Therefore, not all bytes in a byte code object represent byte code instructions.

Let's try it out. We will see if there is a problem.

```
from dis import opname
for byte in code:
    print('Byte {} -> opcode {}'.format(byte, opname[byte]))
```

```
Byte 116 -> opcode LOAD_GLOBAL
Byte 0 -> opcode <0>
Byte 0 -> opcode <0>
Byte 100 -> opcode LOAD_CONST
Byte 1 -> opcode POP_TOP
Byte 0 -> opcode <0>
Byte 131 -> opcode CALL_FUNCTION
Byte 1 -> opcode POP_TOP
Byte 0 -> opcode <0>
Byte 1 -> opcode POP_TOP
Byte 100 -> opcode LOAD_CONST
Byte 0 -> opcode <0>
Byte 0 -> opcode <0>
Byte 83 -> opcode RETURN_VALUE
```

That looks interesting. We immediately notice the occurrence of `<0>` opnames. Since a lot of the CPython opcodes require arguments to be specified, not all bytes in our bytecode represent opcodes, but references to values. In this case `<0>` is part of a numeric reference to a constant or global value. `POP_TOP` is part of the reference number as well, but we naively interpret it as an opcode.

If we want to know which values our function is referring to, we have to dig into the code object even more. In this particular example we are interested in constants and name references. You will see why.

```
def pformat(tpl):
    return " \n".join("{}: {}".format(n, val) for n, val in enumerate(tpl))

print("Constant values:")
print(pformat(code_object.co_consts))

print("Names:")
print(pformat(code_object.co_names))
```

```
Constant values:
0: None
1: The weather is great!
Names:
0: print
```

Excellent. We were able to retrieve the reference to **`print`** and the string constant **`The weather is great!`**. Let's see whether we're able to match the bytecode references with our references.

Luckily, the **`dis`** module has a few lists of byte codes called **`hasname`**, **`hasconst`**, **`hasnargs`** and similar, that contain all the opcodes that require name, constant or other arguments. Since our code only has constants, global values, and function calls, we can go on without caring about other types of arguments. Function calls require a 2 byte argument that specifies the amount of keyword parameters (significant byte) and the number of positional parameters (least significant byte) that have been placed on the stack.

```
from dis import hasname, hasconst, hasnargs

code_listing = list(code) # copy the code as we are going to pop bytes off
while code_listing:
    byte = code_listing.pop(0)
    addendum = ''
    if byte in hasname:
        index = code_listing.pop(0) << 1 + code_listing.pop(0)
        addendum = '(name {})'.format(repr(code_object.co_names[index]))
    elif byte in hasconst:
        index = code_listing.pop(0)
        addendum = '(const {})'.format(repr(code_object.co_consts[index]))
        code_listing.pop(0)
```

```
    elif byte in hasnargs:
        nargs = code_listing.pop(0)
        nkwargs = code_listing.pop(0)
        addendum = '({} kw params, {} positional params)'.format(
            nkwargs, nargs)
    print('{} \t{}'.format(opname[byte], addendum))
```

```
LOAD_GLOBAL  (name 'print')
LOAD_CONST  (const 'The weather is great!')
CALL_FUNCTION  (0 kw params, 1 positional params)
POP_TOP
LOAD_CONST  (const None)
RETURN_VALUE
```

Holy cow! A working CPython disassembler! I doubt that the CPython developers are be able to come up with something like that!

```
from dis import dis
dis(show_weather)
```

```
2           0 LOAD_GLOBAL              0 (print)
            3 LOAD_CONST               1 ('The weather is great!')
            6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
            9 POP_TOP
           10 LOAD_CONST               0 (None)
           13 RETURN_VALUE
```

Never mind. There already is a disassembly method.
Back to work.