ICS 32 Winter 2020
Project #5: *The Fall of the World's Own Optimist (Part 2)*

Due date and time: *Friday, March 13, 11:59pm*

This project is to be done individually

Background

In the previous project, you implemented a Columns game with a fairly primitive user interface that ran within the Python shell, in the spirit of the user interfaces we've been building all quarter. This didn't make for a particularly fun game, but it did let us focus our energies on the precise rules of our game, so we could get all of the various mechanics right, before turning our attention to making a playable game out of it. In this project, we'll switch over to making the game playable.

In recent lectures, we've been learning more about **PyGame**, a third-party library — not one built into Python, but nonetheless one that is easily downloaded and installed for your use. This project asks you to take the program you wrote for the previous project and replace its user interface with a graphical one. You'll build your game using the **PyGame** library, as we've been doing in lecture. Rather than simulating the passage of time by pressing the **Enter** key, the game will move on its own; this will add the key elements of challenge and fun to it.

The program

This project asks you to build a graphical, playable version of Columns, based on the game mechanics from the previous project. However, we'll need to make a couple of minor changes to it:

- The field will always consist of 13 rows and 6 columns, and will always start out empty.
- The seven colors of jewels were represented as uppercase letters previously. Instead, choose seven colors — you can choose any colors you'd like, though they should be different enough that you can easily tell the difference between them at a glance. (You might want your game mechanics to still refer to them using uppercase letters, and that's fine; but when you display them, what differentiates the jewels should be their colors.)
- There needs to be some kind of visual cue when fallers land. Additionally, if you support matching, you would ideally want a visual cue when jewels match before they disappear, as well. You have your choice about these visual cues — you can use colors or other visual effects.
- Rather than the user adding fallers manually, they appear randomly. Whenever there isn't a faller in the field, a new one appears in a randomly-chosen column and consisting of random colors. (When choosing a column randomly, never choose a column that is already filled with frozen jewels, unless all columns are already filled with jewels.)
- Rather than the user pressing the **Enter** key to simulate the passage of time, you'll instead "tick" the game mechanics once per second automatically.
- Rather than the user typing commands like **R**, **<**, and **>** in the Python shell to rotate and move the faller, the user instead should move them by pressing keys on the keyboard; every keypress rotates or moves the faller once. So that we'll know how to grade your project, we'll all use the same keys: *left arrow* and

right arrow should move the faller to the left and right, respectively, while the spacebar should rotate the faller.

Which rules must be implemented?

The only game mechanics that are absolutely necessary — for full credit on this project — are the mechanics of creating fallers, moving and rotating them, and having them land and freeze. All of the other mechanics — matching, particularly — are optional. If you already completed Project #4, you'll want to include them here. But we do want those of you who have only partial implementations (within bounds of reason) to be able to proceed with this project without penalty.

What should it look like?

You have a fair amount of leeway about how the game looks. The basic feel should be similar to the 1990 Sega version, in the sense that you would display the jewels in a grid, and you would see them move one grid cell at a time. Jewels should be displayed with different colors, and you can opt to use different shapes, though that is not required.

Additional challenges

You are certainly welcome to add new features to your game, though they are not required (and extra credit is not offered in this course). Some ideas of what you might add include:

- Displaying the contents of the "next" faller that will appear, before it appears.
- A mechanism for keeping score and displaying it.
- Representing the jewels using small images, instead of drawing them programmatically. (There are limitations on the size of those images; see below.)
- Adding sound effects and/or music. (There are limitations on the size of the files needed for this, as well.)

Approaching the problem

Reusing your game mechanics

If you have a complete set of game mechanics from the previous project, you'll find that your task this time is primarily one of visualization and taking input. Your game mechanics, if designed appropriately before, should be largely usable without modification. In some ways, it may not (e.g., if your game mechanics made assumptions about the user interface running in the Python shell, or about the specifics of the input and output like **EMPTY** or **F 1 X Y Z**); in the areas where your game mechanics aren't as good of a fit lie learning opportunities. Reflect on what changes you had to make and why, and understand how you could have avoided making those changes.

Note, too, that you are eligible for up to full credit on this project even if you don't have absolutely perfect game mechanics, though there are some requirements that do need to be met, as described above.

What to do if you didn't complete your game mechanics previously

If you did not complete your game mechanics previously, you will need to complete the most basic requirements now — see above — and I would suggest focusing your energy on just what's necessary, since it will be somewhat easier to implement than the full rules. Since implementing only the necessary rules leaves open the possibility of receiving full credit on this project (i.e., there are no deductions for not implementing more than this, and there are no bonuses for implementing everything), there's no benefit — from a grade perspective — in implementing the full rules (though, of course, you're welcome to do so, if you'd like).

