ICS 32 Winter 2020
Project #4: *The Fall of the World's Own Optimist (Part 1)*

Due date and time: *Wednesday, March 4, 11:59pm*

This project is to be done individually

Background

My first exposure to computers, as a kid in school, was in the context of computer games; some were educational games (it was school, after all), though many were not. The first time I remember sitting behind a computer — a Radio Shack TRS-80 Model I — I played a game called FASTMATH, which pitted two players against one another, trying to alternately solve arithemtic problems and type in the answers as quickly as possible. Sure, it was just a boring educational game, one that was ridiculously simple by today's standards, but at the time I was captivated, and I still remember it to this day. (I especially loved winning, though I didn't always win.)

Thanks to the wisdom and generosity of my parents, it wasn't long before I had my own computer at home (a Commodore 64), complete with its own collection of games. None of the games I played on my own computer could be classified as educational in a direct sense, though those games were sneaky: They taught me a surprising collection of lessons and motivated me to ask many interesting questions about computing, as I endeavored first to win them outright, then to modify them (to cheat or to change how a game was played to make it more fun), and finally to write them from scratch. Games in those days, of course, didn't have the same photorealistic, three-dimensional, surround-sound appeal that they have today, but they were nonetheless fun and exciting; their simplicity made writing one's own game seem more possible with limited skills than it does today, in an era of tremendously complex games built by gigantic teams of programmers, designers, and artists. (In truth, it's easier to build simple games now than it was then, because the computers have become so much more powerful and the tools have gotten better. It's just harder to compete with the large-scale, professionally-developed games.) Unfortunately, my skills didn't develop quickly enough — I always aimed too high, relative to what I knew how to do, but it was tougher when there was no Internet to search when you got stuck on something — and I never realized the goal of writing my own games before I became interested in other things, though I certainly learned a lot trying.

This project is the first of a two-part sequence that offers you to opportunity to build your own game. The first of the two projects focuses on developing a clean set of game logic and a test apparatus that runs in the Python shell to let you (and us) verify that it works as expected. The second one pivots into drawing graphics, generating sounds, and other techniques to turn it into a playable game. Along the way, we'll focus on finding a design that serves both purposes, on finding ways to simplify our code by eliminating duplication of boilerplate, and continuing our journey into understanding the mechanics and the benefits of classes and object-oriented programming in Python. Finally, we'll take a step out of Python's standard library and into the world of third-party, open-source libraries that the Python community has to offer.

Games may seem frivolous to some of you — I know that not everyone likes to play them — but they provide a fascinating combination of problems to be solved: software engineering, human-computer interface, computer networks, psychology and cognition, and even (in multiplayer online games) economics and sociology. Game developers push the envelope — in some cases further than just about any other kind of software developers — and many of these lessons can be applied in more seemingly serious contexts. Even if you're not that interested in games, you'll be surprised what building games can teach you about software.

The game of Columns

In the late 1980s, a puzzle-based game called Columns showed up on at least one personal computing platform. Later, it became somewhat more well-known when it was ported to the Sega Genesis, and later to other video game consoles and personal computers, becoming one of the better-known puzzle games of the early 1990s. In this project and the one that

follows it, you'll be building a version of this game. It's not going to be our goal to completely duplicate the original, but what we'll be building will maintain its spirit throughout.

The best way to start getting your mind around the game is actually to watch someone play it; the game is simple in concept, and you'll probably be able to deduce most of the rules just from watching a little bit of it, so best to start there.

- Columns gameplay video (Sega version from around 1990)

Having watched the video, you'll have likely figured out many of the core rules of the game already, though I should point them out, not only to be sure that we all agree on what they are, but so that we can agree also on a set of terminology that we'll use to describe how the game is played. (A common vocabulary goes a long way in letting people talk about abstract concepts without confusion.)

