

COMP 690 Data Fusion Fall 2009 Programming Assignment 2

1. Write a Python script that reads from a free-format text file of UFO sightings in the Greensboro area for a day. The file contents are partitioned into sentences that describe sightings. Each sentence contains a UFO identifier, the time of the sighting, and the latitude and longitude of the sighting. An identifier consists of the letter 'B' or 'F' followed by two decimal digits. The time may be in military time (0 to 23 hours) or it may be followed by 'AM', 'am', 'PM', or 'pm' preceded by zero or more spaces. In any case, the time includes only the hours and minutes (not the seconds), with the hours separated from the minutes by a colon. The hours may be given with one or two digits, but minutes are always expressed with two digits. Latitude is expressed in terms of degrees and minutes (but not seconds), separated by an underscore and followed by 'N' or 'S' preceded by zero or more spaces. The degrees may be expressed with one or two digits, and likewise for the minutes. Longitude is similar to latitude but with 'E' or 'W' instead of 'N' or 'S'. The following is an example input file. Note that two of the sentences are incomplete, one lacking the time, and the other the longitude. (All lines in this listing are terminated with newlines—they do not wrap.)

```
At 2:01PM, B56 was at 38_8N and 79_47 W. At 13:38, F43 was at 37_23N and 79_18W.
Fred reported B56 at 37_45 N and 79_35W around 1:29 PM. We saw F43 at 39_30N and
78_56 E. F43 was at 38_42N and 79_53W at 13:51. They reported B56 at 38_32N at
1:59 PM. We found F43 at 37_53N and 78_58W at 2:03 PM. B56 was at 38_23 N and
79_32W at 13:41.
```

There are only two identifiers in this file, but generally there could be any number of identifiers.

Your script should split this text into sentences and extract from each sentence the identifier, the time, the latitude, and the longitude. If a sentence is missing any of these components, it should echo the sentence to the terminal along with an indication of what is missing. The script should output to file all the valid sightings grouped by identifier. Under each identifier, the sightings should be sorted by time. Convert all times to the format HH:MM:SS (where SS is always 00), with HH in the range 00 to 23. Convert all latitudes and longitudes to the form DD_MM_X, where DD is the degrees, MM is the minutes, and $X \in \{N, S, E, W\}$. Output identifiers flush with the left margin, and indent sightings two spaces. Output each sighting on one line with the time (followed by a colon and a space) first, then the latitude (followed by a comma and a space), and finally the longitude.

When my solution¹ is run with the above file, the session is

```
E:\Old D Drive\c690f09\HW\HW2>prob1.py
[ We saw F43 at 39_30N and 78_56 E] is missing the time.
[ They reported B56 at 38_32N at 1:59 PM] is missing the longitude.
```

Note that the two incomplete sightings have been identified. The contents of the output file are

```
F43:
  13:38:00: 37_23N, 79_18W
```

¹ My solution is 90 lines, including a reasonable number of empty lines for readability. The code is not very complex: the deepest nesting is two levels of control structure.

```
13:51:00: 38_42N, 79_53W
14:03:00: 37_53N, 78_58W
B56:
13:29:00: 37_45N, 79_35W
13:41:00: 38_23N, 79_32W
14:01:00: 38_08N, 79_47W
```

You will find it useful to use the `time` class, which is available in the `datetime` module. To construct a `time` instance, pass the hour (an integer in the range 0-23) and the minutes (in the range 0-59) to the class instantiation operator. Conversion of a `time` object to a string gives a string in the format “HH:MM:SS”. For example,

```
>>> from datetime import time
>>> t = time(13, 30)
>>> t
datetime.time(13, 30)
>>> print t
13:30:00
```

You may implement your script anyway you want as long as it has the generality indicated in the description above. The following is a brief description of my solution, which you may find helpful.

The main function constructs a dictionary whose keys are all the identifiers found in the input and whose values are lists of dictionaries for the correspond sightings. Each sighting dictionary has keys ‘time’, ‘lat’ (latitude), and ‘long’ (longitude). The value associated with the ‘time’ key is an instance of the `time` class, and the values associated with the ‘lat’ and ‘long’ keys are strings already formatted as specified above.

The main function opens the input file for reading and compiles four regular-expression objects, one for each component of a sighting (including the identifier). It then reads in the file a line at a time. Each line ends with a newline, which must be removed before that line is concatenated with the string being built up from the file content. Use the string method `rstrip()`. If `str` is a string ending with ‘\n’, `str.rstrip('\n')` returns a string identical to `str` but with the terminal ‘\n’ removed. (This will not remove a ‘\n’ not at the end of the string.) For portability, you should actually use `str.rstrip('\r\n')`, which also works with UNIX systems.

