Introduction to Computational Complexity Theory

Hitesh Kumar Rawat*

Department of Aerospace Engineering, Indian Institute of Technology Kanpur

E-mail: hiteshkr@iitk.ac.in

Abstract

This paper covers as per 'End-term Evaluation' rules the summary of the sources that I have read, analyzed and worked upon in past few months. The paper very briefly covers the major aspects of Theoretical Computer Science in the name of following topics:

1. Efficient Universal Turing Machine and its application
2. py-py Interpreter - A practical example of Universal Turing Machine
3. Decidability and Computatbility
4. The Gödel's Approach towards
5. The Class P
6. The Class NP
7. The idea of NP-completeness

Introduction : The Idea of general purpose Computers

Turing was the first to observe that general-purpose computers are possible, by showing a universal Turing machine that can simulate the execution of every other TM M given Ms description as input. Of course, since we are so used to having a universal computer on our desktops or even in our pockets, today we take this notion for granted. But it is good to remember why it was once counterintuitive. The following section states an efficient version of Universal Turing Machine.

Universal Turing Machine

There exists a TM U such that for every x, $\alpha \in \{0,1\}$, $U(x,\alpha) = M_\alpha(x)$, where M_α denotes the TM represented by α. Moreover, if M_α halts on input x within T steps then $U(x,\alpha)$ halts within $CTlogT$ steps, where C is a number independent of $|x|$ and depending only on M_α s alphabet size, number of tapes, and number of states.

Given U, for a turing machine M, U can mimic M into a Turing Machine $\widetilde{M}$ that satisfies the same properties of M. This mimic will at most lead to a quadratic slowdown.

For example a common exercise is to create an interpreter for a particular programming language using the same programming language. And being inspired and motivated by this exercise, I decided to take on this as a project. So I had build a Python Interpreter written in Python. It's called 'py-py Interpreter'. It is, in its current state only compatible with Python 2.x and 3.5 and lower. This follows the next section.

py-py Interpreter

The interpreter, named Byterun is a Python interpreter implemented in Python. Through my work on Byterun, I was surprised and delighted to discover that the fundamental structure of the Python interpreter fits easily into the 500-line size restriction. In this section, we would

walk-through the structure of the interpreter, giving enough context to explore it further. Its structure is similar to the primary implementation of Python, CPython, so understanding Byterun will help you understand interpreters in general and the CPython interpreter in particular.

Preface

The word "interpreter" can be used in a variety of different ways when discussing Python. In this context, "interpreter" has a more narrow meaning: it's the last step in the process of executing a Python program. Most interpreted languages, including Python, do involve a compilation step. The reason Python is called "interpreted" is that the compilation step does relatively less work (and the interpreter does relatively more) than in a compiled language.

Understanding Python's Stack Interpreter

The Python interpreter is a virtual machine, meaning that it is software that emulates a physical computer. This particular virtual machine is a stack machine: it manipulates several stacks to perform its operation.

Building an Interpreter

To make this concrete, start with a very minimal interpreter. This interpreter can only add numbers, and it understands just three instructions. The three instructions are these:

- LOAD_VALUE
- ADD_TWO_VALUES
- PRINT_ANSWER

Since we're not concerned with the lexer, parser, and compiler in this chapter, it doesn't matter how the instruction sets are produced. Suppose that

```
7+5
```

produces the following instructions:

```
what_to_execute = {
    "instructions": [("LOAD_VALUE", 0),  # the first number
                     ("LOAD_VALUE", 1),  # the second number
                     ("ADD_TWO_VALUES", None),
                     ("PRINT_ANSWER", None)],
    "numbers": [7, 5] }
```

Since Python interpreter is a stack machine. The instructions will be stacked as shown in Figure 1. These three functions implement the three instructions our interpreter understands.