- The *field* is a grid of cells, each of which is either empty space or contains a colored jewel. At the outset, the field is empty, but it begins filling up once the game begins.
- One at a time, a *faller* is created and begins descending from the top of the field downward. Each faller is made up of three jewels arranged vertically, with the three jewels colored randomly.
- As the faller descends, it can be *rotated*, which is to say that the bottom jewel becomes the top one, while the other two move down to make room for it. Because the colors of the jewels are relevant, the act of rotating them serves a vitally important role in the mechanics of the game.
- Fallers are subject to a crude sort of gravity, so they descend through the field until they *land* either on the bottom of the field or on top of a jewel that is already present in the field.
- Shortly after a faller has landed, it *freezes*, which is to say that the faller has nowhere left to fall, but that the jewels contained within it will remain in the field, wherever they landed.
- After each faller freezes, if there are three or more jewels of the same color in a row — either horizontally, vertically, or diagonally — they are said to *match*, in which case they are displayed specially and then disappear. After they disappear, any jewels above them moving down immediately to fill the empty space.

The basic objective of the game is to continue as long as possible, with the game ending as the field becomes to filled to continue. There are rules around scoring, of course — because a game like this is a lot less fun if there's no way to measure one's performance — but we'll skip them for the time being. For now, our goal is to implement the game's basic mechanics of fallers, movement and rotation, freezing, and the matching and disappearance of adjacent sequences of jewels of the same color.

The program

This project is focused on implementing the mechanics of Columns, along with a program that you can run in the Python shell with a very spartan user interface that you'll use to test it — and that we'll use to *automatically* test it, making it crucial that you get the format of the program's input and output right according to the specification below. Spacing, capitalization, and other seemingly-minor details are critical; every part of your output needs to be correct to the individual character.

Using your test program, you won't be able to play a fully-functioning game of Columns, but you will be able to determine whether you've handled all of the game's mechanics correctly: fallers, movement and rotation, freezing, matching, and so on. Note, too, that you don't need to implement all of the logic to receive at least some credit for the project; details are described a little later in the project, but you can receive substantial partial credit for a partially complete implementation, as long as some of the features are working with precisely correct output.

The next project will allow you to take the game mechanics you've implemented here and build a playable game out of them. But, as with a lot of game implementation, first thing's first: Without the underlying logic working, you can't have a game you can play.

A detailed look at how your program should behave

Your program will read its input via the Python shell (i.e., using the built-in **input()** function), *printing no prompts to a user* with no extraneous output other than precisely what is specified below. The intent here is not to write a user-friendly user interface; what you're actually doing is building a tool for testing your game mechanics, which we'll then be using to *automatically* test them. So it is vital that your program reads inputs and write outputs precisely as specified below. You can freely assume that

the input will match the specification described; we will not be testing your program on any inputs that don't match the specification.

