

BREAKOUT OVERVIEW

SYNOPSIS

This project, including some of the wording of the instructions, is adapted from assignments by Professor Eric Roberts at Stanford University, and Professor Walker White at Cornell University. The task of this project is to code the classic arcade game Breakout.

ASSIGNMENT SOURCE CODE

The first thing to do for this project is to make sure all of the files are in the Breakout directory (located in Documents/CS.003/breakout). The following files/directories should be there:

`controller.py` – this file contains the controller class (`Breakout`) for this application. This is the primary file that you will modify for this project. However, note that it has no application code.

`__main__.py` – this module contains the application code for this project. It is the module you will run from the command line; you will also slightly modify this file for this project.

`constants.py` – this is a module containing all of the necessary global constants that you will use for this project. If you wish to add more, you **can modify this file**.

`breakoutGraphics.py` – this is a module with the graphics object classes that you will use for drawing in the game window. You should **not modify this file**. See the below documentation for how to use these classes.

`graphics.kv` – this is the Kivy file associated with `breakoutGraphics.py`. **Do not modify this file**, we haven't taught you how these files work.

`Sounds` – this folder contains the sound effect files that you may wish to use as part of your project. You are also free to add more if you wish; just put them in the folder. All sounds must be WAV or OGG files, recorded at 16 bits and sampled at 44.1 kHz (i.e. CD quality).

`Fonts` – this folder contains a collection of True Type Fonts, should you like to use a font other than the default Kivy font. You can put whatever font you want in this folder, provided it is a .tff file. Other font formats (such as .ttc, .otf, or .dfont) are not supported.

`Images` – this folder contains a collection of images that you may wish to use as part of your extensions. The class `graphicsImage` allows you to animate images in this game, should you wish. You can also use them to provide a background (if you add the image *first*).

Because of the module `__main__.py`, to run this program you should change the directory in your command shell just *outside of the folder breakout* and type: `python breakout`. In this case, Python will "run the folder" by executing the application code in `__main__.py`.

You will only modify the first three modules listed above. The class `Breakout` is a subclass of the class `GameController`. Please note, in order to best understand the pre-made modules in the project, read the documentation.

ASSIGNMENT SCOPE

According to the documentation, the class `Breakout` needs to implement five main methods. They are as follows:

Method	Description
<code>initialize()</code>	Initializes the game state. Because of how Kivy works, initialization code should go here and not in the constructor (which is called <i>before</i> the window is sized properly)
<code>update(dt)</code>	Update the graphics objects for the next animation frame; called 60x a second; dt is time since last call
<code>on_touch_down(view, touch)</code>	Called when the user presses the mouse/screen, but has not yet released. The parameter <code>touch</code> holds information about the touch event
<code>on_touch_up(view, touch)</code>	Called when the user releases the mouse/finger on the screen. The parameter <code>touch</code> holds information about the touch event.
<code>on_touch_move(view, touch)</code>	Called when the user moves the mouse/finger on the screen; the parameter <code>touch</code> holds information about the touch event

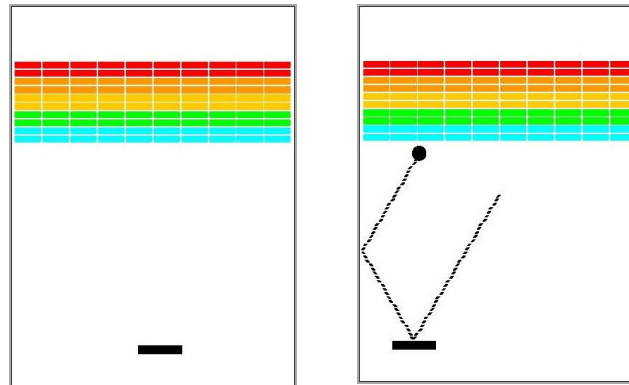
For readability purposes, you are not going to put all of your code within those five methods; remember to develop new methods when you need them – this will help keep each method small and manageable. As a general rule, methods should not be more than 50 lines long (including specification).

You can also create new classes (such as the ball object explained later), or add new fields/attributes to existing classes. When you add a new field, make sure to state the invariant for the field as a constant. You don't need to provide properties (aka enforce invariants), because most of the fields won't be accessed outside of the controller method.

For example, many fields have already been provided to you in `Breakout`, for each one we have provided comments to specify the invariant. Follow those fields as an example for fields you create.

