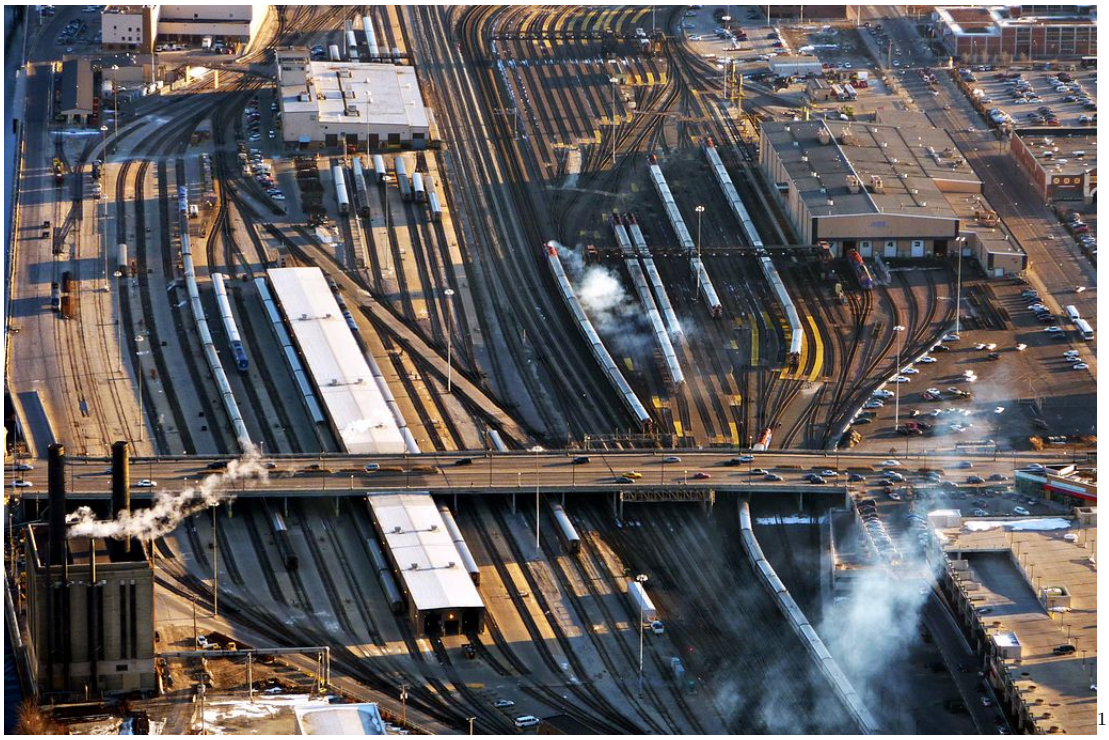
CS 120 (Spring 22): Introduction to Computer Programming II

Long Project #4 - Railyard

due at 5pm, Tue 15 Feb 2022

 [1]

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

1 Overview

In this project, you will be writing a small game. (Sorry, it's going to be text-only, so it's not going to be pretty.) You will read commands from the keyboard, and display the game state again after each command.

Conceptually, the game is fairly simple: move train cars around, from track to track, until a track contains cars only going to a single location; then the train departs.

You can see an online version of this program! Log on to to
`http://52.89.101.165/railyard_live/game.html`
and see how the game works.

[1] By Basil D Soufi - Own work, CC BY-SA 3.0,
`https://commons.wikimedia.org/w/index.php?curid=16408186`

2 Background: Railyards

A railyard is a series of tracks that are used for storing railcars and locomotives; often, they are used for "sorting" - that is, disassembling incoming trains, and building new ones. A very common form for a yard is "ladder track," where a single main line diverges into many parallel tracks:

In the real world, the tracks usually re-converge back to a single mainline at the other end; however, in our game, we will imagine that there is only one exit from the yard, headed to the right. Thus, the only way to move cars around is to cut a few off of a train, pull onto the main line, and then push the cars back onto another track.

In the real world, a single track sometimes contains multiple cuts of cars, which are not connected to each other. But in our game, any time you put cars on the same track, they are **always** connected together, no matter what.