- First, your program needs to know the size of the field. It will always be a rectangle, but the number of rows and columns can vary.
  - First, your program reads a line of input specifying the number of rows in the field. You can assume this will be no less than 4.
  - Next, your program reads a line of input specifying the number of columns in the field. You can assume this will be no less than 3.
- At any given time, the field contains jewels that are one of seven colors. The colors are represented by these uppercase letters (and only these letters): **S**, **T**, **V**, **W**, **X**, **Y**, **Z**. In both the input and output, we'll use these seven letters to denote the seven colors.
- Now, your program needs to know what jewels are in the field to begin with. There are two situations: We might want to start with an empty field, or we might want to specify the contents of the field in the input.
  - If the field is to begin empty, the word **EMPTY** will appear alone on the next line of input.
  - If instead we want to specify the contents of the field in the input, the word **CONTENTS** will appear alone on the next line of input. Given that there are *r* rows and *c* columns in the field, there would then be *r* lines of input, each of which will contain exactly *c* characters; these characters represent the contents of each of the field's cells to start with.
    - For a cell that should contain a jewel of some color, an uppercase letter describing each color will be used.
    - For a cell that should be empty, a space will be used instead.
    - Note that when we're specifying the contents of the field explictly, the spaces will always be present in the input for every cell that's empty; the program should expect to read exactly the correct number of characters.
- At this point, the game is ready to begin. From here, we will repeatedly do two things: Display the field, then read a command from the user.
- The rules for displaying the field are:
  - Given that the field has *r* rows, the field will be displayed as a total of *r* + 1 lines of output. The first *r* will correspond to the *r* rows of the field, which each row displayed like this:
    - The vertical bar character **'|'**, followed by three characters for each of the *c* columns in that row, followed by another vertical bar character **'|'**. For each column in that row, the three characters will be:
      - Three spaces if the cell is empty
      - A space, followed by an uppercase letter if the cell contains a jewel that has been frozen.
      - A left bracket character **'['**, followed by an uppercase letter, followed by a right bracket character **']'** if the cell contains a jewel that is part of the faller (if any).
      - A vertical bar character **'|'**, followed by an uppercase letter, followed by another vertical bar character **'|'** if the cell contains a jewel that is part of a faller that has landed but not yet frozen.
      - An asterisk character **'*'**, followed by an uppercase letter, followed by another asterisk character **'*'** if the cell contains a jewel that has frozen and has been recognized as a match.
    - After the last row of the field, a space, followed by 3*c* dashes, followed by another space is displayed.
- The commands that you would read are:
  - A blank line, which is a crude representation of the passage of time. (In our complete game, this would happen without any input; instead, when a certain amount of time passes, we would see the appropriate effect.)
    - If there is a faller present, it falls; if there is a faller that has landed (and has not been moved so that it is no longer in a landed position), it freezes; and so on.
  - **F**, followed by an integer that is a column number (the columns are numbered 1 through *c*, if there are *c* columns), followed by a space, followed by three uppercase letters (representing colors), each of these things separated by spaces (e.g., **F 1 S T V**). This means to create a faller in column 1, with a jewel of color **S** on the top, a jewel of color **T** below it, and a jewel of color **V** below that.
    - The faller begins with only the bottommost of the three jewels visible. See the example outputs below for more details.
    - Note that there can only be one faller at a time, so this command has no effect if there is a faller that has not already been frozen.
  - **R** alone on a line, which rotates the faller, if there is one. If there is no faller currently, this command has no effect. Note, though, that it is possible to rotate a faller that has landed but not yet frozen.
  - **<** alone on a line, which moves the faller one column to the left, if there is one (and if it not blocked by jewels already frozen on the field or by the edge of the field). If there is no faller or the faller can't be moved to the left,

this command has no effect. Note, though, that it is possible to move a faller that has landed but not yet frozen, which can take it out of its "landed" status (if it moves to a column with nothing underneath it).
    - **>** alone on a line, which moves the faller one column to the right, if there is one (and if it not blocked by jewels already frozen on the field or by the edge of the field). If there is no faller or the faller can't be moved to the right, this command has no effect. Note, though, that it is possible to move a faller that has landed but not yet frozen, which can take it out of its "landed" status (if it moves to a column with nothing underneath it).
    - **Q** alone on a line, which means that to quit the program.
- There are three ways for the program to end:
    - If the user specifies the command **Q**, the program ends, with no additional output being printed.
    - When a faller freezes without all three of its jewels being visible in the field (for example, if it lands on a jewel that's two rows below the top and then freezes), the game ends, so the program ends, as well. In that case, you would print **GAME OVER** before ending the program. Note that matching can keep the game from ending in this case (i.e., if the freezing faller triggers matching that causes it to fit after jewels disappear, the game doesn't end).
    - When a faller is created in a column that is already filled with frozen jewels. (One consequence of this rule is that the game doesn't end if the board is completely filled with jewels; it's the subsequent creation of a faller that ends the game, in that case.)

Two complete examples of program execution

Below are two examples of the program's execution, which you can consult when there are minor issues that you're not sure how to resolve. Boldfaced, italicized text indicates input, while normal text indicates output. Note that blank lines are actually blank input lines; there are no blank output lines in this program's design. Some additional commentary appears to the right of the example, italicized and written in the form of Python comments; that would not appear in your program's output, but will help you to understand the examples.

