CSE 337: Scripting Languages (Fall 2019)

Unit Homework #2

Assignment Due: Sunday, December 15, 2019, by 11:59 PM

Getting Started

This assignment requires you to write Python code to solve several computational problems. To help you get started, we will give you a basic starter file for each problem. These files will contain *function stubs* and a few tests you can try out to see if your code seems to be correct (**note that the test cases we give you in the starter files are just examples; we will use different test inputs to grade your work!**). **Do not, under any circumstances, change the names of the functions or their parameter lists.** The automated grading system will be looking for functions with those exact names and parameter lists; if you change anything related to a function header, the grading program will reject your solution as incorrect.

Directions

Solve the following problem to the best of your ability. The automated grading system will execute your solution several times, using different data files each time. Each test that produces the correct/expected output will earn 1 or more points. This assignment contains 3 parts, worth a total of 70 points. **Note that not every problem may be worth the same number of points!**

- ⚠ Each starter file has a comment block at the top with various important information. Be sure to add your information (name, ID number, and NetID) to the first three comment lines, so that we can easily identify your work. **If you leave this information out, you may not receive credit for your work!**
- ⚠ Submit your work as a set of individual files (one per problem). **DO NOT** zip or otherwise compress your files, and **DO NOT** place all of your work in a single file. If you do so, we will not grade your work and you will receive a 0.
- ⚠ Every function **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).
- ⚠ Every function must explicitly ***return*** its final answer; the grading program will ignore anything that your code prints out. Likewise, do **NOT** use `input()` anywhere before the `if __name__ == "__main__":` statement); your functions should get all of their input from their parameters. Programs that crash will likely receive a failing grade, so test your code thoroughly before you submit it.
- ⚠ Blackboard will provide information on how to submit this assignment. You **MUST** submit your completed work as directed by the indicated due date and time. We will not accept (or grade) any work that is submitted after the due date and time, or that is submitted before you have signed the course's Academic Honesty agreement.
- ⚠ **ALL** code that you submit (for each problem) **MUST** be your own work! You may not receive assistance from or share any materials with anyone else, except for the instructor and the (current) course TAs.

Unit Homework 2: Data File Validation

Earlier this semester (during the "virtual lecture" on September 26), we started to discuss the creation of a Python game in the vein of the classic text adventure games of the 1970s and 1980s.[1] Creating data files for these kinds of games is tedious and error-prone. This assignment will develop a program to verify that a text adventure data file does not contain several types of common errors.

For the purposes of this assignment, a text adventure game is described by a single plain-text data file in a specific format:

```
ROOM <integer identifier>
TITLE <name of room>
DESC <up to 60 characters of text> -- this line may be repeated
EXITS <north> <east> <south> <west>
ITEMS <one or more strings representing the names of in-game objects) (optional)
PUZZLE <name of item needed to solve puzzle> (optional)
--- <separator indicating the end of a particular room's description
```

These lines always appear in the order shown above; the room's identifying number always appears first. The last room described in the data file **does not** contain a separator line at the end. **Assume the data file is always formatted correctly.**

A sample room description might look like the following:

```
ROOM 12
TITLE Living Room
DESC This is a large, dilapidated-looking room. A stained sofa
DESC that has clearly seen better days covers most of one wall;
DESC across from it, a large tube TV with a tin foil-covered
DESC antenna and a rotary knob for changing channels displays a
DESC screenful of static. The carpet was most likely dark green
DESC originally; now, it's a combination of faded green tatters
DESC and holes displaying the wooden floor beneath.
EXITS -1 8 17 -1
ITEMS screwdriver
---
```

represents a room ("Living Room") with the identifier 12. Room 12 has exactly two exits: one to the east (leading to room 8) and one to the south (leading to room 17). This room also contains one item (a screwdriver) that the player can pick up and use to solve a puzzle in another room. This room doesn't contain a puzzle.

Note the following:

- Every room **MUST** have a unique integer identifier.
- For this assignment, ignore any `TITLE` and `DESC` lines in the data file.
- The `EXITS` line is required, and contains four integers, separated by space. Each integer is either a room number (the identifier from a `ROOM` line), or -1 if there is no exit in that particular direction.
- The *optional* `ITEMS` lines contains one or more words; each word represents a single item that can be picked up.
- Likewise, the *optional* `PUZZLE` line holds the name of the single item needed to solve that puzzle.

[1] These types of games still continue to be created, under the name "interactive fiction", although their popularity has waned over the years.

Problem Description

(Place all of your work in a file named "validate.py". You may implement one or more helper functions, but they must be placed in the same file as the `validate()` function.)

Function Summary

Parameters: A string representing the name of a plain-text data file in the format described above.

Return Type: A dictionary whose keys are specific strings (described in each problem) and whose values are either lists or dictionaries, depending on the specific problem.

Complete the `validate()` function, which takes a single string argument: the name of a plain-text file in the format described above. This function returns a dictionary with a specific set of keys (strings), as described in the parts below. The structure of his program is up to you. You may find it easier to do everything in a single (enormous) `validate()` function, or you may wish to separate the tasks below into smaller helper functions and combine their results in `validate()` before returning the final dictionary.

Either way, we recommend processing and parsing the entire data file first (using a dictionary or some other data structure to hold the relevant data for each room) and then analyzing it according to the directions below. You may also want to create data structures to record information that may be relevant later.