It should go without saying — but prior experience has shown otherwise — that you are required to submit *your own* game mechanics in this project, not someone else's. It's certainly true that the focus of this project is on building the graphical portion of the game, but one of the learning objectives here is becoming accustomed to reusing your own code to satisfy new requirements, so you can determine which parts of your own design worked well in the new context, which ones didn't, and reflect on why.

Module design

As before, you are required to keep the code that implements the game mechanics entirely separate from the code that implements the graphics and input handling for your game. To that end, *you will be required* to submit at least two modules: one that implements your game mechanics and another that implements the graphical portion. You're welcome to break these two modules up further if you find it beneficial, but the requirement is to keep these two parts of your program — the logic and the user interface — separate.

At least one of your modules should be executable (i.e., should contain an **if __name__ == '__main__':** block), namely the one that you would execute if you wanted to launch your game.

Using classes to implement your game

Because your event handlers will need access to the game's current state, the graphical portion of your game should be implemented as one or more classes, in the style we've been discussing in recent lectures, with (at least) an object representing the game itself; you may also find it useful to have additional kinds of objects that represent parts of your graphical implementation, but this is up to you to decide and is not a requirement.

Drawing the state of the game and resizing the window

You'll need to draw the game board using **PyGame**, which you can do programmatically using something like **pygame.draw** or by using small images and "blitting" them on to the display instead. Additionally, your game window must support resizing, and resizing it must cause the area in which the board is drawn to change size correspondingly, with the game's graphics redrawn to fill the available space in some reasonable way, so the user can decide how large of a window he or she would like to play the game on.

How we will be running your program

Because we will be using the third-party library **PyGame**, it is important that we all agree on how the various files and configuration will be done. This is what we're going to do when we run your program:

- First, we'll create a virtual environment, using the techniques shown in the Third-Party Libraries notes, then install the latest version of **PyGame** into it.
- Next, we'll create a **src** directory within that virtual environment, then take all of your source code (i.e., all of your **.py** files) plus any additional files needed for your game (images, sounds, etc.) and put them into that **src** directory. (Your game will need to be implemented to assume that all of these files are in the same directory.)

You'll want to be sure, before you submit your work, that this technique for setting up your project will work properly.

Where to find more information about PyGame

When you want to know more about the PyGame library, a good place to start is the Notes and Examples on the course web site, where I've tried to outline the big-picture concepts and demonstrate a few examples. Most

likely, though, there will be things that you run into that don't happen to covered in those notes, so you'll also want to be able to navigate PyGame's online documentation, which is actually extensive and quite good. You can find that documentation at the link below.

- PyGame documentation

The documentation is organized into sections for each part of the library; you'll see links to those sections listed near the top of the page. For example, if you're unsure about some part of the **pygame.display** library, you'll find a link titled **display**. I generally find myself using those navigation links near the top when I want to jump around and look at the details of things I don't remember or haven't seen yet.

There are also some tutorials available, though you'll find that they can take you into a lot of places where we won't need to go for our work.

Limitations

The only third-party library you are permitted to use is **PyGame** — which, in fact, you're required to use. Otherwise, you're free to use modules in Python's standard library to the extent that they're helpful to you.

There is a hard limit of 256KB on any additional files — images, sounds, and so on — that you want to submit above and beyond the Python source code for your game. This will allow you to spruce up your game to some extent, but disallows something like a large MP3 file for background music, a video that plays when the game start, and things of that nature. (This limitation is mainly motivated by the fact that this is a very large course, yet we're all sharing the same submission system and working to the same deadline, so we can't have hundreds of megabytes of data being submitted to Checkmate.)

Deliverables

Put your name and student ID in a comment at the top of each of your **.py** files, then submit all of the files to Checkmate. Take a moment to be sure that you've submitted all of your files.

If you have additional files — images or sounds, for example — that are also part of your game, you'll need to submit these, as well; there will be an additional area within the Project 5 dropbox on Checkmate to submit them. (Note that we will be assuming that we can copy these files into the same directory as your source code in order to run the game; see above.) We aren't kidding when we say that there is a limit of 256KB per student for "assets" like these; unfortunately, we're not going to be able to allow massive sets of files to be submitted, since there is a very large number of students enrolled in this course.

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link.

Modifications and clarifications by Alex Thornton, Winter 2018.
Originally written by Alex Thornton, Fall 2017.