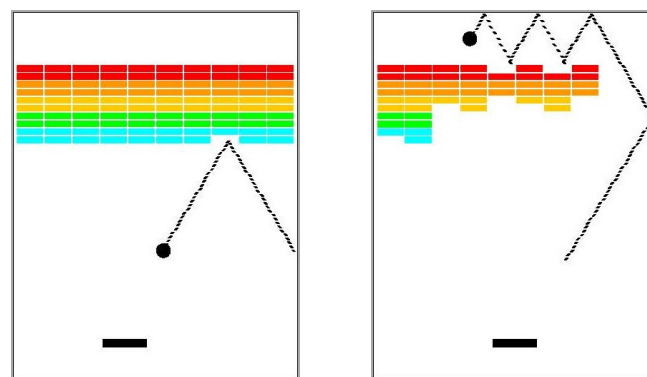
BREAKOUT

The initial configuration of the game `Breakout` is shown in the left-most picture below. The colored rectangles in the top part of the screen are bricks, and the slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension; it moves back and forth horizontally across the screen along with the mouse (or finger, on a touch-screen device) — unless the mouse goes past the edge of the window.

**Starting Position****Hitting a Brick**

A full game gives a user three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. The ball bounces off the paddle and the walls of the game, using the well-known physics principle expressed as—"the angle of incidence equals the angle of reflection"—don't worry, even if you don't know physics, it's easy to implement! Shown in the above-right picture is a start of a potential trajectory, with the ball bouncing off the paddle, and then off the left wall (the dotted line is only there to show the ball's path – i.e. this will not actually appear on the screen).

In the second diagram above, the ball is about to collide with a brick on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears. The left-most diagram below shows the game after that collision and after the player has moved the paddle to put it in line with the oncoming ball.

**Intercepting the Ball****Breaking Out**

Playing a turn continues in this way until one of two conditions occurs:

1. The ball hits the lower wall, which means that the player missed it with the paddle. In this case, the turn ends. If the player has a turn left, the next ball is served; otherwise, the game ends in a loss.
2. The last brick is eliminated. In this case the player wins, and the game ends.

Clearing all the bricks in a particular column opens a path to the top wall. When this situation occurs, the ball may bounce back and forth several times between the top wall and the upper line of bricks without

the user having to worry about hitting the ball with the paddle. This condition, a reward for "breaking out", gives meaning to the name of the game. The last diagram above shows the situation shortly after the first ball has broken through the wall. The ball goes on to clear several more bricks before it comes back down the open area.

Breaking out is an exciting part of the game, but you do not have to do anything in your program to make it happen. The game is operating by the same rules it always applies: bouncing off walls, clearing bricks, and obeying the "laws of physics".

GAME STATE


One of the challenges with making an application like this is keeping track of the *game state*. In the description below, we can identify four distinct phases of **Breakout**. To simplify the project, we have provided you with constants (in module `constants.py`) for four states:

- `STATE_INACTIVE`: before the game starts, when the bricks are not yet set up
- `STATE_ACTIVE`: while the game is ongoing, and the ball is in play
- `STATE_PAUSED`: while the game is ongoing, but the player is waiting for a new ball
- `STATE_COMPLETE`: after the game is over, when the player has won or lost

Keeping these phases straight is an important part of implementing the game. You need this information to implement `on_touch_down` correctly. For example, if the game is ongoing, and the ball is in play, the method `on_touch_down` is used to move the paddle. However, if the game has not started yet, the method `on_touch_down` should set up the bricks and start a new game.

The current state of the application is stored in the field `_state`. You are allowed to add more states when you work on your game, or after Wantagh techDay! However, a basic game should stick to these four states.

BREAKOUT INSTRUCTIONS



Press to play

PART 1A – CREATE A WELCOME SCREEN

To begin, we will start by getting used to adding and removing graphic elements from the view. When the user starts up the application, they should be greeted by a welcome screen, which will consist of a text message. It should look something like the one shown above, but note it does not need to say “Press to Play”—it could say something else, as long as it is clear to the user that they need to click the mouse (or press the screen) to continue. You also can use any font you would like, and later, enhance the welcome screen as much as you would like.

To create a text message, you need to create a `graphicsLabel` object, and add it to the view. Since the Breakout class is a subclass of `GameController`, it inherits an attribute called `view`, which is an instance of `GameView`. Objects of the class `GameView` have a method called `add`, which allows you to add graphics objects “bottom up” using the order they were added to the view (so graphics objects added later are drawn on top).