Once you have the file contents as a string without newlines, split it at the periods, giving a list of sentences. The last element of this list is always a string with only whitespace (if it isn’t the empty string); you should discard it.

Now loop over the sentences. Get a match object on the sentence for each of the four patterns. Check each match to see whether it failed. If it did, inform the user and go back to the top of the loop. If not, assign the part of the string that was matched to a variable. It’s easiest if you here do conversions (to a `time` object or to a properly formatted latitude or longitude). Once all four components of a sighting are in hand, check whether the overall dictionary has a key identical to the identifier for this sighting. If so, append a dictionary (with keys ‘time’, ‘lat’, and ‘long’) to the list of dictionaries for this identifier. If not, add to the overall dictionary an association whose key is the identifier in the sighting and whose value is a singleton list containing the dictionary for this sighting. Note that the overall dictionary must be initialized to the empty dictionary before the loop. When the loop exits, the main function returns the overall dictionary.

The function that converts a string such as “2:30 PM” to a `time` object compiles a regular-expression object for the ‘AM’ or ‘PM’ (or ‘am’ or ‘pm’) part and another for the hours-and-minutes part. The match on the hours and minutes can provide a tuple whose elements are converted to integers. If a ‘PM’ is present, the hours should be incremented by 12 as long as they are not already 12; if ‘AM’ is present, the hours should be set to 0 if they are currently 12. After making these adjustments if needed, the hours and minutes can be used to construct the `time` object this function returns.

The function that adjusts a string giving the latitude or longitude to the required format can use a regular expression to extract the strings for the degrees, the minutes, and the direction (N, S, E, or W). The string returned is constructed by concatenating the parts back together, with a “0” added in front of a degree or minute value that is only one character long.

The function that outputs the information takes the name of the output file and the overall dictionary returned by the main function. After opening the file for writing, it loops over the key-value pairs in the dictionary. Recall that a key is a UFO identifier and the associated value is a list of dictionaries, each with the other three components of a sighting. The body of the loop outputs the key, sorts the list of dictionaries on the ‘time’ keys of the dictionaries, then loops over this list, outputting each sighting in the required format. To learn how to sort a list of dictionaries, see the set of slides on this subject.

2. For this problem, you will develop a simulation program that keeps track of friendly (“blue”), enemy (“red”), and neutral (“white”) air and ground forces in a given area of operation. The program reads a file that describes the position, velocity, and two killing ranges of each asset. One killing range indicates its lethality against air assets, the other its lethality against ground assets. (For example, an anti-aircraft battery might have 700 m for the first and 0 m for the second.) The blue, red, and white assets are stored in separate lists. The program goes through these lists and tells the user of (1) blue assets within the killing range of red assets, (2) red assets within the range of blue assets, and (3) white assets within the range of blue assets.

The program then asks the user for a length of time (in seconds) and updates the positions of all assets to where they will be after this amount of time has elapsed. Finally, the program goes through the lists again, telling the user the same three things as before (but now for the updated positions). Assume that this all happens on a very flat plane so that positions of ground assets can be given with two coordinates, although the positions of air assets require three coordinates. Assume there is some fixed origin to the coordinate system (lying in the plane occupied by the ground assets). Distances are in meters, times are in seconds, and velocities in meters per second.

Each line in the input file has the form

`<Id> <Color> <Type> <Position> <Velocity> <AirKillingRange> <GroundKillingRange>`

The fields are as follows.

- `<Id>` is a two-digit code.
- `<Color>` is “red” (indicating enemy), “blue” (indicating friend), or “white” (indicating neutral).
- `<Type>` is “air” (an aircraft) or “ground” (a ground vehicle).
- `<Position>`, if `<Type>` is “air”, is a three-tuple giving the position of the aircraft from the assumed origin as well as its altitude. If `<Type>` is “ground”, the `<Position>` is a two-tuple giving the position in the plane but not an altitude.

- $\langle \text{Velocity} \rangle$ is a two- or three-tuple, depending on $\langle \text{Type} \rangle$, giving the asset's velocity.
- $\langle \text{AirKillingRange} \rangle$ is the asset's killing range against air assets.
- $\langle \text{GroundKillingRange} \rangle$ is the asset's killing range against ground assets.

Define at least the following classes.

- `Asset`
- `Air`, inheriting from `Asset`
- `Ground`, inheriting from `Asset`
- `Position`
- `Position2D`, inheriting from `Position`
- `Position3D`, inheriting from `Position`
- `Velocity`
- `Velocity2D`, inheriting from `Velocity`
- `Velocity3D`, inheriting from `Velocity`

Define the components of positions and velocities to be managed attributes ("properties").