Complex Parsing of Time Logs

Jacob Jones

December 1, 2025

1 Abstract

This problem stems from a project assignment given by Dr. Weidong Xiong for Comparative Program Languages. This project involved a complex string parsing problem among many files. The parsing required retrieving a time interval found within a timelog file. A solution to this problem required the use of pattern recognition due to the big restriction of not using any python modules except for the sys module. The solution proposed runs within parameters and produces good output.

2 Introduction

This problem was a project assignment given by Dr. Weidong Xiong for Comparative Program Languages. The problem entailed parsing five given "time log" files into their respective time. Each file would produce an output in the form of hours and minutes to standard output. The format of these files differed slightly but followed the same principle. [Date] [Time Intervals] [Message] (e.g. 2/23/12: 9:10pm - 11:40pm getting familiar with Flash). A line like this should return a time of two hours and thirty minutes. Then the next time interval should be read and added to the total number of hours and minutes. Iterating like this until the end of the file is reached and a total number of hours and minutes is written to standard output. With how strong modern day programming languages have come, a solution to this problem could be solved simply, so a number of restrictions were put in place to inspire unique solutions. The restrictions were as follows...

- The program must be written in the Python programming language
- The program should report the line number if there is something wrong with the format of the time data in that line that your Python program cannot parse. We cannot assume that there will be a time data value in each line
- Your program starts to count the time after it finds the hard-coded "Time Log:" (case insensitive, there is a space between Time and Log)
- The "pm" and "am" should be case insensitive, i.e., 9:10pm or 9:10PM or 9:10pM or 9:10Pm are all valid time values.
- The only Python module you can use in your source code is the "sys" module. Do NOT use other modules such as "re", "datetime", "pandas", "numpy", "dateparser", etc.
- We assume there will be no space within "9:10pm"
- Please write a function "GetTimeValue" with appropriate parameters to parse the time value like "9:10pm"
- Please write another function "GetTimePeriod" with appropriate parameters to parse the time period like "9:10pm - 11:40pm". It will call the function "GetTimeValue" to parse the time values "9:10pm" and "11:40pm".

3 Methodology/Scheme/Algorithm Design

A problem such as this could be handled in a multitude of ways. Without the use of modules I was forced to handle string parsing in a creative way. The solution I found the most intuitive was to look for patterns in the time entries. I noticed that each entry was formatted as a date followed by a time then a message afterwards. With this knowledge my initial idea was to break down each file into a series of lines, iterate through each line until hitting a space. While doing this each character is added to a string named preprocess. At this point preprocess is a string containing just the date at the beginning. I could remove this substring from the current line then iterate again until hitting another space. This string would be the first time given in the current line. I could then continue iteration until hitting another space which would be the second time given in the current line. These time would then be converted into military time so they could be subtracted to get a true time value between the two.

This initial thought needed refinement. As per the restrictions listed above, some lines do not have any time intervals in them. This was done by adding new line characters into the message which is a real possibility in a real world scenario. Another issue is if the line has a time interval but no date associated with it. This would mean the file was opened then closed then reopened and reclosed on the same date. Converting to military times only partially fixes the issue of time conversions. The program also needs to be able to handle time changes from late night to early morning. An example would be the first time being 9:00am and the second being 9:00pm. Subtracting the two would yield zero hours which is incorrect. The true value is twenty-four hours.