As you can see from the documentation for `graphicsLabel` and `graphicsObject`, graphics objects have a lot of attributes to specify things such as position, size, color, font size, etc. You *don’t* have to assign all of these attributes before you add the label to the view. If you change them after adding them to the view, then the view will alter its display of the graphics object *at the next animation frame*. You can experiment with these attributes to customize the welcome screen to your liking. Remember that screen coordinates in Kivy start from the *bottom-left corner* of the window.

Since the welcome message should appear as soon as you start the game, it belongs in the method `initialize`, which is one of the first important methods of the class `Breakout`. Put your code there and try running the application. Does your welcome message show up?

PART 1B – INITIALIZE THE GAME STATE

Another thing that you have to do in the beginning is initialize the game state. The field `_state` should start out as `STATE_INACTIVE`, this way we know that the game is not ongoing, and that the program should (not yet) be attempting to animate anything on the screen. Initialize the field `_state` inside of the method `initialize` at the same time you display the welcome message.

PART 1C – DISMISSING THE WELCOME SCREEN

The welcome screen should not stay up forever. The player should be able to dismiss the welcome screen (and start a game) when he or she clicks the mouse or touches the screen. When this event happens, the application will call your function `on_touch_down`.

This means that you should add code to `on_touch_down` that removes the message in your welcome screen. Also, the function should change the game state from `STATE_INACTIVE` → `STATE_PAUSED`, indicating that the game has now started (*but there is no ball yet*). You are not ready to actually write the code for the game, but switching states is an important first activity.

When implementing `on_touch_down`, you need to write the code so that it only removes the message if the state is currently `STATE_INACTIVE` (think *conditional statement!*). Remember, the method `on_touch_down` is called *whenever* the application receives mouse or touch input. While you are currently writing this function to make it remove the welcome message, you will use the method for other things in the future (like move the paddle). You do not want your Breakout application trying to remove a message that is not there. Hence, you now have your first example of a *non-trivial invariant* (conditional circumstances) in `Breakout`:

If the state is `STATE_INACTIVE`, then there is a welcome message; if the state is not `STATE_INACTIVE`, the welcome message should not be shown. Now try out the application. If it is working correctly, you should start up with a welcome message and then get a blank screen when you click the mouse. ** See Appendix for Important Considerations **

PART 2A – SET UP THE BRICKS



Once the game starts, the first step is to put the various pieces on the playing board. Despite the name, you ***should not do this*** in `initialize`. That method is for things that happen *before* the game has started (think the welcome screen). You do not place the bricks until you start the game, which is when the state changes to `STATE_PAUSED`.

Because we change the state in `on_touch_down`, this is a good place to put your code to set up the bricks. However, you will quickly discover that `on_touch_down` gets very complex, because it has to do so many things. Therefore, the best option is to create a helper method (with a name of your choosing) that sets up the bricks and then call that helper method in `on_touch_down`. The helper should be called just after you set the state to `STATE_PAUSED`, which you've already done.

Please place this and all other helper functions that you write after the comment starting "ADD MORE HELPER METHODS", to make it easier for you to find them. It's good practice to specify all helper functions clearly with a docstring. We also recommend that your helper methods be hidden (start with an underscore). You are free to use either camelCase or underscores in naming your methods, but it's good to be consistent.

The helper function should set up the bricks as shown above. The number, dimensions, and spacing of the bricks, as well as the distance from the top of the window to the first line of bricks, are specified using global constants given in module `controller.py`. The only value you need to compute is the x coordinate of the first column, which should be chosen so that the bricks are centered in the window, with the leftover space divided equally on the left and right sides (**Hint**: the leftmost brick should be placed at x-coordinate: `BRICK_SEP_H/2`). The colors of the bricks remain constant for two rows and run in the following sequence: `RED`, `ORANGE`, `YELLOW`, `GREEN`, `CYAN`. If there are more than 10 rows, you are to start over with `RED`, and do the sequence again (conditional statements). We

suggest that you add a constant `BRICK_COLORS` (type: `List`) to your module that lists these colors in an appropriate way to help with this.

All objects placed on the playing board are instances of subclasses of the class `graphicsObject`. For example, bricks and the paddle are objects of subclass `graphicsRectangle`. To define a rectangle, use the attributes `pos`, `size`, `linecolor`, and `fillcolor` to specify how it looks on screen. You can either assign the attributes after the object is created, or assign them in the constructor using keywords; see the documentation for more. When you color a brick, you should set *that color so that its outline is the same color as the interior* (instead of black).

