An Exercise in Assigning Meaning to a Python Program

M. Devi Prasad

dp@vlabs.ac.in

deviprasad.m@iiit.ac.in,

1 Introduction

In this article we show the application of simple yet effective ideas of 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 and programmers to whom we wish to impress the fact that the techniques used in this exercise may 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 The Problem Statement

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 procedure definitions. A procedure 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 procedure names (along with the optional braces) and ignore key-value pairs as well as function bodies. The program is expected to print the names of the functions one per line on the *standard output* stream. If we were to execute our program with the input shown above, we would expect the following output:

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

3 The Program

In this section we will take a close look at a Python program that appears to meet the stated requirements. We will rely on informal reasoning to satisfy ourselves 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) constitutes an open curly brace. Similarly, `block_end` function checks for a closing curly brace. Taken together these functions will be used to recognize the beginning and end of a statement block.

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

Let us now turn our attention to `collect_fun_names` that teases out procedure names by recognizing procedure definitions in the input text. The `src` argument of `collect_fun_names` is a Python file object representing the external text file. This function returns a list of procedure names found in the input.

The working of `collect_fun_names` is straightforward. It sets off looking for the beginning of a statement block - an open-curly bracket on a line by itself. When it detects one, it looks for the matching close-curly bracket appearing on an independent line. Since a procedure name precedes the statement block, `collect_fun_names` stores the contents of the previous line on detecting the the beginning of a potential statement-block. The identifier `prev_line` in the listing below serves this purpose.

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

In the program listing above, line 1 creates an empty list that would contain, when the program terminates, names of procedures found in the input. The *for* loop on line 5 starts reading each line from the input stream `src` looking for statement blocks. Line 7 stores a potential procedure name in `pn`. The code on lines 8, 9 and 10 look for the end of a statement block, and if found, insert the name of the procedure into the list of procedure names (`proc_names`).

3.1 The semantics of the *for* statement

At this point we should draw the attention of the reader to an important aspect of Python programming language. The nested iteration in the listing above, represented by two *for* statements on lines 5 and 8, use the same identifier - `line` - for binding successive lines from the input. This is perfectly valid in Python because the identifier `line` gets bound *each time* through the iteration. This is unlike most other imperative programming languages where either it is wrong to mutate the value of the loop index or it will adversely affect iteration. In the following discussion, we will pay some more attention this particular aspect because it is an important detail for understanding our program. At an altogether different level, this discussion will highlight a serious flaw with the definition of iterator interface in Python.

Iteration over built-in sequence types and user-defined containers is defined in terms of two key ideas: an *iterable* and an *iterator*. Iterables are types that implement `__iter__()` method to yield an iterator object. Iterators implement an interface with two methods: `__next__()` and `__iter__()`. The former returns the next item from the container. It raises the StopIteration exception when no further elements are available. The `__iter__()` method of an iterator object is required to return a reference to itself. This makes it possible to use containers and iterators with the *for* and *in* statements.

In Python the *for* statement is used to iterate over the elements of a sequence such as a string, tuple, list or any *iterable* object. In order to understand the semantics of *for* statement we shall take a look at its syntax using the BNF notation. For brevity and to focus only on the essentials, we show only a fragment of the actual rule from the official Python language documentation while eliding the details:

