Assignment 1 - Painting with Layers

Welcome to A1 - Painting with Layers!

Hopefully you are about to enjoy, or are enjoying, the first 4 weeks of content for FoA. There's a fair bit of content here, and so it's time to put those skills to the test!

You are likely already familiar with a plethora of prehistoric paint programs. These all have a simple premise: you draw a colour on the canvas, and this overwrites, or sits on-top of, the existing canvas colours.

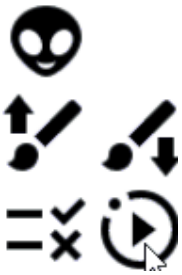
Some applications support transparency, allowing you to blend colours. Some allow you to draw shapes, and others allow you to click and drag certain elements. None of them have the *liveliness* that the FoA Admin team is looking for however. We want to be able to paint sparkles, or a rainbow, on top of our images. Rather than painting with colours, we want to paint with **Effects**!

Throughout this assignment, you'll be working on a paint application, with the following features:

- Multiple different ways to combine effects, and paint with such effects
- Undo/Redo
- A painting replay
- A special action

In doing this, you'll need to demonstrate knowledge on the following topics:

- Stacks, Queues, Lists
- Applications of the Data Structures above



We couldn't just be pleased with that though, we want to make sure you are using the Data Structures mentioned in the unit content! As such, for certain tasks, you'll see a red **X**, denoting particular data structures you cannot use. Using these will **void** *half the correctness marks and all of the design marks for that selection of tasks*.

The only exception to this rules is tuples, which you can use everywhere as a simply way to store (finite) multiple elements in a single location. For example, you can store a tuple within the StackADT to have a stack where each element contains multiple things.

You also cannot access / change any variables within the `data_structures` folder of classes. You can only interact with these classes through the methods they define. **Similar penalties apply if this is ignored**.

Please make sure to read the next slide for some important information before diving into the rest of the assignment.

The template git repository can be found here. Follow the instructions in the "Getting Started with Git" document to copy the repository and get coding.

The rubric for this assessment is available here.

While it isn't a requirement, I suggest acquainting yourself with the Group Contract so you know how we'd like you to use Github for group assignments that follow.

Important Information, Tips and Tricks

Before you get working on the application, please read these tips & tricks to ensure you don't get lost, or lose any silly marks!

Common Mistakes

- You are marked on correctness *in Ed*. It doesn't matter if your code passes locally, it needs to pass in Ed. Contact a TA if you are having trouble achieving parity. Be careful of the specific Python version used, and don't import any other third party modules excluding those already in `requirements.txt`.
- Follow the rules regarding ban on lists, other collections and use the given `data_structures` almost always. Check the task specific page to see what collections are banned for that feature.
- Write clear and concise docstrings for methods you introduce, and type-hint all methods / variables. If introducing new classes, consider using dataclasses to simplify this definition. Separate your code into small methods of at most 20 lines, and try to abstract logic if you find lots of duplicated work. Almost all methods in the sample solution are 10 lines or less. (Please ignore the mess that is `main.py` :})

Initial Setup + Running Tests

To get up and running you want to do a few things:

- Import the template repository into your own - Follow the instructions in the Getting Started with Git page.
- [Optional] Make a virtual environment for the project: `python3 -m pip install virtualenv && python3 -m venv venv` (And then activate the virtual environment) (You may need to change `python3` to `python` or `py` depending on your Operating System and Python version)
- Install the required packages: `python3 -m pip install -r requirements.txt` (Same deal here with `python3`)
- Test it is working by running `python3 main.py` or `python3 -m visuals.basic` (These will probably raise `NotImplemented` errors if you haven't implemented anything, but if your setup didn't work you'll get errors complaining packages aren't installed.)

To run tests, call `python run_tests.py`. Note that you can restrict which tests are run using a second argument. For example, running `python run_tests.py 3` will only run tests which have `number("3.x")` preceding. If you are from in the advanced stream 1054 and would like to include these tests, run with the advanced flag: `python run_tests.py -a`.