To begin, you might want to create a single Brick object of some position and size and add it to the playing board, just to see what happens. Then think about how you can place the `BRICK_ROWS` (in this assignment, 10) rows of bricks. You will probably need a loop of some kind. It can either be a for-loop or a while-loop; but a for-loop would be easier.

PART 2B - WRITE FUNCTION `FIX_BRICKS`

When you are testing your project, you should play with just 3-4 bricks per row and 1-2 rows of bricks. This will save time and let you quickly see whether the program works correctly when the ball breaks out (gets to the top of the window). It will also allow you to test when someone wins or loses. If you play with the default number of bricks (10 rows and 10 bricks per row), then each game will take a very long time to test and debug.

Unfortunately, testing in this manner would seem to require changing the values of the global constants that give the number of rows and number of bricks in a row. This is undesirable (you might forget to change them back). Instead, we can use the fact that this is an application and give the application some values when it starts out. When you run your application (again, assuming that it is in a folder called `breakout`) try the command: `python breakout 3 2`. When you do this, python puts the list of string values `['breakout', '3', '2']` into the variable `sys.argv` (a global variable in the `sys` module). In the application code in `__main__.py`, we pass this list to the function `fix_bricks`.

Your task is to write the body of `fix_bricks` following its specification (and the hint given as a comment) carefully. You will then be able to start the program as shown above and have just 3 bricks per row and 2 rows. You will need a `try-except` statement in your program to handle the case when the values given are not integers. For example: `python breakout Hello World` should not cause the program to crash. Instead, it should just use the default number of bricks.

PART 3A - CREATE (AND ANIMATE) THE PADDLE

Now, you need to create the paddle (we suggest to make it black, but it can be any color you'd like!). You will need to reference the paddle often, so again we have provided you with a hidden field called `_paddle`. As with the bricks, create an object of type `graphicsRectangle` and add it to the view. You should do this immediately after creating the bricks (either in that helper function, or as part of `on_touch_down`).

The real challenge with the paddle is making it move. In order to do that, you will have to take advantage of all of the `on_touch` methods provided in `Breakout`: `on_touch_down`, `on_touch_up`,

and `on_touch_move`. You may find the code in the appendix, explained by your teacher, to be a useful tool.

To move the paddle, you will take advantage of the fact that each of the `on_touch` methods has a parameter `touch`, which stores a `MotionEvent` object. This object has attributes `x` and `y` which together store the location of the mouse or touch event.

You should really only move the paddle when the user clicks/touches inside of the paddle to begin with (i.e., in `on_touch_down`). To do this, take advantage of the method `collide_point`, which `graphicsRectangle` inherits from its superclass `graphicsObject` (which actually inherits it from Kivy's `Widget` class). See the documentation for more information.

Note: Most of the paddle animation is going to happen in `on_touch_move`. There are actually two ways to approach this problem: the easy way and the hard way. While the hard way creates a better user interface, for sake of time we suggest doing it the "easy way". However, if you are ambitious enough, you try the "hard way" either during or after techDay!

PART 3B – EASY WAY: CENTERING ON THE TOUCH

In the easy way, you always move the paddle so that the x-position of its center aligns with the current touch position. This is as simple as finding `touch.x` and putting the center of your paddle there. Note: This approach causes a minor glitch where the paddle "jumps" to your mouse the first time you touch it. But if you are fine with this glitch, this approach works fine.

PART 3C – HARD WAY: TRACKING TOUCH MOVEMENTS

The "hard way" method will prevent the glitch mentioned above. To do this, you will want to move the paddle in such a way that the x-coordinate of the spot where the player clicked on the paddle remains "locked" to, or aligned with, the touch's current `x` position. To implement this, you need to know the location of both the current `touch` event, and the previous `touch` event. The method `on_touch_down` only receives the current touch as an argument; so again, you will need to create a new field (we did not provide this).

We suggest that the new field store some sort of information about the location of the previous touch event, perhaps relative to the left edge of the paddle when the touch happened. **Do not store the touch event itself**, as that is a mutable object that Kivy may reuse and modify (i.e., this won't work). It is a good idea to make this field `None` whenever the mouse/touch is released. Making it `None` represents valuable state information; both `on_touch_move` and `on_touch_up` can safely do nothing if the field is `None`.

