Homework 3

Due: Wednesday, April 12, at 11:59 PM. Submit your completed Python source file through Blackboard (multiple submissions are permitted; we will only grade the last/latest submission). Submissions that do not execute due to syntax or other errors will receive a 0.

Description

Many years ago (in the 1970s and 1980s), most computer games didn't have fancy graphics or elaborate controls. Instead, they took the form of "text adventures" like *Adventure* (AKA *Colossal Cave Adventure*) and the *Zork* series, where the game presented written descriptions of various locations in the game and players typed in commands to move around and solve puzzles (this genre still exists today under the title of "interactive fiction"). Some of these games became extremely elaborate, featuring dozens or hundreds of in-game locations and the ability to understand complicated commands (e.g., "open the door with the green key", not just "open door"). For this assignment, you will write a parser that reads in a data file written in a special format and runs the resulting text adventure game. We won't create anything as sophisticated as the classic text adventure *Zork*, but we will still support a number of basic commands and simple puzzles.

Note that the following instructions are ***long***, but if you follow them closely, you will find that the actual solution process for this assignment is not extremely complicated. It mainly consists of a number of loops and nested `if-elif-else` chains. ***Even so, start this homework early; don't wait until the last minute to begin it!***

Part 1: Loading the Data File

Every game consists of a collection of locations, called *rooms*. Every room has four required pieces of information:

- A unique *room number* (this value is used internally to track the player's current position; it is hidden from the player)
- A room *title* (a word or short phrase that identifies the room, like "Kitchen")
- A room *description* (a string containing a text description of the room's appearance). A description may be specified using multiple lines of the data file.
- A list of *exits* (directions that the player may go to travel to a different room). A room may have exits in any combination of the four cardinal directions (north, south, east, and west).

A room may also have one or both of the following optional elements:

- A list of one or more *items* (objects that the player can pick up during the course of the game for use in solving puzzles)
- A *puzzle* that must be solved in order to progress in the game. Puzzles generally award points for being solved, and may reveal new locations in the game and/or produce new items that the player can use to solve other puzzles (for example, the player may need to solve Puzzle 1 to gain a new item that will let the player solve Puzzle 2 in another room). Like room descriptions, puzzles are specified using multiple lines within the data file. Each room may have up to one puzzle.

Each type of element (except for the room number) is preceded by a specific identifying header (like "title" or "exits"). For example, the data file may contain a line like:

```
exits null 2 null 3
```

indicating that the current room has exits to the south (leading to room 2) and to the west (leading to room 3). Exits are listed in order — north, south, east, and west — with "null" used as a placeholder to indicate that there is no exit in that direction.

Within the data file, the data for a room is always terminated by a line containing exactly 5 dashes ("-----").

For example, here is a complete room from a sample data file:

```
1
title Entry Portal
exits null 2 null 3
items screwdriver
description This is a white, featureless room about ten feet square.
-----
```

This room (location number 1) is called the "Entry Portal". It has two exits (to the south and to the west, leading to rooms 2 and 3 respectively), and contains one item (a screwdriver) that the player can pick up and use. Finally, it contains a short description of the room's appearance.

We will use a (global) dictionary to store the rooms of our game. The room's number will serve as the key; its value will be another dictionary containing all of the other information about the room.

Start by defining a Python function with the following header:

```
def loadRooms(filename):
```

Inside this function, create several temporary variables to hold information about the current room. You will need:

- A variable to hold the new room number
- A string to hold the new room title
- A dictionary to hold the set of room exits
- A list to hold any items that may be present in this room
- A dictionary to hold any puzzle-related data for this room
- A string to hold the new room's description text

Then open the data file whose name was passed in a the function argument. For each line in the file:

1. If the line matches or begins with "-----" (use the `startswith()` command), then the current room description is complete. Do the following:

a. Create a new dictionary and store the elements of the current room in it (use appropriate strings as keys, like "title" for the room title and "items" for the list of items). If your puzzle dictionary contains a key named 'result' and a value named 'solved' (more on this later), then add the puzzle dictionary to your room data dictionary as well.

   At the end of this step, you should have a dictionary that looks something like the following (the actual content will vary based on the data file you're currently processing):

```
roomData = { 'title': 'Throne room',
  'items': ['scepter', 'crown'],
  'exits': {'north': 5, 'east': 3, 'south': 15, 'west': 'null'},
  'description': 'This is, or was, the king's throne room. An
enormous golden chair, sparkling with gems, sits in its center.
The room is dusty and cobwebs are everywhere, suggesting that
this room has not been used or visited in some time.' }
```

b. Assign your room data dictionary to your room dictionary using the room number as its key.

c. Finally, set your room data dictionary and your temporary variables back to empty strings, lists, or dictionaries in preparation for the next set of room data.