Also, in the real world, engineers only leave when they're told. But in our game, the engineers are very impatient - they will leave **immediately,** if you ever place them on a track where all of the cars are going to the same destination.

[2]

[2]Source:
https://jlandtrailroad.blogspot.com/2018/11/tweaked-staging-yard-ladder.html

3 The Game

In this game, you will have a certain number of parallel tracks. We won't model the switches that connect them; we will only focus on the tracks themselves. Your goal is to sort the trains and send each car to its proper destination.

We will represent each car with a lowercase letter, which represents its destination; locomotives will be represented by a capital T. We will print out the state of the yard, after each move, using a format like this:

```
1: --------------------T-
2: ----------cbbbbaaaaT-
3: ----------aaabbbbbbT-
4: ----------------------
5: ----------------------
Locomotive count:  3
Destination count: 3
```

In this example, Track 1 has only a locomotive. Track 2 has a locomotive, followed by 4 cars headed for destination `'a'`, 4 for `'b'`, and 1 for `'c'`.

3.1 Yard Rules

In this game, we make certain simplifying assumptions, which are not true in a real railyard, but which make the game interesting:

- No track can ever have more than one locomotive, and the locomotive, if any, must always be at the "head" of the list of cars (that is, the far right).

  Of course, empty tracks are OK. And it's also OK to have a track with a loco but no cars. It is also OK to have cars but no loco.

- Any cars that are in any track are always connected together, and shoved as far right as possible - even when we cut off some cars, and leave others behind.

(spec continues on the next page)

3.2 Moves

The only possible move in this game is to "cut" out a certain number of cars (using the locomotive on that track) and move these cars to another track. (This is only a legal move if the "from" track has a locomotive, and the "to" track does not, and also that the "to" track has enough free space for both the cars, and also the locomotive.)

For example, given the example above, suppose that we moved 6 cars from Track 2 to Track 5. The new state is below; you will notice here that "moving 6 cars" meant moving 6 cars plus the locomotive, so 7 units overall. (Also, note what the program prints out, when you perform this move.)

```
The locomotive on track 2 moved 6 cars to track 5.

1: --------------------T-
2: ------------------cbb-
3: ----------aaabbbbbbT-
4: ----------------------
5: --------------bbaaaaT-
Locomotive count:  3
Destination count: 3
```

(spec continues on the next page)

3.3 Departing

After we have moved a cut of cars to a new track, it's possible that the engineer (driving the locomotive) may depart. The player **cannot control** when the engineers depart the railyard - instead, they will **automatically depart** as soon as both of these are true:

- There is a locomotive on the track, and at least one car
- All of the cars on the track are headed to the same destination

So, following on to the previous example, let's suppose that we moved 4 cars from Track 5 to Track 4. This will create a new train, with 4 cars, all headed to `'a'`, and the engineer will depart immediately. To show this, draw the yard after the move, then tell the user that the train is departing, and then draw the yard again. Do **not** ask the user for any command until the train has departed.

Notice, in the example output below, that the cars left behind on Track 5 do **not** depart - because they are not attached to a locomotive.

```
The locomotive on track 5 moved 4 cars to track 4.

1: --------------------T-
2: ------------------cbb-
3: ----------aaabbbbbbT-
4: ----------------aaaaT-
5: --------------------bb-
*** ALERT***  The train on track 4, which had 4 cars, departs for destination a.

1: --------------------T-
2: ------------------cbb-
3: ----------aaabbbbbbT-
4: ----------------------
5: --------------------bb-
Locomotive count:  2
Destination count: 3
```

(spec continues on the next page)

3.4 Empty Moves

If, at any point, we ask the game to move **zero** cars, then that means you are moving only the locomotive, with nothing attached. This is perfectly valid; you can split off the locomotive from a bunch of cars, to place it alone in a track (no, it will not depart in that case). You can also move a locomotive to go meet up with an existing train (so long as it doesn't have a locomotive already). You can even move the locomotive to a different, empty track - although that's kind of pointless.