Given both the current touch location and the previous touch location, you can now move the paddle so that the point where it was initially touched by the mouse has the same x-coordinate as the current touch location. This is definitely more challenging than the "easy way", so we suggest you determine if you have enough time to do it this way (i.e., already ahead of most of your peers, or if we make a decision as a group to do it this way!). However, the application definitely works much better using this method, so we suggest that if you don't do it here, try doing it at home! We will post our version of the game on Google Drive after the event to compare your code to ours!

PART 4A - CREATE A BALL AND MAKE IT BOUNCE

Congratulations, you have now finished the "setup" phase of the game, and into the "play" phase. In this phase, a ball is created and moves and bounces appropriately (as you would suspect from our demo). For the most part, a ball is just an instance of `graphicsEllipse`. However, since the ball moves it does not just have a position. It also has a *velocity* (`vx`, `vy`). Since velocity is a property of the ball, these must be attributes of the ball object. There are no such attributes in `graphicsEllipse`, so we have to subclass `graphicsEllipse` to add them. This is what we have done for you in the class `Ball`, which is included at the end of your skeleton code.

Note: the class just includes the fields for the velocity. Properties for these fields, as well as a constructor to initialize them or any other attributes are up to you. Keep in mind that the `pos` attribute inherited by this class specifies the bottom-left corner, and *not the center of the ball*.

You should initialize the fields `_vx`, and `_vy` in the constructor for `Ball` (which you should write). Initially, the ball should head downward, so you should use a starting velocity of -5.0 for `_vy`. The game would be boring if every ball took the same course, so you should choose component `_vx` randomly. Do this with the module `random`, which has functions for generating random numbers. In particular, you should initialize the `_vx` field as follows:

```
self._vx = random.uniform(1.0,5.0)
self._vx = self._vx * random.choice([-1, 1])
```

The first line sets `_vx` to be a random float in the range 1.0 to 5.0 (inclusive). The second line multiplies it by -1 half the time (e.g. making it negative). This is a great feature to make the game more fun, but also more difficult for the user!

PART 4B - SERVE THE BALL

Serving the ball is as simple as adding it to the view (using the provided field `_ball`). In addition, when you serve the ball, you should set the state to `STATE_ACTIVE`, indicating that the game is ongoing and there is a ball in play.

The only difficult part is figuring out where to do this. An obvious place to serve the ball is immediately after you create the bricks and the paddle. But if you do this, then the ball will move before the player can orient him or herself to the game (and hence the player is likely to miss the ball, rendering the game both extremely difficult, and not fun). Ideally, we would like to delay the ball by 3 seconds, giving the player time to get ready.

How do you delay something from happening for 3 seconds? Each of the five basic methods of `Breakout` takes place in a single animation frame. If you add code inside of any of them to stop and wait, the game will appear to lock up and freeze, as it cannot keep animating. The solution is to give `Breakout` a *callback function* which it will execute for you 3 seconds later. You do this with the `delay` method inherited from `GameController`. Give the function name to delay and a time of 3 seconds. The implementation of `GameController` will handle the rest.

If you cannot remember how to use callback functions, remember the callback example that was shown earlier (remember, the callback usage is shown in that example in the application code). You pass the function name (with `self.` in front if the function is a method) without the parentheses. You are not calling the function yet; it will be called after the delay. To work properly, your callback function cannot require any arguments (or arguments other than `self` if it is a method).

Note: In class we showed off the Kivy `Clock` object, which can be used to delay functions. Indeed, `delay` is implemented via `Clock`. However, do **not** use `Clock`, as there are very subtle issues with getting it to work right. You should use `delay` instead. Also, we will walk you through this implementation, as it is difficult for beginners.

PART 4C – MOVE THE BALL

To move the ball, you must implement the last major method, `update`. Essentially, this method is the body of a loop that is called 60 times a second (you don't implement that part, it's done by Kivy automatically). This loop will run "forever" – that is, as long as the application is still running. You may find the code for the animation application, found in the appendix, to be a useful example.

Each time `update` is called, it should move the ball bit-by-bit and change the ball direction if it hits a wall. For now, **do not worry about collisions with the bricks or paddle**. Because of this, the ball should travel through them like a ghost; don't worry, you will deal with collisions in the next part.