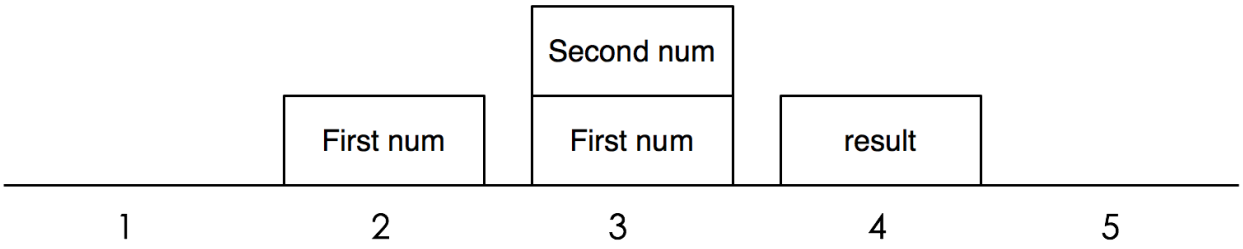

Figure 1: A stack machine

The interpreter needs one more piece: a way to tie everything together and actually execute it.

Variables require an instruction for storing the value of a variable, STORE_NAME; an instruction for retrieving it, LOAD_NAME; and a mapping from variable names to values. To keep track of what names are bound to what values, we'll add an environment dictionary to the __init__ method. We'll also add STORE_NAME and LOAD_NAME. These methods first look up the variable name in question and then use the dictionary to store or retrieve its value.

Understanding Real Python Bytecode

The structure of bytecode is similar to our toy interpreter's verbose instruction sets. To understand this structure, we'll walk through the bytecode of a short function. Consider the example below:

```
>>> def cond():
...     x = 3
...     if x < 5:
...         return 'yes'
...     else:
...         return 'no'
```

Here comes the use of Python's *dis* module comes into picture. *dis* is a bytecode disassembler. A disassembler takes low-level code that is written for machines, like assembly code or bytecode, and prints it in a human-readable way. See Documentation. [1]

```
>>> dis.dis(cond)
  2           0 LOAD_CONST               1 (3)
              3 STORE_FAST               0 (x)

  3           6 LOAD_FAST                0 (x)
              9 LOAD_CONST               2 (5)
             12 COMPARE_OP               0 (<)
             15 POP_JUMP_IF_FALSE       22

  4          18 LOAD_CONST               3 ('yes')
             21 RETURN_VALUE

  6     >>   22 LOAD_CONST               4 ('no')
```

```
 25 RETURN_VALUE
 26 LOAD_CONST               0 (None)
 29 RETURN_VALUE
```

Frames

In the examples above, the last instruction is RETURN_VALUE, which corresponds to the return statement in the code. But where does the instruction return to?

To answer this question, we must add a layer of complexity: the frame. A frame is a collection of information and context for a chunk of code. Frames are created and destroyed on the fly as your Python code executes. There's one frame corresponding to each call of a functionso while each frame has one code object associated with it, a code object can have many frames. If you had a function that called itself recursively ten times, you'd have eleven framesone for each level of recursion and one for the module you started from. In general, there's a frame for each scope in a Python program.

Frames live on the call stack, a completely different stack from the one we've been discussing so far.The stack we've been examiningthe one the interpreter is manipulating while it executes bytecodewe'll call the data stack. There's also a third stack, called the block stack. Blocks are used for certain kinds of control flow, particularly looping and exception handling. Each frame on the call stack has its own data stack and block stack.

See Documentation. [3]

Byterun : The py-py Interpreter

There are four kinds of objects in Byterun:

- A VirtualMachine class, which manages the highest-level structure, particularly the call stack of frames, and contains a mapping of instructions to operations. This is a more complex version of the Intepreter object above. [2]

