An Exercise in Assigning Meaning to a Python Program

M. Devi Prasad

dp@vlabs.ac.in

deviprasad.m@iiit.ac.in,

1 Background

In this short article, we show the application of simple and interesting ideas from Computing Science to understand the behavior of a Python program. The motivation is to apply a small number of "pure" ideas of Computing Science to the "impure" settings of real world (Python) programs with the hope that this exercise yields useful insights.

In what follows, we employ basic ideas of first-order logic to track the execution state of our Python program. In so doing, we will discover a bug in our program. Further, we will clarify one aspect of Python's iterator mechanism that appears to provide a stream-like interface to external files. Our attempt to express, in logical terms, the meaning of file iterators reveals that all iterators of an underlying file share the same state. We have found that this invariably surprises programmers whose intuitions are shaped by iterators for in-built sequence types of Python. While passing we would like to mention that the official documentation of Python language (and its libraries) do not state this fact.

The primary audience of this article includes students of ITWS2 course to whom we teach two "scripting" languages - Python and JavaScript - to build 2D games. Our goal is to impress the fact that the techniques used in this exercise can be easily practiced by novices, and with a little practice, can be used as an effective tool to guide the creation of test cases. It is also useful to recognize that these very techniques may be used to formulate program assertions.

2 Context

We intend to assign meaning to a Python program that parses text files generated by running the `set` command under Linux ("bash") shell. Therefore, it is essential to understand the structure of the input to our program. But first, we shall see the actual command and the input text creation step:

```
$ set > env.txt
```

Note that the output of the `set` command is redirected to a text file named `env.txt`. When the command executes successfully, this file contains a series of key-value pairs representing

the environment variables active in the context of the currently running shell. Each line in the text file contains one key-value pair.

In addition to key-value pairs, the text file contains a series of function definitions. A function definition consists of a name followed by an optional pair of braces ("(" and ")"), followed by a block of statements. A statement block is enclosed within a pair of matching curly brackets ("{" and "}"). It contains a sequence of statements (or shell commands). The open and close curly brackets appear on separate lines with no other lexical token either preceding them or following them (other than the "invisible" newline character!). Here is a sample definition to help us visualize the typical form of a function:

```
BASH=/bin/bash
BASH_ARGC=()
CHROME_DEVEL_SANDBOX=/usr/local/sbin/chrome-devel-sandbox
COLORTERM=gnome-terminal
COLUMNS=76
__git_log_pretty_formats="oneline short medium full email raw format:"

dequote ()
{
    eval echo "$1" 2> /dev/null
}
_ssh ()
{
    local cur prev configfile;
    local -a config;
    COMPREPLY=();
    ...
}
__git_complete_file ()
{
    __git_complete_revlist_file
}
```

Our goal is to recognize only function names (along with the optional braces) and ignore key-value pairs as well as function bodies. The program is expected to output names of the functions one per line. If we were to execute our program with the input shown above, we would expect the following listing in the output:

```
dequote ()
_ssh ()
__git_complete_file ()
```

3 The Program

In this section we will take a closer look at a Python program that appears to meet our requirements mentioned in the previous section. Our intention is to rely on informal reasoning to quickly check that the program indeed works in most cases and that there are no glaring logical errors.

Let us begin with the trivial definition of two helper functions. The `block_begin` function checks if *line* (argument) contains a single open curly brace. Similarly, `block_end` function checks for a closing curly brace.

```
1 def block_begin(line): return line == '{\n'
2 def block_end(line):   return line == '}\n'
```

Notice

```
1  def collect_fun_names(src):
2    fun_names = []
3    prev_line = None
4
5    for line in src:
6      if block_begin(line):
7        fun_name = prev_line
8        for line in src:
9          if block_end(line):
10           fun_names.append(fun_name)
11           break
12     else:
13       prev_line = line
14
15   return fun_names
```

4 Assigning Meaning to the Program

```
1  def collect_fun_names(src):
2    fun_names = []
3    prev_line = None                          prev_line = ⊥
4    for line in src:                          (|src| = N) ∧ (0 ≤ i < N)
5      if open_curly(line):                    line = src[i] ∧ i = 0 ⇒ prev_line = ⊥
6        fun_name = prev_line                  line = "{" ∧ i = 0 ⇒ fun_name = ⊥
7        for line in src:                      (i + 1) ≤ j < N ∧ line = src[j]
8          if close_curly(line):
9            fun_names.append(fun_name)        src[i] = "{" ∧ src[j] = "}" ∧ (i ≤ j < N)
10           break                             (i = 0) ⇒ ⊥ ∈ fun_names
11     else:
12       prev_line = line
13   return fun_names
```