The update method moves the ball one step at a time. To move the ball one step, simply add the ball's velocity components to the ball's corresponding coordinates. You might even want to add a method to the Ball class that does this for you (try something like `updatePos()`). Once the ball has moved one step, do the following:

Suppose the ball is going up. Then, if any part of the ball has a y-coordinate greater than or equal to `GAME_HEIGHT`, the ball has reached the top, and its direction has to be changed, so that it goes down. You will do this by setting `_vy = -_vy`. Similarly, check the other three sides of the game screen in the same fashion. When you have finished this, the ball will bounce around the playing screen forever – until you stop it.

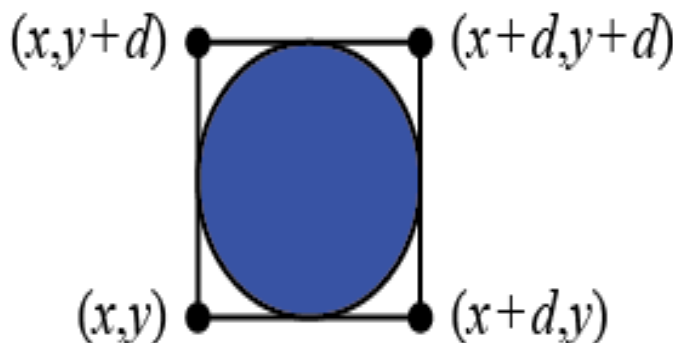
Now, the only tricky part is checking if any part of the ball has reached (or gone over) one of the sides. For example, to see whether the ball has gone over the right edge, you need to test whether the right side of the ball is over that edge; on the other hand, to see whether the ball has gone over the left edge, you must test the left side of the ball. See the attributes in `graphicsObject` for clues on how to do this.

PART 5A – CHECK FOR COLLISIONS

In order to make `Breakout` into a fully functioning game, you have to be able to tell when the ball collides with another object in the window. As scientists often do, we make a simplifying assumption and then relax the assumption later. Suppose the ball were a single point (x,y) rather than a circle. Then, for any `graphicsObject` (`foo` for example), the method call: `foo.collide_point(x,y)` returns `True`, if the point is inside of the object, and `False` if it isn't.

Unfortunately, the ball is not a single point. It occupies a physical area, so it may collide with something on the screen even though its center doesn't. For our purposes, the easiest thing to do — which is typical of the simplifying assumptions made in real computer games — is to check a few carefully chosen points on the outside of the ball, and then see whether any of those points has collided with anything. As soon as you find something at one of those points (other than the ball, of course) you can declare that the ball has collided with that object.

One of the easiest ways to come up with these "carefully chosen points" is to treat everything in the game as rectangles. A `graphicsEllipse` is actually defined in terms of its bounding rectangle (i.e., the rectangle in which it is inscribed). Therefore the lower left corner of the ball is at the point (x,y) and the other corners are at the locations shown in the diagram to the right (d is the diameter of the ball). These points are outside of the ball, but they are close enough to make it appear that a collision has occurred.



You should write a hidden helper method (for what class?) called `_getCollidingObject` with the following specification:

```
def _getCollidingObject(self):
    """Returns: graphicsObject that has collided with the ball
```

```
This method checks the four corners of the ball, one at a time. If one of
these points collides with either the paddle, or a brick, it stops the
checking immediately and returns the object involved in the collision. It
returns None if no collision occurred."""
```

In writing this method, you will find the fields `_brick` and `_paddle` of `Breakout` to be very useful. That is why they are there.

You now need to modify the update method of `Breakout`, discussed above. After moving the ball, call `_getCollidingObject` to check for a collision (remember it's looping, so it will be constantly checking). If the ball going up collides with the paddle, do not do anything. If the ball going down collides with the paddle, **negate the vertical direction of the ball**. If the ball (going in either vertical direction) collides with a brick, remove the brick from the board and negate the vertical direction. Remember our comments above about removing bricks.

PART 6A – FINISH THE GAME

Congratulations, you now have a (mostly) working game! However, there are two minor details left for you to take care of before you can say that the game is truly "finished". Realistically, this game will never be "finished", because you can always continue to extend the game. We've actually uploaded to the Drive a list of extension ideas, and their implementations, for you to look at when you go home. We suggest you try to implement them yourself, you'll learn a lot more. We can't stress enough, be creative! We certainly have not done every possible extension, think of a feature you'd like to have as part of the game, and try to code it. You can always email us (wantaghtechday@gmail.com) if you need help!