Part I: Finding Phantom Rooms (20 points)

Room identifier numbers do not need to be consecutive, only unique (and ≥ 0), This makes it difficult to keep track of which integers are valid room identifiers and which ones refer to rooms that don't actually exist. Add a key named `"phantoms"` (with that exact spelling and capitalization) to the dictionary that your function will return. This key has an initially empty list as its value.

Using each room's list of possible exits as a source, create a list of room identifiers that don't actually exist. For example, if traveling west from Room 3 leads to Room 5, but Room 5 is not defined in the data file, append 5 to the `"phantoms"` list. Every room number in this list should only appear **once**.The order of items in the list doesn't matter; only its contents do.

Part II: Finding Missing Items (20 points points)

Remember that every puzzle requires a specific object to solve. If that item cannot be found somewhere within the game environment, then the game has a significant error. Add a key named `"missing"` (with that exact capitalization and spelling) to your solution dictionary. This key also maps to an initially empty list.

For each room that contains a puzzle, make sure that the associated object can be found somewhere within the set of rooms. If the required item is not present anywhere, append it to the "missing" list. You may assume that every item occurs exactly once, if it occurs at all within a game. Once again, the order of items in the list doesn't matter; only its contents do.

Part III: Mismatched Exits (30 points points)

One of the most critical problems in the design of a game map is making sure that room exits match correctly. For example, if Room 3's West exit leads to Room 5, then Room 5's East exit must lead back to Room 3 (we will assume that every exit is bidirectional, and that M.C. Escher has not tinkered with the map). Add a key named `"mismatches"` (with that exact capitalization and spelling) to your solution dictionary. This key also maps to an initially empty list. As with the previous parts, the order of items in the list doesn't matter; only its contents do.

For each room, check each of its four exit directions. If the exit in a given direction is not -1, check to see if the value provided actually matches a known room. If it does **NOT** lead to a known room, append a tuple of the form (*room number*, *exit direction index*, `"unknown"`) to the list for `"mismatches"` (e.g., `(2, 1, "unknown")`).

If the room's reciprocal (matching opposite) exit doesn't match the current room, append the tuple (*room number*, *exit direction index*, `"mismatch"`) to the `"mismatches"` list, e.g., `(2, 1, "mismatch")` (later, you'll probably add a similar tuple for the other room). You **DO NOT** need to worry about the case where several rooms have "one-way" exits that all lead to the same exit in a specific destination room.

Hint: If you store each room's exit destinations in a 4-element list (with 0 representing north, 1 representing east, etc.), then each exit's reciprocal (opposite) exit direction is its number plus 2, modulo 4. For example, the south exit has index 2. (2 + 2) % 4 is 0, which is the index for the north exit of the connecting room.

Sample Program Output

```
validate("map1.txt") produces:
{'phantoms': [], 'missing': ['hat'], 'mismatches': [(19, 1, 'mismatch'),
 (19, 2, 'mismatch'), (14, 1, 'mismatch'), (14, 2, 'mismatch'), (29, 2, 'mismatch'),
 (29, 3, 'mismatch'), (17, 0, 'mismatch'), (17, 2, 'mismatch')]}

validate("map2.txt") produces:
{'phantoms': [23, 29], 'missing': [], 'mismatches': [(27, 0, 'mismatch'),
 (27, 1, 'unknown'), (27, 3, 'mismatch'), (7, 0, 'mismatch'), (7, 1, 'mismatch'),
 (7, 3, 'mismatch'), (3, 0, 'mismatch'), (3, 1, 'mismatch'), (3, 2, 'mismatch'),
 (12, 0, 'mismatch'), (12, 3, 'mismatch'), (19, 1, 'unknown'), (19, 2, 'mismatch'),
 (19, 3, 'mismatch')]}

validate("map3.txt") produces:
{'phantoms': [], 'missing': [], 'mismatches': [(2, 1, 'mismatch'), (25, 2, 'mismatch'),
 (25, 3, 'mismatch'), (30, 0, 'mismatch'), (30, 2, 'mismatch'), (17, 3, 'mismatch')]}

validate("map4.txt") produces:
{'phantoms': [36, 20, 39, 8, 16, 25, 23, 38, 26, 5, 18, 4, 9, 10, 33],
 'missing': ['wrench', 'battery'], 'mismatches': [(6, 0, 'mismatch'),
 (6, 1, 'unknown'), (6, 3, 'mismatch'), (30, 0, 'mismatch'), (30, 1, 'mismatch'),
 (30, 2, 'unknown'), (34, 0, 'unknown'), (34, 1, 'mismatch'), (34, 2, 'unknown'),
 (34, 3, 'mismatch'), (37, 0, 'mismatch'), (37, 1, 'mismatch'), (37, 2, 'unknown'),
 (37, 3, 'unknown'), (29, 0, 'mismatch'), (29, 1, 'mismatch'), (29, 2, 'mismatch'),
 (29, 3, 'mismatch'), (27, 1, 'unknown'), (27, 2, 'unknown'), (27, 3, 'unknown'),
 (19, 0, 'mismatch'), (19, 1, 'unknown'), (19, 2, 'unknown'), (19, 3, 'unknown'),
 (22, 0, 'unknown'), (22, 1, 'unknown'), (22, 2, 'unknown'), (22, 3, 'mismatch'),
 (40, 0, 'unknown'), (40, 2, 'unknown'), (40, 3, 'unknown'), (32, 0, 'unknown'),
 (32, 1, 'unknown'), (32, 2, 'unknown'), (32, 3, 'unknown')]}
```