Documentation for Solution to the Coding Test

Abstract

This is the readme.md documentation for the solution to the Coding Test. The purpose of this documentation is to explain the program logic and algorithm of one of the working solutions to the test. This document will divide into several sections : 1) background information, explaining the necessary background of the solution, choosing programming language etc.; 2) program logic and algorithm, explaining the idea behind the solution and how to tackle the problem; 3) assumption, explaining any assumptions made; 4) installation: explaining how to install the program and test run it; and 5) conclusion : a final conclusion of the solution and any enhancement worth to be done in the future.

Background

The coding test requires a software solution to read a text file which comprises of one or several lines of text. The last line of the file is a "search term" which is used as a searching criterion to check if any line of the text above would contain this search term. The software program will then extract these lines and display them on screen, if any.

Here is an example of an input :

I am a boy
You are!!a girl
We are friends
How are you today?
ou

So, the last line "ou" would be the "search term" and the program will need to find if there is any line above containing this "ou" word. The required output would then be :

[You are a girl]
[How are you today]

i.e., with words separated by single space and with square brackets outside. So

technically it involves 1) finding the line with the matching term, 2) removing the non-alphabet parts (e.g., 0-9 and special characters like !, @, #, $, % etc.) and 3) display the matching line with square brackets outside.

The Python programming language was chosen for this task because it is a high-level language which is easy to write and understand. Also, it is cross-platform and straightforward to run, both in installation and implementation, and contains powerful API (application programming interface) to manage many high- or low-level problems, including text manipulation.

Algorithm

The algorithm is divided into 4 parts:

1) Read the input file and display the file name
2) Identify the search term at the last line and display it
3) Check if any lines above the search term match with it
4) Extract the matching lines with only the alphabets involved and display them as output

While step 1) can use a fixed input file with its file name hard coded inside the program or, let the input as a parameter and changes every time as the user would like to. The latter one was chosen as this is more flexible in terms of flexibility and usability.

The Python syntax is :

```
inputFile = sys.argv[1]
```

where [1] denotes the first input argument when running the Python program.

Step 2) would involve running through all the lines in the file and extracting the last line as the “search term”. Since the file is read in a line-by-line manner into a list, when the end-of-file is reached, the line containing the search term is at the last line. To extract this line, we assign the last item in the list to a variable say, searchTerm :

```
searchTerm = lines[-1]
```

where the index [-1] denotes the last item inside the list.

Step 3) would involve matching the lines with the search term. To do this, we just need a conditional statement :

```
if (searchTerm in lines)
```

which checks if the search term exists inside the lines. To avoid the search term itself (which is at the last line) being read together which is not necessary, we modify the logic to include this checking :

```
if (searchTerm in lines) & (searchTerm != lines):
```

The "!=" simply means "NOT equal to".

The last step would involve extracting the matched lines and displaying them as output. This is probably the trickiest part of the program since only alphabets are extracted but not special characters like "!", "@", "#", "$"… etc., meaning these special characters are to be removed before they are displayed as output.

We used a Python function called "regular expression" with "substitution" to achieve this. For example, to substitute a "?" with a space :

```
myline = re.sub("?", " ", lines)
```

The "re" is the function for "regular expression" whereas the "sub" is the function's "method" for substitution.

Now to extract only the alphabets, we used "a-z" and "A-Z" to denote all alphabets and the operator "^" to denote "NOT", and the Python syntax to do this is :

```
myline = re.sub("[^a-zA-Z]", " ", lines)
```

which means substituting all characters which are NOT alphabet in the line with space, and then store the result to a variable called "myline".

But this would not be a perfect solution to the problem. To realize why this is so, consider the line with the following structure :

1111tired1111of1111sitting1111

which contains the words “tired of sitting” separated with a series of “1111” (four number “ones”). So, with the “re.sub” function only, the output will be :

[tired of sitting]

which is not what we want.

To tackle this, we need to remove extra spaces but retain only one single space between the words. In Python, the “re.sub” function has a “+” parameter, meaning “repetition”. So, to remove multiple spaces, we call the function again with a slight modification :

```
myline = re.sub(' +', ' ', myline)
```

meaning we are going to substitute multiple spaces with one single space, then store it again in the variable “myline”.

Finally, we output the result with the “strip()” function to truncate any leading and trailing spaces :

```
print('[' + myline.strip() + ']')
```

Assumptions

It was assumed that the input files are plain text only which contains no special control characters inside which are invisible. It means if the input is from a format made by special software like Word, or an image file like .jpeg etc., then the program won’t work as expected. It is because we only want to demonstrate the working principle of the program logic in a straightforward way, instead of tailor make it to tackle a special kind of files, other than a plain text one.

Installation and Running

The program, testing file and documentation file are all zipped into the file match.zip. Please unzip the file and you will get the following :

- match.py (the Python program source code)
- README.md (this file)
- sample1.txt (sample input file)
- sample2.txt (sample input file)
- sample3.txt (sample input file)
- sample4.txt (sample input file, empty)
- solution.bat (batch file to run the program)

To run the program in Windows DOS prompt environment :

1) Install the Python compiler/interpreter
2) Create a folder, unzip and copy all the above source code and testing files into it
3) Open DOS prompt
4) Change to the folder created in 2) above
5) Type : solution<space>sample1.txt<Enter>

The content of the solution.bat file has just one line :

```
python match.py %1
```

It will substitute the parameter %1 with the name of the sample text file. For example, if you want to use sample3.txt to test it, type the following :

solution<space>sample3.txt<Enter>

The program will run and display the output in the DOS prompt.

Conclusion

The algorithm of the program is simple : just input a plain text file, parse it, do some checking and then output a result. Although the program does have some checking on the input e.g., if it is a text file or not, it cannot avoid an erroneous input with a e.g.,

binary file or other types of file incapable of parsing by the program. Also, the program does not allow the user to try another sample input without leaving it, which is a nice-to-have feature. As a future enhancement, one might want to modify the program with more features like a GUI environment, or having a loop to ask for another test file input before the program ends, among other things.

Note : you can try any plain text file with this program. The samples included are for illustration only.

***** END OF README *****