2. Otherwise:

   a. Split up the contents of the current line.

   b. Call Python's `isnumeric()` function on the first element of the split line to determine if it is a potential room number (i.e., if it contains only digits). If `isnumeric()` returns `True`, convert the first element to an integer and assign it to your room number variable:

```
if pieces[0].isnumeric():
    roomNumber = int(pieces[0])
```

   c. If the first element of the split line is 'title', use `join()` to reassemble the remaining pieces of the line (positions 1 onward) and store the result in your room title variable.

   d. If the first element of the split line is 'exits', add the next four pieces of the split line to your exits dictionary under the keys (in order) 'north', 'south', 'east', and 'west'.

   e. If the first element of the split line is 'items', add the remaining pieces of the line to your items list. Note that every item in the data file will have a one-word name (e.g., 'hammer' but not 'circular saw'), so each "word" is a separate item.

   f. If the first element of the split line is 'description', use `join()` to reassemble the rest of the line and ***append*** it to your room description string. **DO NOT** simply assign it to that string, as the room description text may be broken up across multiple 'description' lines, and assignment will replace any earlier parts of the room description.

g. If the first element of the split string is 'puzzle', then we need to add several values to the puzzle dictionary for this room. The format for a "puzzle" statement is as follows:

   `puzzle` *required-item point-value new-item new-direction new-room-number*

   For example:

   ```
   puzzle hammer 15 null south 22
   ```

   or:

   ```
   puzzle phaser 25 tricorder null null
   ```

   or:

   ```
   puzzle key 10 ring west 13
   ```

   That is, solving the puzzle (using the required item or "magic word") may award points for the player's score, may add a new item to the room's item list (if solving the puzzle doesn't produce a new item, this field will be 'null'), and may reveal a new exit (if the puzzle doesn't change the room's exit situation, then the last two fields will each be 'null').

   i. Add the string 'solved' (all lowercase) to your puzzle dictionary with the second line element (e.g., pieces[1]) as its key.

   ii. Add the third element of the line (as an integer) to the puzzle dictionary with the key 'points'.

   iii. Add the fourth element of the line to the puzzle dictionary with the key 'item'.

   iv. Add the fifth element of the line to the puzzle dictionary with the key 'direction'.

   v. Add the sixth element of the line to the puzzle dictionary with the key 'newroom'.

h. If the first element of the split string is 'puzzle-text', then ***append*** the remainder of the line (as a joined string) to your puzzle dictionary with the key 'description' (use `setdefault()`) to test for the presence of 'description' first).

i. Finally, if the first element of the split string is 'puzzle-result', then ***append*** the remainder of the line (as a joined string) to your puzzle dictionary with the key 'result' (use `setdefault()` to test for the presence of 'result' first).

Part 2: Playing the Game

Now that we've loaded the game data from a text file, we need to build a parser that will let the player move around within the game. We will use two functions to accomplish this.

1. First, we want to display information about the player's current status (i.e., score and number of moves) and location. Start by defining a new function with the following header:

   ```
   def printRoom(number, moves, score)
   ```

   In this function:

   a. Print out the current room's title, the player's number of moves so far, and the player's current score (out of 100), all on a single line. For example:

      Transporter Room (15 move(s), score: 25/100)

   b. Print out the room's description.

   c. If the current room contains a puzzle and that puzzle has a 'description' key, print the puzzle's description text.

   d. Print a list of the room's exit directions (separated by spaces), preceded by the text "You can go:". You should only print exit directions whose values are not 'null'. If the room has no valid exit directions, print "There are no visible exits" instead. For example:

      You can go: north south

      or

      There are no visible exits

   e. If the room contains any items, print out the elements of that list (separated by spaces), preceded by "Items you can see:". For example:

      Items you can see: hammer screwdriver wrench

2. Next, we need a function to interpret the player's commands. A user command consists of one or two words, like a direction name or an instruction to use a particular item from the player's inventory. User commands can be entered in any case, but should be converted to lowercase when read in to simplify processing. Our interpreter function will use the global dictionary of room data, and should do the following when called:

   a. Initialize the game variables:

      i. A list representing the player's inventory (the items that the player is currently carrying). For simplicity, we will assume that the player's inventory is of unlimited size.

      ii. An integer representing the player's current score.

      iii. An integer representing the number of moves taken so far.

iv. An integer representing the number of the player's current room. For simplicity, our games will always begin in room 1.

v. (optional) A variable to keep track of whether the game has ended or not. The game will not end until the player types "exit" or "quit".

b. Enter a loop to play the game, by reading and responding to user commands. While the game is still in progress:

i. Print the description of the current room.

ii. Read in the player's next command. Use a single right arrow bracket ('>') followed by a single space as the command prompt. Be sure to convert the player's command to lowercase.

iii. Process the player's command. If the command is "exit" or "quit", end the main game loop. Otherwise, add 1 to the player's number of moves, and act on the command as follows:

1. If the command is "inventory", print out a list of the player's current inventory items with an appropriate message. If the player's inventory is empty, print a message to that effect.