PART 6B - PLAYER LIVES

You need to take care of the case that the ball hits the bottom wall. Right now, the ball just bounces off this wall like all the others, which makes the game really easy. In reality, hitting the bottom means that the ball is gone. In a single game, the player should get **three** balls before losing.

If the player can have another ball, the `update` method should put a message on the board somewhere (as you did on the welcome screen) telling the player that another ball is coming in 3 seconds. At this point the state should again be `STATE_PAUSED`, indicating that the game is in session but no ball is currently moving (otherwise, `update` will continue to try to move a ball that does not exist).

In three seconds, you will create a new ball, switch the state back to `STATE_ACTIVE`, and continue the game as normal. Again, do this with the delay function and a callback function, just like you did at the start of the game (note, we walked you through this part, and will again if you need).

PART 6C - WINNING OR LOSING

Eventually the game will end. Each time the ball drops off the bottom of the screen, you need to check if there are any tries left. If not, the game is over (remember conditional statements!). Additionally, as part of the `update` method, you need to check whether there are no more bricks. As you have been storing the active bricks in `_brick`, an easy way to do this is to check the length of this list. When the list is empty, the game ends and the player has won (conditional again!).

When the game ends, and the player has either won or lost, you should put up one last message. Use a `graphicsLabel` to put up a congratulator (or losing) message. Finally, you should change the state one last time to indicate that the game is over. That is what the state `STATE_COMPLETE` is for.

THE END!

APPENDIX

PART 1: IMPORTANT CONSIDERATIONS

While the first part of the project seems relatively straightforward, it is good practice with dealing with controller state. Immediately in this part, you had to add fields beyond the ones that we have provided.

In particular, to remove a graphics object from the view, you have to create a variable that stores the graphics object you wish to remove (in this case, our `graphicsLabel` object). If you put the `graphicsLabel` object that you create in initialize in a local variable, that variable is lost once the application starts, and other methods cannot remove the message. You must put the `graphicsLabel` object in a field so that `on_touch_down` can instruct the view to remove it!

Note, we have not provided a field like this for you; you must add it and the corresponding invariant (described above) to the class. Due to the order in which we create objects, you should only do any assignment of `graphicsLabel` objects to your new field inside a method, the `graphicsLabel` object gets created before the view window is created, and causes the view window to resize in a bad way. Therefore, initialize your new field to `None`.

As you progress through this project, you will find yourself adding more and more fields to the `Breakout` class. Every time one method needs the result from *another* method, you need to use a field to store the information that carries over. The challenge is to add enough fields to get the work done without having too many of them. If two different fields are doing similar things, you should consider consolidating them into a single field. Please place all fields that you add just after the comment starting "ADD MORE FIELDS", to make it easier for you to find them. Make sure to include an appropriate invariant comment with each field; it is good practice in computer science.

PART 2: IMPORTANT CONSIDERATIONS

When you add a brick, it needs to be added in two places. It needs to be added to the view, which draws it; it also needs to be added to a field in the controller, so that you can remove it later (when the ball hits it). We have provided the hidden field `_bricks` for precisely this purpose. Note that `_bricks` is a list and is intended to hold all of the bricks, not just one.

As a result, when you want to add a brick with name (stored in variable) `brick`, you will need to use **both** of the following commands:

```
self.view.add(brick) # Add to view
self._bricks.append(brick) # Add to controller
```

Similarly, when you want to remove a brick, you use the commands

```
self.view.remove(brick) # Remove from view
self._bricks.remove(brick) # Remove from controller
```

You will discover later on that if you added a brick to the view, but not the field `_bricks`, then your ball will pass right through it. On the hand, if you add it to `_bricks` but not to the view, you will get an invisible brick that will block your ball (not yet, however, as you have not yet implemented collisions).

You should make sure that your creation of the rows of bricks works with any number of rows and any number of bricks in each row (e.g. 1, 2,..., 10, and perhaps more). This is one of the things we will be testing when we run your program.

All you need to do is to produce the brick diagram shown above (after the welcome screen). Once you have done this, you should be an expert with graphics objects. This will give you considerable confidence that you can get the rest done.

PART 3: IMPORTANT CONSIDERATIONS

We do not care which approach you take. Whatever approach you use, please ensure that the paddle stays completely on the board even if the touch moves off the board. Our code for this is 3 lines long; it uses the functions `min` and `max`.

Your implementation of the `on_touch` methods should only allow the paddle to be moved when the game is ongoing. That is, the state should either be `STATE_PAUSED` or `STATE_ACTIVE`.