Beginning with an empty field

```
4
3
EMPTY           # begin with an empty field with 4 rows and 3 columns
|          |
|          |
|          |
|          |
 ---------
F 3 X Y Z       # create a faller containing colors X, Y, Z in column 3
|       [Z]|
|          |
|          |
|          |
 ---------
                # this is a blank line of input; there are lots of them in these examples
|       [Y]|
|       [Z]|
|          |
|          |     # the faller begins falling
 ---------

|       [X]|
|       [Y]|
|       [Z]|
|          |
 ---------

|          |
|       |X||
|       |Y||     # the faller has landed
```

```
|       |Z||      # note the multi-step process here
 ---------

|           |
|        X  |
|        Y  |
|        Z  |      # the faller has frozen
 ---------
F 1 Y Z X
|[X]        |
|        X  |
|        Y  |
|        Z  |
 ---------

|[Z]        |
|[X]     X  |
|        Y  |
|        Z  |
 ---------

|[Y]        |
|[Z]     X  |
|[X]     Y  |
|        Z  |
 ---------
R                 # rotating the faller
|[X]        |
|[Y]     X  |
|[Z]     Y  |
|        Z  |
 ---------
>                 # moving the faller to the right
|    [X]    |
|    [Y] X  |
|    [Z] Y  |
|        Z  |
 ---------
>
|    [X]    |
|    [Y] X  |
|    [Z] Y  |
|        Z  |      # moving to the right again has no effect; it's blocked
 ---------

|           |
|    |X| X  |
|    |Y| Y  |
|    |Z| Z  |
 ---------

|           |
|     X  X  |
|     Y  Y  |
|     Z  Z  |
 ---------
F 1 T Z S
```

```
|[S]      |
|    X  X |
|    Y  Y |
|    Z  Z |
 ---------

|[Z]      |
|[S] X  X |
|    Y  Y |
|    Z  Z |
 ---------

|[T]      |
|[Z] X  X |
|[S] Y  Y |
|    Z  Z |
 ---------

|         |
||T| X  X |
||Z| Y  Y |
||S| Z  Z |
 ---------
R               rotation and moving is permitted before freezing
|         |
||S| X  X |
||T| Y  Y |
||Z| Z  Z |
 ---------

|         |
| S  X  X |
| T  Y  Y |
|*Z**Z**Z*|     # we have a match!
 ---------

|         |
|         |
| S  X  X |
| T  Y  Y |     # and now the matching jewels disappear
 ---------
F 1 V W Z
|[Z]      |
|         |
| S  X  X |
| T  Y  Y |
 ---------

||W|      |
||Z|      |
| S  X  X |
| T  Y  Y |
 ---------

| W       |
| Z       |
| S  X  X |
```

```
| T  Y  Y |     # landed, but it doesn't fit!
 ---------
GAME OVER
```

Beginning with the field contents specified

```
4
4
CONTENTS
 Y X
S V
TXYS
X XY
|            |
| S     V  X |
| T  Y  Y  S |  # all jewels immediately fill empty space below them
|*X**X**X* Y |  # which can trigger matching, as happens here
 ------------

|            |
|          X |
| S     V  S |
| T *Y**Y**Y*|
 ------------

|            |
|            |
| S        X |
| T     V  S |
 ------------
F 2 X Y Z
|   [Z]      |
|            |
| S        X |
| T     V  S |
 ------------
Q              # quitting early is allowed; program ends
```

Working incrementally (and a reward for doing so)

This problem is larger and more complex than you may have worked on before, so it will be important that you're able to control that complexity in your mind by working on the problem incrementally. You won't be able to solve the entire problem all at once — and it's possible that some of you won't end up solving the entire problem at all — yet you do want to be able to get some positive feedback about your progress along the way (and partial credit on the project if you don't finish it), so we're breaking down the grading of this project into a set of features, which may help you to organize your thoughts the same way. While we aren't (and won't be) explicitly listing a point value on each feature here, this will give you an idea of the way we'll be organizing the grading process.

Your best bet, overall, is not to implement the entire program at once, haphazardly. Implement it one complete feature at a time. (This is good advice even if this wasn't being graded; working incrementally is the only way to build programs too large to work through in one sitting.) There isn't a single ordering of features that is the "right" one, though, of course, some depend on others. So what you'll want to do is choose one feature to start with, implement just that feature, test it, and then you'll be ready to move on to the next one. You might also want to keep a copy of your code after you complete each feature, so you can "roll back" to the previous copy if you find that the next step you take leads you in a direction you're not happy with.