4 Implementation with Findings

The first thing to implement was global variables that would be used in the program.

```
import sys

totalCount = 0
currentLine = ""

if len(sys.argv) < 2:
    print("Usage: parse input_file")
    exit()
try:
    file = open(sys.argv[1], 'r')
except FileNotFoundError:
    print('Could not open file')
```

This imports the only module allowed in the project as well as initializes a totalCount global variable and a currentLine global string. The total count is going to be the final count of all the times added up and the currentLine variable is the line that needs to be parsed. We also open the file via a command line argument provided by the user. If no file is given the program exits with the message "Usage: parse input file". If the file cannot be found it exits with the message "Could not open file". Next is the getLine function which is in charge of getting the next line in the program for parsing.

```
def getLine(file):
    global currentLine
    currentLine = ""
    currentLine = file.readline()
```

This function takes in a file and clears the current line and replaces it with the next line in a sequence. Another important function is the truncate function.

```
def truncate(value):
    value = int(value * 10**2) / 10**2
    return value
```

This function looks strange only because of the module's restriction. Python only supports truncation of floats using their math library so I needed to get resourceful. This function essentially multiplies the given value by 10^2 and casts that number as an integer. This essentially moves the decimal place to the right twice, padding it with zeros. Then we divide that number by 10^2

which restores the number to the original value, effectively truncating it. The reason this is so important is because we do not want the python interpreter rounding the minutes of a given time. A number like 1.015 should translate to an hour and one minute. If python is allowed to round this number then it would be one hour and two minutes which would throw off the output. One of the required functions is GetTimeValue, which parses a time like 9:10pm and 11:40pm into two respective military times.

```
def GetTimeValue(time):
    # parse a value like 9:30pm
    if time[-2:] in ("am", "pm"):
        ap = time[-2:]
        time_part = time[:-2]
    else:
        ap = time[-1] + 'm'
        time_part = time[:-1]
    hours, mins = time_part.split(':')

    hours = float(hours)
    mins = float(mins)

    if ap == 'pm':
        if hours == 12:
            return truncate(hours), truncate(mins)
        hours = float(hours)
        mins = float(mins)
        hours += 12.00
        return truncate(hours), truncate(mins)
    if ap == 'am':
        if hours == 12:
            hours = hours - 12.00
            return truncate(hours), truncate(mins)
        return truncate(hours), truncate(mins)
```

This function takes in a given time provided by GetTimePeriod. The first step is to break the time into two components. The actual time provided and if it is an "am" time or a "pm" time. To do this we check the last two characters of the time string. If it is "am" or "pm" then we set the variable "ap" to "am" or "pm". Then we set a variable named "time part" to what comes before the last two characters. If "am" or "pm" are not in the time, then we append an 'm' to the end of the string and set it equal to "ap". "Time part" is then set just like in the first example. The reason we need to append the 'm' at the end is due to how we parse the string and pass in the time.

This will be explored later when talking about the GetTimePeriod function. We break down the time part into two variables, separating them by the colon that connects them, “hours” and “mins”. We also need to cast them as floats because they are currently strings. We then check to see if “ap” is set to “pm”. If it is then we check to see if hours are equal to twelve, if it is then we return the truncated hours and minutes. If not then we add twelve to the hours and return hours and minutes truncated again. This converts the time into military time and ignores if it’s twelve pm because that is a valid military time. If the “ap” is set to “am” then we check to see if the hours is equal to 12 and if it is subtract twelve from hours then return truncated hours and minutes. Next is the GetTimePeriod function.

```
def GetTimePeriod():
    # to parse 9:30pm - 11:45pm
    # it will call GetTimeValue() in order to
    # return 9:30pm and 11:30pm seperatly
    global currentLine
    preprocess = ''
    part1 = ''
    part2 = ''
    for c in currentLine:
        if c != ' ':
            preprocess += c
        else:
            break
    if preprocess == "":
        currentLine = currentLine.strip(' ')
    else:
        currentLine = currentLine.replace(preprocess + ' ', "")
        currentLine = currentLine.strip()
    for c in currentLine:
        if c == '-':
            part1 = '0'
            part2 = '0'
            break
        if c != ' ':
            part1 += c
        else:
            break
    if currentLine == '' or not currentLine[0].isdigit():
        return 0
    currentLine = currentLine.replace(part1 + ' ', "")
    for c in currentLine:
        if c != 'm':
```

```
            part2 += c
        else:
            break
    part2 = part2.replace("- ", "")
    if not part1[0].isdigit():
        return 0
    valuea, valueb = GetTimeValue(part1)
    valuec, valued = GetTimeValue(part2)
    start = (float(valuea) * 60) + float(valueb)
    end = (float(valuec) * 60) + float(valued)
    if end < start:
        end = end + (24 * 60)
    return (truncate(end) - truncate(start)) / 60
```

This function is the most complex due to the heavy use of logic and loops, however it can be broken down into four simple procedures.