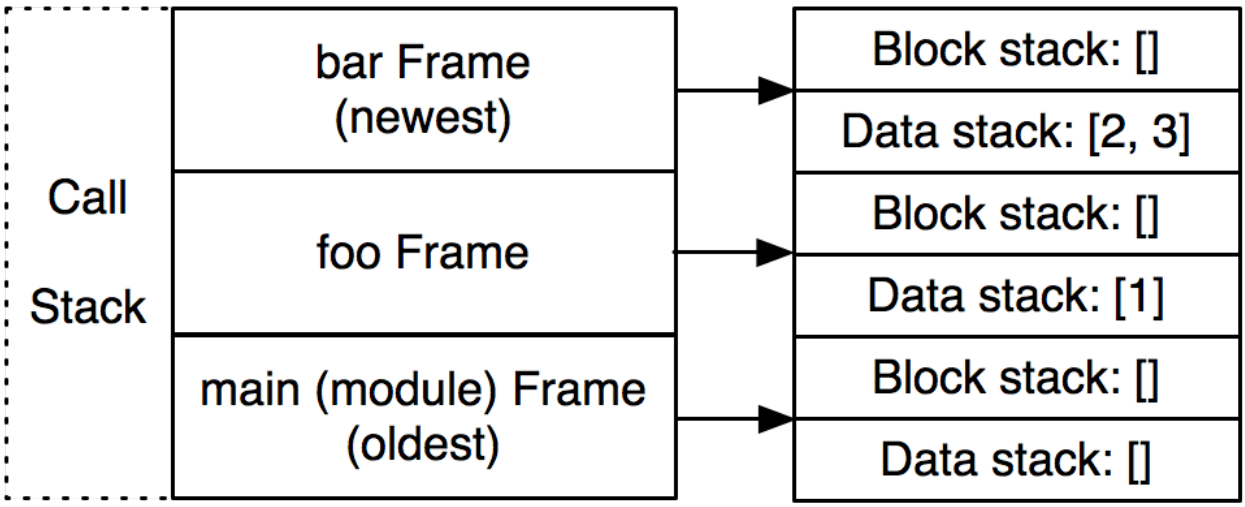

Figure 2: A call stack

- A Frame class. Every Frame instance has one code object and manages a few other necessary bits of state, particularly the global and local namespaces, a reference to the calling frame, and the last bytecode instruction executed. [3]
- A Function class, which will be used in place of real Python functions. Recall that calling a function creates a new frame in the interpreter. We implement Function so that we control the creation of new Frames. [4]
- A Block class, which just wraps the three attributes of blocks. [5]

The Function Class [4] [7]

The Block Class [5] [7]

The Instructions [6] [7]

All that's left is to implement the dozens of methods for instructions. Following are abstract form some actual instructions, but the full implementation is avavailable on GitHub.

- Building Variables, Lists, Maps [6]
- Indexing [6]

- Stack Manipulation: push, pop [6]
- Names: Loading names, storing names, loading locals and globals [6]
- Operators as python functions [6]
- Jumps in Loops [6]
- Blocks for Loops [6]
- Function Definitions [6]
- Function Returns [6]

Conclusion

py-py Interpreter is a compact Python interpreter that's easier to understand than CPython. py-py Interpreter replicates CPython's primary structural details: a stack-based interpreter operating on instruction sets called *bytecode*. It steps or jumps through these instructions, pushing to and popping from a stack of data. The interpreter creates, destroys, and jumps between frames as it calls into and returns from functions and generators. This implementation shares the real interpreter's limitations, too: because Python uses dynamic typing, the interpreter must work hard at run time to determine the correct behavior of a program.

Other Topics

The Gödel's Approach towards Uncomputablity

In layman's language, it states: For every sound system S of axioms and rules of inference, there exists true number of theoretic statements that cannot be proven in S.

> "Any sufficient powerful axiomisation of mathematics cannot prove its own consistency."
>
> Citation of Pg.20 . [8]

The Class P

The Class NP

The idea of NP-completeness

Acknowledgement

Thanks to Ned Batchelder, Allison Kaptur, Michael Arntzenius, Leta Montopoli and Recurse Center community for originating this project debugging the code and editing the prose, edits. A big thank to my mentor, Aravind Reddy. Any errors are my own.

References

(1) Chapter 4, documentation.pdf, py-py Interpreter

(2) Section 6.1, Chapter 6, documentation.pdf, py-py Interpreter

(3) Section 6.2, Chapter 6, documentation.pdf, py-py Interpreter

(4) Section 6.3, Chapter 6, documentation.pdf, py-py Interpreter

(5) Section 6.4, Chapter 6, documentation.pdf, py-py Interpreter

(6) Chapter 7, documentation.pdf, py-py Interpreter

(7) Explanation out of scope. See documentation.

(8) Section 1.5.2, Computational Complexity: A Modern Approach, ISBN-10 0521424267