The features

This program can be considered to require the following set of features. We will be grading these relatively independently of one another, except in the sense that some of them depend on others. For example, if fallers don't move down as time passes, then you won't ever be able to get them to the point where they've landed and frozen.

These aren't listed in any particular order — and are not listed in anything like the order in which I implemented them in my own solution — so don't feel like you need to work through these from top to bottom. But this will serve as a checklist to tell you which issues you have left to solve.

- The game can begin with an empty field of the correct size.
- The game can begin with the contents of the field specified, with jewels in some of the cells.
- When there are "holes" in the contents of the field specified at the beginning, the jewels "fall" immediately to fill the empty spaces below them.
- It is possible to quit the program with the **Q** command. (I did this one fairly early; you might want to do the same.)
- A faller can be created in a column and appear with only the bottommost of its three jewels visible.
- Fallers can be moved to the left and to the right at any point until they have frozen.
- Fallers cannot be moved to the left or right if they are blocked by jewels that are previously frozen.
- Fallers can be rotated at any point until they have frozen.
- Fallers land when they can't be moved down any further.
- Fallers freeze at the next "tick" of time after they have landed.
- The freezing of a faller can be postponed by moving it to a column with empty space underneath it after landing.
- The game ends when a faller freezes but cannot be displayed entirely in the field because it didn't move down far enough.
- Matching can be performed horizontally (i.e., three jewels of the same color horizontally would be considered to match).
- Matching can be performed vertically.
- Matching can be performed diagonally.
- Matched sequences longer than three jewels are handled.
- More than one matching sequence at the same time can be handled.
- The ending of a game can be postponed if a faller freezes without fitting in the field, but which matches enough jewels that everything then fits.
- When there are three or more jewels in a row in the contents of the field specified at the beginning, matching is triggered.
- When there are "holes" in the contents of the field specified at the beginning and jewels fill the spaces immediately, matching is triggered if three or more jewels in a row are present.

Note that substantial partial credit is available for submissions that implement some but not all of these features. In particular, if you can get everything working except the more esoteric matching scenarios, or even if you can get everything working except matching, you can still receive a fair amount of the 20-point correctness score (plus potentially all of the 10-point quality score).

However, it is vitally important that the structure of your output is correct, because the grading here will be done in an automated fashion. Something that appears similar to a casual observer, but is not actually identical to the required output, will nonetheless result in the failure of our tests. See the section titled *Sanity-checking your output* below for details on how you can prevent this kind of outcome.

Thinking through your design

Module design

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

Note that this requirement is motivated partly by a desire to build good design habits, but also by the practical reality that maintaining that separation properly will give you a much better chance of being able to reuse your game mechanics, as-is and with little or no modification, in the next project, when you'll be asked to build a complete Columns game (with graphics, keyboard controls, and so on). In a big-picture sense, you can think of the user interface in this project as being a

"throwaway"; while we'll be using it to grade your game mechanics and you'll be using it to verify that the mechanics are correct, the true goal here is the complete version of Columns. So keeping this "throwaway" code completely separate from your game mechanics means that it will be easy to leave it out of your completed version, without causing anything else to break.

One key thing to keep separate is input and output. Your game mechanics code should be neither reading input nor printing output — except maybe temporarily, if you're debugging something.

Module naming

One of your modules must be named **project4.py**, spelled and capitalized exactly that way. That module must be executable (i.e., it should contain an **if __name__ == '__main__':** block), and running that module must be the way to run your user interface and play your game. No other modules will need an **if __name__ == '__main__':** block in them but, of course, you can feel free to add one if you feel that it's necessary.

Using classes and exceptions to implement your game mechanics

Your game mechanics must consist of at least one class whose objects represent the current "state" of a Columns game, with methods that manipulate that state; you can feel free to implement additional classes, if you'd like. Note that this is in stark contrast to the approach used in **connectfour.py** in Project #2, where we used a namedtuple and a set of functions that returned new states. Classes offer us the ability to mix data together with the operations that safely manipulate that data; they allow us to create kinds of objects that don't just know how to *store* things, but also to *do* things.

Some of the methods I found useful in my own implementation of the Columns game state are listed below; this is not an exhaustive list, and you'll probably find a need for additional methods beyond these.

