Advanced Python Programming

Assignment 4

'Static Analysis and Complexity'

Today we are going to touch one of the most scaring and vague topics in computer science: Static analysis and Complexity. Generally it is given at the end of the course as advanced extension. Nevertheless, we are going to break it through without much of preparation. Static analysis (SA) is a broad term with quite varying purposes. But what you really need to remember is that (in most of cases) it equals to Static ***code*** analysis. Which means we analyse the source code as *ordinary text* without actually executing it. Complexity in its turn can be considered as one of the *metric* we can measure within SA realm. The purpose of the analysis may vary from simply determining the amount of code (LOC) and documentation coverage till spotting malicious patterns and extracting dependency graph among the files. Actually you can create endless number of metrics once you comprehend the fact that the code can be seen as a text (just an English one in our case) or, so to say, *structured literature*.

Now, our task is to implement so called Halstead's complexity. Which, by its original author's intention, is to quantitatively measure algorithm complexity by counting operators and operands directly from source code. As an object of analysis we will use Python code (no surprise we guess). Since the Python code itself is platform-independent so will our measurements which is really nice feature of static analysis in general. However, there is one big hurdle towards accurate results – that is *syntax sugar* or *implicity* that hides a lot of operations behind the scene. Anyway, whatever we've got and whether you believe it or not, we can use our measurements as a rough *maintenance* indicator. That is how difficult for us – as developers – to look after the code. Since the more operations and operands we see per n lines of code the more energy it takes the reader to apprehend from.

This assignment is going to be checked by machine! So it is *your* task to fit your program's output into a given format so that it will not crash under our execution environment, as well as dataset and simple parsing mechanism. Don't worry though, humans are there! So if something goes wrong we immediately check it by hands. Details will be given as we move along. What is important now:

- The **deadline** is the next Monday, 3:30pm (just before the lecture)
- The assignment is going to be challenging. So please, start small and grow slowly. Do not rush. We hardly expect you all to get the same results. You might get 10 points even if only 50% of the assignment is accomplished but you have to make your best (honestly)
- Use **Python 3.8**
- Upload your solution as an archive containing `main.py` and `answer.txt` (see below)
- Do not use **external libraries** and **regexp**. No exceptions! `tokenizer` wouldn't be a bonus either but if it's too challenging for you we will accept it as OK.

What about our grading system? We keep it the same: **5 tasks** and **10 points** for the whole assignment. By numerous demand on *explicit* criteria for the assessment, we state

it clearly – they are *adaptive*. It means that they might *change* while assessing you all. Since any arbitrary text (and even specification) has infinite number of interpretations we adjust our criteria based on how most of you "got it". We hope it is enough and we are ready to start:

1. For a given set of **distinct operators** ($\eta_1 = 20$):

   ```
   if, elif, else, try, for, with, return, def, import, except,     10
   calls, arithmetic (+, -, /, *), logic (==, !=, and, not),       9
   assignment (=)                                                  1
   ```

   Create a program that counts them and summarize **the total number of operators** (N_1) from a source given as stdin:

   ```
   $ python main.py < bc.py
   [operators]
   if: 10
   elif: 3
   else: 6
   try: 5
   for: 6
   with: 2
   return: 4
   def: 7
   import: 4
   calls: 33
   arithmetic: 14
   logic: 6
   assign: 30
   N1: 130
   ```

   `bc.py` here is just your program from previous assignment. It's reasonable – as a good software engineer – to analyse your own working. `calls` basically means you need to count anything that looks like a function call. Example:

   ```
   print(foo)
   a = list(dict(bar))


   calls: 3
   assign: 1
   ```

   When it comes anything to count, be sure you don't include what within the literals: (that might be tricky!)

   ```
   a = "a = 1 + 3 + 4" + "== 8 and != 9"


   arithmetic: 1
   logic: 0
   assign: 1
   ```

2. Now, extend your program to count **distinct operands** ($\eta_2 = 5$) and summarize **the total number of operands** (N_2). Operands are identifiers denoting all kinds of program entities (variables, functions, classes etc.) as well as literals like 1, 7.8, "string" etc. Two little exceptions here: (1) we added docstrings and inline comments because they directly affect our reading perception; (2) we simplified the task by giving *fixed* number of distinct operands η_2 (in our case they are just operand categories) so we do not impose you to check if occurrence is unique within subgroup. Let's dive in:

```
$ python main.py < bc.py
...
N1: 130

[operands]
docstrings: 6                                                    1
inlinedocs: 20                                                   1
literals: 13                                                     1
entities: 27                                                     1
args: 10                                                         1
N2: 76
```

We know that's too abstract! Here is more:

```
$ cat test.py
def main():
    '''
    Main function of the shell
    '''
    do_exit = False
    while not do_exit:
        # get absolute current path,
        # then truncate every folder to 1 char (2 if starts with a ".")
        path = os.path.abspath(os.getcwd())
        path = "/".join([i[0] if i[0] != "." else i[0] + i[1] for
            i in path.split("/") if len(i) > 0])

$ python main.py < test.py
...
[operands]
docstrings: 1
inlinedocs: 2
literals: 8
    "/"         x2
    "."         x1
    0           x4
    1           x1
entities: 4
    def main()  x1 (new symbol)
    do_exit =   x1 (new symbol)
```

```
    path =      x2 ("new" symbol)
args: 8
    os.path.abspath(..) x1
    join([..])  x1
    i[0]        x3 (yes indices should be counted as well)
    i[1]        x1
    split("/")  x1
    len(i)      x1
N2: 23
```

(note: all the lines with `x1, x2, x3, etc.` are comments. You are not supposed to actually print it! We need only *numbers* from you!)
(hint 1: hm...maybe you can use number of assignments and defs from previous task to get the number of entities?)
(hint 2: seems like without recursion you can't count args properly so leave it to the end)

3. Given the data you've got from previous two, derive the following:

   Program vocabulary: $\eta = \eta_1 + \eta_2$

   Program length: $N = N_1 + N_2$

   Calculated program length: $L = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

   Volume: $V = N \times \log_2 \eta$

   Difficulty: $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$

   Effort: $E = D \times V$

   And put all these data as a new section in stdout. Here is how:

```
$ python main.py < bc.py
[operators]
if: 10
...
assign: 30
N1: 130

[operands]
docstrings: 6
...
args: 10
N2: 76

[program]
vocabulary: x
length: x
calc_length: x
volume: x
difficulty: x
effort: x
```

4. Give us an intuitive explanation *why* the formula of estimating program length uses *logarithm to base 2*? Why not something else? Why base two? Why exactly logarithm? What's the reason? Is that somehow related to how we think? Put your answer inside the `answer.txt` along your `main.py`.

5. The last task is to fit your program into our assessing format. Make sure your code is neat, commented and documented where necessary, output is formatted correctly & neatly and finally be sure you're using Python 3.8.

Congratulations! You're almost real computer scientist who can measure complexity of any algorithm in the wild and that *proves* you are capable of doing static analysis!

Resources

1. https://en.wikipedia.org/wiki/Halstead_complexity_measures
2. https://github.com/pyenv/pyenv
3. Slides 26-28 from 'Lecture 2. Slides' (available at moodle)