CPSC 231 – Fall 2017
Assignment 4: The Object of the Game

Jeffrey E. Boyd

November 2017

Due: *27 November 2017, 9:00AM*

In this assignment you will write a computer version of a game popularly known as *Rush Hour*, manufactured by *ThinkFun* (`http://www.thinkfun.com/play-online/rush-hour/` – you can play an implementation of the game on this site). *ThinkFun* has a video on YouTube that describes the game play (`https://www.youtube.com/watch?v=HI0rlp7tiZ0`).

In the process of implementing the game, you will improve your programming skills and learn to organize a complex program using object-oriented programming. Beyond objects, you may use any features of Python, including those not covered in the course (although going beyond the course should not be necessary).

Puzzles and Games as Software

A puzzle like *Rush Hour* has a state. Therefore, to implement *Rush Hour* in a Python program, you will need to represent the game state in your program. There are some obvious aspects to this – *Rush Hour* is played on a 6-by-6 grid that could be represented with nested Python lists. The puzzle also has cars. Cars have positions on the grid, a size, and an orientation. You will need to represent these in your program too.

The puzzle also has rules for changing states. Look at the YouTube video – it will show you the basics of the game. The cars can move up/down or left/right in their column or row. They cannot skip over other cars. They cannot change row or column. They cannot change orientation. The game is over (or the puzzle is solved) when the game state satisfies a certain condition. In the case of *Rush Hour*, the puzzle is solved when the target car (the one in the second row from the top, starting with row zero), covers the exit square, $(2, 5)$ in the game grid.

When you have a representation of the state, and implementation of the rules for transitions implemented, you have most of the puzzle complete. After that, you need only stitch them together in *game loop* – a loop that gets transitions as inputs, checks that they are legal, and updates the game state accordingly.

Instructions

Write a program to do the following.

1. Read in a puzzle description from a file. I have five sample puzzles for you. In the files there is one line per car, each car with a comma-separated list. The first car is always the target car (it also must be in row two). The values in the list are the car orientation (horizontal or vertical), the size, and the row,column coordinates of the upper-left portion of the car. You should call your program from the command line like this:

   ```
   python3 rushhour.py game0.txt
   ```

   where rushhour.py is the name of your program and game0.txt is one of the puzzle files that I provide (or your own if you want to create more).

2. Write a version of the program that is texted based (*rushhour.py*, i.e., there is a conventional command prompt for a move, and the game state gets printed with *print()* statements or similar. This will be worth the bulk of the marks for the assignment.

3. Write a final version of the program that has a graphical interface *rushhour_gui.py*. I recommend that you do this with pygame (`http://pygame.org/`). Pygame is covered in the course text book. It is complex, but you can use the subset of the pygame described in the text to make a good solution to this assignment.

To get full marks, you must make effective use of objects to organize your code.

Here are some suggestions to help you.

1. Spend some time studying the puzzle and try a few ways to represent the entities in the puzzle – like the car and the grid. Think about how you will implement the state transitions and check that the moves are legal.

2. Write a text-only version of the game. This has a couple of advantages. First, it allows you to get your game running without the added complexity of the pygame graphics. Second, this will also encourage you to separate your code in to entities (objects?) that are organized around function – you will separate the game functions from the display functions. The text-only version will be clumsy to play, but it is a smart step in development.

   I will not force you to structure you objects any particular way. I want you to learn by trying different ideas. When you first try this, you will make mistakes, but your organization of code will get better with practice. I will, however, suggest that you implement a car object and a game/puzzle object. To give you ideas, these are some of the methods in my version of the game object (you may be able to do better).

   - *__init__* – the constructor – but that is obvious.
   - *addCar* – to add a car to the game
   - *loadGame* – to load a puzzle from a file, might make use of *addCar*
   - *gameOver* – a test to help a game loop to know when the puzzle is solved
   - *printState* – pretty obvious – good for debugging and helps with the output for text-only
   - *moveCar* – obvious what this does, but it has to check the rules

   This is not a complete set of methods, nor is it necessarily the best way to solve the assignment. But it should give you a start.

3. The graphical game is best done as a separate object that is a layer on top of the game object. You can use inheritance if you like, but this is not necessary. My solution created a game object, loaded a puzzle, then passed the game instance to the constructor of my graphic interface object. The graphic interface object merely translates mouse clicks to moves of cars, and maintains a visual representation of the game state – it relies on the game object to guide it.

4. Only simple mouse clicks are necessary in the interface. You may choose to drag and drop or otherwise animate the game for bonus marks, but this is not a requirement.

5. I implemented the solution in two files. The first played the text-only game from the command line when run as *__main__*. The second file was the graphical version which imported the game object from the first file. My first file has 179 lines, and the second has 115 lines (that length is with only partial commenting) – so start early and do not underestimate the assignment.

6. Use the list in the *sys* module, *sys.argv*, to get a file name from the command line.

runs: text version runs w/o run-time errors (1)	yes no	
runs: graphical version runs w/o run-time errors (1)	yes no	
functionality: text version plays the game correctly (2)	excellent good poor	
functionality: graphic version plays the game correctly (2)	excellent good poor	
style: (4)	excellent good fair poor	
bonus - glitz: (3)	excellent good fair poor	
Total:		/10 + 3

Table 1: Grading rubric.

Hand In

1. Electronic copy of your programs using D2L, as per your TAs instructions. The TA will run your program sto test that they functions correctly.
2. A readme.txt file with any instructions the TA might need to run your solution correctly. Keep this *brief*.
3. A game.txt file in the format described above. It may be one of the ones that I have provided.
4. Any other files that you might need for bonus glitz.

Marking

Table 1 shows the grading rubric for the assignment.

To get a grade above 3/10, your text-only program must run without run-time errors.

Style refers to all of the following:

- use of comments and white space to enhance readability,
- use of descriptive names to enhance readability,
- avoidance of function/method side effects, and
- logical organization of code using classes, functions, and modules (use of objects required to get above 2/4).

Bonus glitz refers to any of the following (to challenge yourself further):

- enhanced appearance of the graphical game with textures and images,
- a feature to keep track of the move count in the graphical game, and
- animated movement of the cars with a mouse drag.

Late work

After the deadline and up to 24hrs late: -2.5pts. After 24hrs and up to 48hrs late: -5pts. Over 48hrs late: -10pts, i.e., no assignment will be accepted beyond 48hrs after the deadline.

Collaboration

Although it can be helpful to you to discuss you program with other people, and that is a reasonable thing to do and a good way to learn, the work you hand in must ultimately be *your own work*. This is essential for you to benefit from the learning experience, and for the instructors and TAs to grade you fairly. Handing in work that is not your original work, but is represented as such, is *plagiarism* and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code that you hand in that are not your original work. You can put the citation into comments in your program. For example, if you find and use code found on a web site, include a comment that says, for example,

   `# the following code is from http://stackexchange.com`.

   Use the complete URL so that the marker can check the source.

2. Citing sources will avoid accusations of plagiarism and penalties for academic misconduct. However, you may still get a low grade if your work is not the product of your own efforts.

3. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you find you are exchanging code by electronic means, writing code while sitting and discussing with a fellow student, typing what you see on another person's console, then you can be sure that your code is not substantially your own, and your sources must then be cited to avoid plagiarism.

4. We will be looking for plagiarism in your code, possibly using automated software designed for the task. For example, see *Measures of Software Similarity* (MOSS - `https://theory.stanford.edu/~aiken/moss/`).

Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help, than it is to plagiarize.