- Get the number of rows and/or columns on the board.
- Determine whether the game is over.
- Create a new faller.
- Handle the passage of time (e.g., moving the faller down, etc.).
- Moving the faller to the left.

Even if your user interface does error checking, your game mechanics must not assume the presence of a particular user interface, so it must check any parameters it's given and raise an exception if the parameters are problematic (e.g., a non-existent row or column, an attempt to make an invalid move, an attempt to make a move after the game is over). Create your own exception class(es) to represent these error conditions.

Using test-driven development

One issue that comes up in the implementation of a program like this one is that it's difficult to test some of the corner cases that come up in the game mechanics by playing your game using this user interface. It can be difficult to duplicate certain situations without going through a lot of work up front. And yet you need to be sure that these issues, and others like them, are handled correctly by your game logic.

As you build your game mechanics, consider using the test-driven development approach that we discussed in lecture. You shouldn't test your shell input and output (e.g., calls to **input()** and **print()**) this way. Instead, just test your game mechanics this way. (Note that this means you'll want to keep code that calls **input()** and **print()** out of your game mechanics altogether, because it's not amenable to this kind of testing, but also because it's not part of how the game is played; it's a user interface detail, which belongs elsewhere.)

When going this route, use the **unittest** module from the Python Standard Library, as we did in the Test-Driven Development lecture. Keeping with convention, the name of your unit test module might start with **test_**, and the rest of its name would ideally be the same as the game mechanics module that you're testing.

Aim for the most complete test coverage you can achieve. Remember that you don't only want to test the simplest scenario; you also want to consider the ways that things might go wrong, and test those, as well.

Other notes about testing

Your ability to specify any initial arrangement of jewels in the field when starting the game is one thing that will really help you (and us!) to test these kinds of scenarios, because you won't need to figure out a sequence of moves that lead from a "traditional" initial state to the scenario you want to test.

Another approach is to figure out some interesting scenarios and write program input that covers these scenarios, saving each one into a file using your favorite text editor. You can then copy and paste these into your program to test and re-test interesting cases as you work.

Sanity-checking your output

We are providing a tool that you can use to sanity check whether you've followed some of the basic requirements above. It will only give you a "passing" result in these circumstances:

- It's possible to run your program by executing a correctly-named module (**project4.py**), spelled and capitalized correctly.
- Executing that module is enough to execute your program.
- Your program reads its input and generates character-for-character correct input for one test scenario, which is similar to the example inputs and outputs shown above.

It should be noted that there are many additional test inputs that will be used when we grade your project, as there are a number of interesting scenarios that are possible. The way to understand the sanity checker's output is to think of it this way: Just because the sanity checker says your program passes doesn't mean it's close to perfect, but if you *cannot* get the sanity checker to report that your program passes, it surely will not pass all of our automated tests (and may well fail all of them).

Running the sanity checker is simple. First, download the Python module linked below:

- project4_sanitycheck.py

Put that file into the same directory as your various Project 4 files. Running the sanity-checking module — for example, by loading it in IDLE and pressing F5 (or selecting **Run Module** from the **Run** menu) — will run the sanity checker and report a result, which will be printed to the Python shell.

Thinking about the future in addition to the present

The next project will revisit the Columns game that you're building here, but will ask you instead to build a playable version of the game using the **pygame** library. We'll be talking a lot about **pygame** over the next couple of weeks in lecture; as we learn more about it, be sure to consider how your design for this project, particularly your game mechanics, can be done in a way that allows you to reuse code in the subsequent project, as you will not want to have to start over from scratch. This means you'll need to be cognizant of how you can separate code that handles input and output from code that implements underlying game mechanics. It also means you'll want to start thinking about how your eventual game might need to use your game mechanics, as we learn more about **pygame** in lecture as we move forward.

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. *Do not* submit the sanity-checker!

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.

Clarifications on "game over" scenarios added by Alex Thornton, Winter 2020.

Clarifications on module naming added by Alex Thornton, Winter 2019.
Requirements for test-driven development and unit testing added by Alex Thornton, Spring 2018.
Clarifications and additional advice added by Alex Thornton, Winter 2018.
Originally written by Alex Thornton, Fall 2017.