   For example:

   > inventory
   You currently have: phaser tricorder

   or

   > inventory
   Your inventory is currently empty

2. If the command is a direction ('north', 'south', 'east', or 'west'), check the value of that key in the room's dictionary of exits. If the associated value is 'null', print a message like "You can't go that way". Otherwise, set the player's current room to the direction's value.

3. If the command is "take", check to see if the second word in the command is present in the current room's list of items. If it is, print "Taken", add the item to the player's inventory, and remove it from the room's list of items. If the item is not found, print an appropriate message (e.g., "You can't take the X").

   For example:

   > take hammer
   Taken

   or

> take phaser
You can't take the phaser

4. If the command is "use" or "offer", look at the second word of the command. If the second word (the item name) is not present in the player's inventory, print "You don't have one of those!". Otherwise, if the current room does not have a puzzle, print "Nothing happens". Otherwise, if the second word is not in the puzzle's list of keys, print "It doesn't work". Otherwise:

    a. Print the puzzle's result.

    b. Update the player's score with the puzzle's point value and print a message to that effect ("You have just earned X points.").

    c. If the puzzle has an item, add that item to the room's list of items.

    d. If the puzzle has a non-'null' direction, update the room's list of exits with that direction and the room's "new room" value (as an integer value).

    e. Delete the puzzle from the room's dictionary using the `del` command.

5. If the command is "say", proceed as you did for the "use" command above, but don't check to see if the "magic word" is in the player's inventory first.

6. Otherwise, print "You can't do that" to indicate that the command is not recognized. For example:

    > breakdance
    You can't do that.

c. When the game loop ends, print a message to end the game:

    Your adventure is over. Your final score is X out of 100 points, in Y move(s). Have a nice day!

c. Finally, add a few Python statements to read in the name of a data file from the user and call your functions to start the game.

(Partial) Sample Program Output

(user input is in **bold**; program output is in *italics*)

```
Enter the name of the game file to load: testmaze.txt

Entry Portal     (0 move(s), score: 0/100)

This is a white, featureless room about ten feet square.

You can go: south west
Items you can see: screwdriver
> south

South Room    (1 move(s), score: 0/100)

Here is another empty room with no remarkable characteristics.

You can go: north
> east
You can't go that way

South Room    (2 move(s), score: 0/100)

Here is another empty room with no remarkable characteristics.

You can go: north
> north

Entry Portal     (3 move(s), score: 0/100)

This is a white, featureless room about ten feet square.

You can go: south west
Items you can see: screwdriver
> take screwdriver
Taken.

Entry Portal     (4 move(s), score: 0/100)

This is a white, featureless room about ten feet square.

You can go: south west
> inventory
You currently have: screwdriver

Entry Portal     (5 move(s), score: 0/100)

This is a white, featureless room about ten feet square.
```

```
You can go: south west
> west

Sphinx Room  (6 move(s), score: 0/100)

There is a huge skylight overhead, far out of your reach or ability to climb to. One wall of this
room features a button panel that has seen better days.

The room is mostly filled by an enormous Egyptian sphinx. It turns to look at you as you enter
the room, and wails "Will no one fix this control panel for me?" as it gestures toward the button
panel. You observe that the lower left corner of the panel is missing a screw.

You can go: east
> sleep
You can't do that.

Sphinx Room  (7 move(s), score: 0/100)

There is a huge skylight overhead, far out of your reach or ability to climb to. One wall of this
room features a button panel that has seen better days.

The room is mostly filled by an enormous Egyptian sphinx. It turns to look at you as you enter
the room, and wails "Will no one fix this control panel for me?" as it gestures toward the button
panel. You observe that the lower left corner of the panel is missing a screw.

You can go: east
> use screwdriver

You deftly replace the missing screw in the corner of the button panel. When you are done, the
sphinx breaks into an enormous smile and then claps you on the back with an enormous paw
before tapping a button on the panel to open the skylight. The sphinx then soars up and away,
through the skylight. A new, previously-hidden doorway slides open to the west.
You have just earned 100 points.

Sphinx Room  (8 move(s), score: 100/100)

There is a huge skylight overhead, far out of your reach or ability to climb to. One wall of this
room features a button panel that has seen better days.

You can go: east west
> exit

Your adventure is over. Your final score is 100 out of 100 points, in 8 move(s). Have a nice day!
```

Grading Breakdown

This assignment is worth a total of 30 points, based on program performance as follows:

Point Value	Grading Criterion
5	Program correctly displays the room description, score, and number of moves
5	Program correctly processes movement/direction commands
2	Program correctly processes the 'inventory' command
3	Program correctly processes the 'take' command
5	Program correctly handles puzzles with the "use"/"say"/"offer" commands
5	Program correctly ends the game when requested, or when the player achieves 100 points
5	Program correctly handles unrecognized/misspelled commands