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
expression_list ::=  expression ( "," expression )* [","]
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                 | ...
```

The symbols enclosed within a pair of double-quotes represent either keywords or exact lexical symbols. Symbols enclosed within a pair of square brackets are optional elements, and $*$ represents for zero or more repetitions of a grammar symbol.

The execution of the *for* statement begins with the evaluation of `expression_list`. The `expression_list` is evaluated only once and this it is expected to yield an *iterable* object, which is used in turn to obtain an *iterator* instance. This iterator is further used to obtain successive values from the underlying container, and each value is bound to the identifier in the `targe_list`. The iterator is not directly visible in most cases, with the program accessing only the value returned by the iterator. In other words, the *for* statement effectively hides the iterator interface behind syntactic sugar!

Referring back to our program, it is important to note that the two `for` loops found on lines 5 and 8 share a common iterator representing shared offsets within the underlying

physical file stream. This is in contrast to the iterator interface exposed by most sequence types in Python. We shall return to this topic later in this article.

4 Assigning Meaning to the Program

What do we mean by assigning meaning to a program?

We can imagine the meaning of a program to be various invariants and logical relations that hold among state variables of a program as the execution evolves over a large state space. This attitude leads us to one particular way of understanding how a program computes what it claims to compute. The essential idea would be to develop programs with a clear understanding of the initial conditions of the program, the structure of the expected outcome, and the rules that guide the transformation of values of various variables that constitute the state of an executing program at any given point in time.

Ideally, we would like the program design to be guided by this principle. WHen it is not possible for some reason, we can use this very idea to figure out precisely how the execution of a program proceeds in its state space. With a precise knowledge of the requirements, we may attempt to find out logical relationships among state variables and states of an executing program. And this is what we wish to show in the following subsections.

4.1 The first few cycles

In the card shown below, we have highlighted two sections of the program. Our idea is to find out meaningful logical propositions that relate the values of variables. This proposition as asserted to hold whenever control reaches that point in the execution of the program. Most variables used in these logical propositions come from the program. However, we will see that we "discover" a few variables in order to explicate finer details of Python.

```
1  def collect_proc_names(src):
2    proc_names = []
3    prev_line = None
4    for line in src:                    (i = 0 ∧ prev_line = ⊥) ∨
5      if open_curly(line):                (1 ≤ i ∧ prev_line = src[i − 1] ∧ line = src[i])
6        pn = prev_line
7        for line in src:
8          if close_curly(line):
9            proc_names.append(pn)
10           break
11     else:                               0 ≤ i < |src| ∧ src[i] ≠ '{'
12       prev_line = line
13   return proc_names
```

What will be the state of the program variables when a open curly bracket marks the beginning of a statement block? We see this condition holds when the program reaches the

line 6 in the program text. The logical relation among the values of the state variables is shown in the card below:

```
1  def collect_proc_names(src):
2    proc_names = []
3    prev_line = None
4    for line in src:                    (i = 0 ∧ prev_line = ⊥) ∨
5    if open_curly(line):                  (1 ≤ i ∧ prev_line = src[i − 1])
6        pn = prev_line                  0 ≤ i ≤ |src| ∧ src[i] = '{' ∧ line = src[i])
7        for line in src:
8          if close_curly(line):
9            proc_names.append(pn)
10           break
11     else:                             0 ≤ i < |src| ∧ src[i] ≠ '{'
12       prev_line = line
13   return proc_names
```

```
1  def collect_proc_names(src):
2    proc_names = []
3    prev_line = None
4    for line in src:                    (i = 0 ∧ prev_line = ⊥) ∨
5    if open_curly(line):                  (1 ≤ i ∧ prev_line = src[i − 1])
6        pn = prev_line                  0 ≤ i ≤ |src| ∧ src[i] = '{' ∧ line = src[i])
7        for line in src:
8          if close_curly(line):
9            proc_names.append(pn)       1 < i ∧ src[i] = '}' ∧
10           break                         (∃ j. 0 < j < i ∧ src[j] = '{')
11     else:                             0 ≤ i < |src| ∧ src[i] ≠ '{'
12       prev_line = line
13   return proc_names
```

4.2 And then we find a bug

```
1  def collect_proc_names(src):
2    proc_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        pn = prev_line                        line = "{" ∧ i = 0 ⇒ proc_name = ⊥
7        for line in src:                      (i + 1) ≤ j < N ∧ line = src[j]
8          if close_curly(line):
9            proc_names.append(pn)             src[i] = "{" ∧ src[j] = "}" ∧ (i ≤ j < N)
10           break                             (i = 0) ⇒ ⊥ ∈ proc_names
11     else:
12       prev_line = line
13   return proc_names
```