1. Preprocess the string
2. Get the first time interval
3. Get the second time interval
4. Return the values differences in hours and minutes

For the first part we initialize three strings named preprocess, part1, and part2. Then we iterate through the string until reaching a space character. While doing this we add the character up until the space to the string preprocess. After reaching the space, we remove it from the current string. We also check to see if after the preprocess runs if it is empty. If it is then we remove the white space at the beginning. The second step is similar to the first where we iterate through the string until reaching a space character. Adding everything before the space to variable part1. Following the similar semantics we continue iterating through the string skipping over the '-' character and adding it to part2. After that we run both numbers through GetTimeValue. The time values are returned to the function as a tuple in hours and minutes. We then convert the hours into minutes and combine with minutes to get a total number of minutes. Making sure to split them into a start time and an end time. After that we check to see if the end time is less than the start time. This would mean that a full day passed between the times, meaning we need to add twenty-four hours in the form of minutes. We can then pass this value to the main function.

```
def main():
    global totalCount
    global file
    global currentLine

    getLine(file)
    # Find time log
    while currentLine.lower() != 'time log:\n':
        getLine(file)
    # Now we are after time log
    while currentLine != "":
        getLine(file)
        value = GetTimePeriod()
        truevalue = float(value)
        truevalue = truncate(truevalue)
        totalCount += truevalue
        hours = int(totalCount)
        minutes = int((totalCount - hours) * 60)
    print(f'{hours} hours and {minutes} minutes')
```

This is the main driver of the program and is in charge of calculating the total number of hours and minutes. First we run getLine in order to get the first line of the file. Then we keep getting the next line until we hit the "time log:". This is in compliance with the restriction listed above. Once we hit that value, we keep getting the next line until the line is blank. Each line is then passed through GetTimePeriod in order to get the final number in terms of minutes in that specific line. We then add those numbers to a totalCount variable. We then split the total into hours and minutes. After all the lines have been read in we print to standard out the in terms of hours and minutes.

5 Time Complexities

The efficiency of these functions are a bit hard to judge due to the nature of loops within the program. Initial thinking leads me to believe that the worst case is $O(n^3)$, due to the three for loops. However the loops are not nested meaning each loop runs independently of each other. The only thing that changes is the input string that is being run on. This means that each loop is only $O(n)$, and when judging the notation of a function, we take the highest order, in this case $O(n)$.
Worst Case: $O(n)$
Average Case:$O(n)$
Best Case: $O(n)$

6 Timed Examples

To test efficiency of this program, I used the "time" application found within most linux distributions. This application runs a given program and returns the real time, the user time, and the system time (INSERT MAN PAGES CITATION). The real time is the real world seconds used. The user time is the real time that was used by that specific linux user, and the system time is the amount of CPU time used for this application.

File size: 4.6kb

Real time	User time	Sys time
0m0.016s	0m0.008s	0m0.007s

File size: 248Mb

Real time	User time	Sys time
0m10.373s	0m10.350s	0m0.018s

File size: 1.3Gb		
Real time	User time	Sys time
0m53.572s	0m53.465s	0m0.101s

7 Conclusion

When dealing with string parsing, there are many ways to handle a complex problem such as this. The solution proposed runs within the parameters of the project and performs well when outside of established parameters. It is also structured in a way that makes future modifications straightforward, allowing additional formats or edge cases to be incorporated with minimal disruption. Overall, the approach provides a reliable balance between simplicity, flexibility, and performance.