Again, following our previous example, we now move the locomotive from Track 1 to Track 2, and then use it to move 2 cars from Track 2 to Track 5. It then departs:

```
What is your next command?
move 0 1 2

The locomotive on track 1 moved 0 cars to track 2.

1: ----------------------
2: -----------------cbbT-
3: -----------aaabbbbbbT-
4: ----------------------
5: -------------------bb-
Locomotive count:  2
Destination count: 3

What is your next command?
move 2 2 5

The locomotive on track 2 moved 2 cars to track 5.

1: ----------------------
2: --------------------c-
3: -----------aaabbbbbbT-
4: ----------------------
5: ----------------bbbbT-
*** ALERT***  The train on track 5, which had 4 cars, departs for destination b.

1: ----------------------
2: --------------------c-
3: -----------aaabbbbbbT-
4: ----------------------
5: ----------------------
Locomotive count:  1
Destination count: 3

What is your next command?
```

4 Your Program

You will write a program named `railyard.py` . It will first ask for the name of the "yard file" it should read - this is a file which shows how the yard is arranged when the game begins, and also what tracks exist, and what their lengths are. (We'll detail that format later.)

From then on, it will loop, asking the user for a command (as you saw in the previous example). Each time, you will print the yard state first; then print the prompt, on one line. Then, use `input()` to read the command. Split the command to break it into words and to clean up any whitespace. Perform the command, and then display the yard state again.

The program should end when the last locomotive departs; note that there might be cars left behind. Also, terminate your program (with the message `Quitting!`) if the user types the `quit` command.

4.1 Commands

Your program must support the following commands:

- `move <count> <from> <to>`

  Moves `<count>` many cars (could be as few as zero) from Track `<from>` to Track `<to>`

- `quit`

  Terminates the program

(spce continues on the next page)

5 Yard File Format

The yard file looks like a simplified version of the yard display that you print out for the user. Here is a very simple example (see `yard1.txt` in the project zipfile):

```
--------------------T-
-----------cbbbbaaaaT-
-----------aaabbbbbbT-
----------------------
----------------------
```

(You may assume that the file doesn't include any blank lines, or anything else other than the tracks.)

You will notice that the rightmost column is always empty; so is the leftmost. If you count, you will notice that each of these lines has exactly 22 characters; since the first and last character on each line must always be empty, this means that each of these 5 tracks has a capacity of 20 cars. If you implement the check for too-long trains (see the next section), then any trains that you move around on this map must be limited to 20 cars or locomotives.

Different yards can have different numbers of tracks; you must read the file to figure out how many tracks there are, and what their contents are.

5.1 Strange Yards

You can pass most of the testcases while supporting only very simple railyards: you can assume that all of the tracks are 20 segments long (22 if you count the ends).

However, to pass **all** of the testcases, you need to be more flexible. Some of the testcases use railyards which are different than 20 segments long, such as `yard3.txt`:

```
--------------------ddaccd-
-------------------cbcbbaT-
----------------------aeeeT-
-----------------------cddT-
------------------------deT-
------------------------ceT-
```

In this yard, all of the tracks are the same length, but they have space for 25 cars instead of 20. (You know this because each line in the file is 27 characters.)

But even more advanced is `yard7.txt`, where different lines have different lengths:

```
--aabcT-
-----abbT-
------------
--------ccbbT-
```

(spce continues on the next page)

6 Error Checking

While you can pass most of the testcases without a lot of error checking, to pass **all** of the testcases, you should check for all of the following error conditions. The testcases I've provided included examples of each of these, along with the proper error message for each.

Check for each of the following:

- The yard filename doesn't exist
- The command name is not valid
- The user didn't provide the right number of arguments for the `move` or `quit` command
- One of the parmeters to `move` was not an integer
- The to-track and from-track are the same
- The user asks to move a negative number of cars
- The from-track is empty, or doesn't have a locomotive
- The to-track already has a locomotive
- The from-track doesn't have enough cars to satisfy the required move
- The to-track doesn't have enough space to hold the moved cars (including the locomotive)

7 Turning in Your Solution

You must turn in your code using GradeScope.