Software Engineering 265
Software Development Methods
Fall 2021

Assignment 2

Due: 27th October 2021, 11:55 pm by submission via Brightspace
(no late submissions accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the virtual machines you installed as part of Assignment #0 (which we have taken to calling Senjhalla). This is our "reference platform". This same environment will also be used by the course instructor and the rest of the teaching team when evaluating submitted work from students.

All test files for this assignment are available on the SENG server in seng265.seng.uvic.ca:/seng265work/A2. Any programming done outside of Senjhalla might not work during evaluation. A starter file is NOT provided for this assignment. Make sure that your **submitted file is named** "process_cal2.py".

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden.** Code-similarity analysis tools are used to examine submitted work.

Objectives of this assignment

- Learn or review basic features of the Python language.
- Use the Python 3 programming language to write a less resource-restricted implementation of process_cal (**but without using regular expressions or user-defined classes**).
- Use Git to manage changes in your source code and annotate the evolution of your solution with messages provided during commits. Remember, the only acceptable of submitting your work is in brightspace.
- Test your code against the 15 provided test cases.

process_cal2.py: Returning to the problem

For this assignment please use the description of the problem as provided at the end of this document, and use the test files provided. Some of the limits that were placed on certain values are no longer needed (e.g., maximum number of events, maximum line length, etc.).

The arguments used for the Python script are the same as used for assignment #1. That is, you will indicate the range of dates to be used to generate the schedule by providing "--start" and "--end" arguments. However, the executable will now be named "process_cal2.py" (and **not** "process_cal.py").

```
./process_cal2.py --start=2021/2/14 --end=2021/2/14 --file=one.ics
```

As with the first assignment, all output is to stdout. You must test the output of your program in the same manner as with assignment #1 (i.e., using diff).

However, we will place four different kinds of constraints on your program.

1. For this assignment **you are not to use regular expressions**. We will instead use these in assignment #4 in order to write a somewhat more powerful version of the program that processes more complex .ics files.
2. You must **not write your own classes** as this will be work for assignment #4.
3. You must **not use global variables**. Note that this is not a hardship as it is easier in Python to pass mutable data structures as function parameters.
4. You must **make good use of functional decomposition**. Phrased another way, your submitted work **must not** contain one or two giant functions where all of your program logic is concentrated.

Exercises for this assignment

1. The **only** acceptable way to submit your work is by submitting **process_cal2.py** in the assignment 2 section of **brightspace.**
2. A starter file is not provided for this assignment. You are to write command processing in your solution yourself. Refer to the example command earlier in this file and "TESTS.md" for appropriate command line input. Your solution must accommodate all of these possible command line inputs.
3. Make sure to use the test files provided as part of this assignment.
4. **Keep all of your code in one file (process_cal2.py) for this assignment**. In assignment #4 we will use the multiple-module and class features of Python. Please ensure you also respect all of the other constraints described earlier in this document.
5. Use the test files and listed test cases to guide your implementation effort. Refrain from writing the program all at once, and budget time to anticipate when "things go wrong".
6. For this assignment you can assume all test inputs will be well-formed (i.e., our teaching assistant will not test your submission for handling of input or for

arguments containing errors). Assignments 3 and 4 may specify error-handling as part of the assignment

7. As in assignment 01, **use diff** to check the output to stdout from your program. Do not rely on visual inspection. Our eyes are very poor at comparing white space. Mismatching white space will lead to tests failing and lost marks. USE DIFF.

What you must submit

- A single Python source file named "process_cal2.py" submitted to the assignment 02 section of brightspace.
- No regular-expressions, global variables, or user-defined classes are to be used for assignment #2.

Evaluation

Our grading scheme is relatively simple.

- "A" grade: A submission completing the requirements of the assignment which is well-structured and very clearly written. All tests pass and therefore no extraneous output is produced.
- "B" grade: A submission completing the requirements of the assignment. process_cal2.py runs without any problems; that is, all tests pass and therefore no extraneous output is produced.
- "C" grade: A submission completing most of the requirements of the assignment. process_cal2.py.py runs with some problems.
- "D" grade: A serious attempt at completing requirements for the assignment. process_cal2.py.py runs with quite a few problems; some non-trivial tests pass.
- "F" grade: Either no submission given, or submission represents very little work, or no tests pass.

process_cal2, version 2

The program will be a Python program. Its name must be “process_cal2.py” and it must be found in the assignment 2 section in Brightspace.

Input specification:

1. All input is from ASCII test files.
2. Data lines for an “event” begin with a line “BEGIN:VEVENT” and end with a line “END:VEVENT”.
3. Starting time: An event’s starting date and time is contained on a line of the format “DTSTART:<icalendardate>” where the characters following the colon comprise the date/time in icalendar format.
4. Ending time: An event’s ending date and time is contained on a line of the format “DTEND:<icalendardate>” where the characters following the colon comprise the date/time in icalendar format.
5. Event location: An event’s location is contained on a line of the format “LOCATION:<string>” where the characters following the colon comprise the string describing the event location. These strings will never contain the “:” character.
6. Event description: An event’s description is contained on a line of the format “SUMMARY:<string>” where the characters following the colon comprise the string describing the event’s nature. These strings will never contain the “:” character.
7. Repeat specification: If an event repeats, this will be indicated by a line of the format “RRULE:FREQ=<frequency>;UNTIL=<icalendardate>”. The only frequencies you must account for are weekly frequencies. The date indicated by UNTIL is the last date on which the event will occur (i.e., is inclusive). Note that this line contains a colon (“:”) and semicolon (“;”) and equal signs (“=”).
8. Events within the input stream are not necessarily in chronological order.
9. Events may overlap in time.
10. No event will ever cross a day boundary.
11. All times are local time (i.e., no timezones will appear in a date/time string).

Output specification:

1. All output is to stdout.
2. All events which occur from 12:00 am on the --start date and to 11:59 pm on the --end date must appear in chronological order based on the event's starting time that day.
3. If events occur on a particular date, then that date must be printed only once in the following format:

   <month text> <day>, <year> (<day of week>)

   Note that the line of dashes below the date must match the length of the date. You may use Python's datetime module in order to create the calendar-date line.
4. Days are separated by a single blank line. There is no blank line at the start or at the end of the program's output.
5. Starting and ending times given in 12-hour format with "am" and "pm" as appropriate. For example, five minutes after midnight is represented as "12:05 am".
6. A colon is used to separate the start/end times from the event description
7. The event SUMMARY text appears on the same line as the even time. (This text may include parentheses.)
8. The event LOCATION text appears on after the SUMMARY text and is surrounded by square brackets.

Events from the same day are printed on successive lines in chronological order by starting time. Do not use blank lines to separate the event lines within the same day.

In the case of tests provided by the instructor, the Unix "diff" utility will be used to compare your program's output with what is expected for that test. Significant differences reported by "diff" will result in grade reductions.