Assumptions & Tidbits

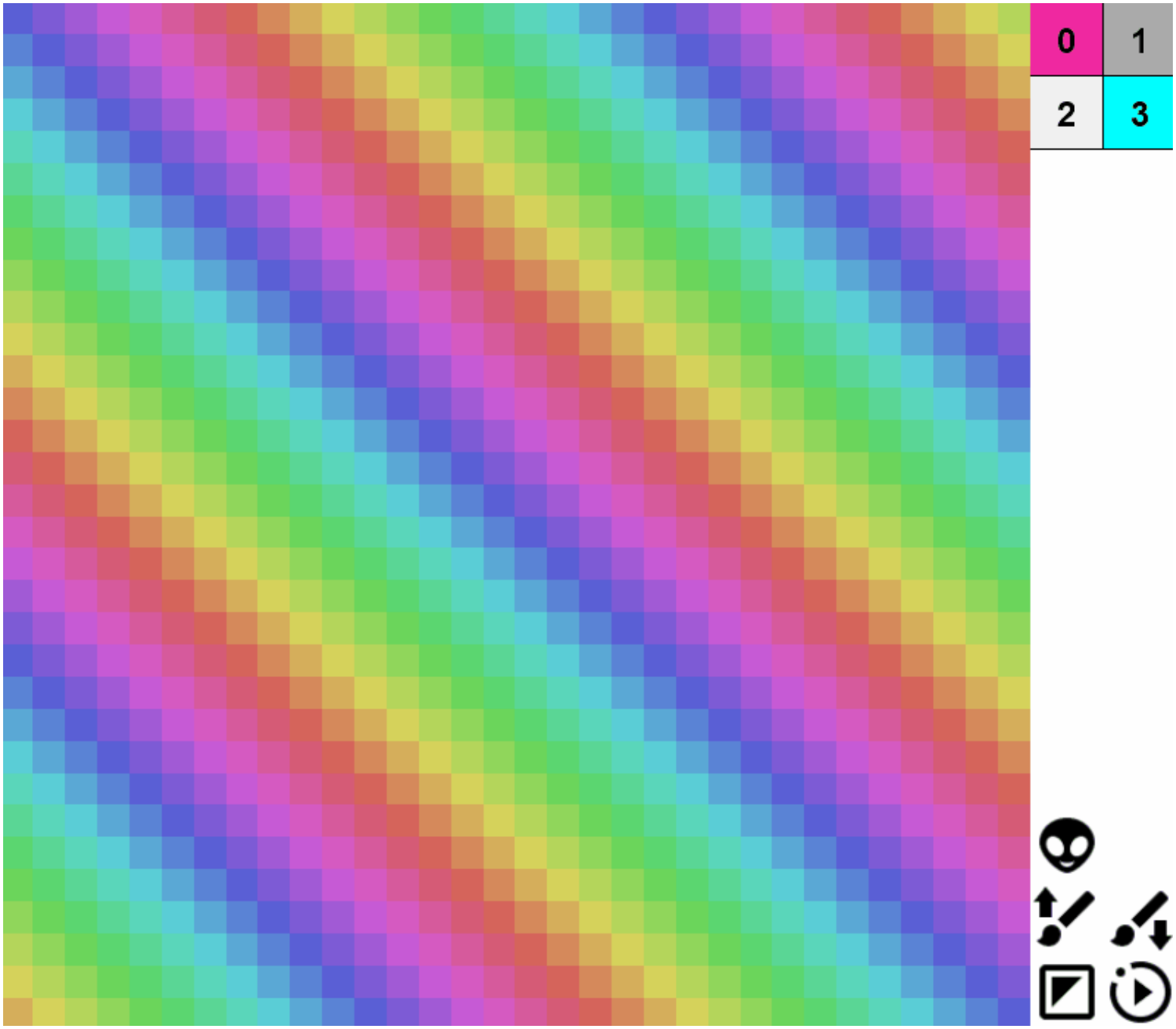
- You can always assume there at most 20 Layers in the application, but design the time complexity of your solution around an unbounded amount.
- dataclasses are used in some of the scaffold code. In short, they do a lot of the initialisation boilerplate for you by inspecting the type hinting for class variables. You can read more about it here.
- This project also uses abstraction with the `abc` module. You can read more about it here.
- Since this is a paint app, here's some recommend listening while you knock these tasks out of the park!

The class/object design

Layer

A Layer/Effect is simply a function that defines, given a previous colour, timestamp, and position on the screen, what the new colour should be. For example, the following snippet defines a beautiful rainbow that moves along the screen:

```
def rainbow(color, timestamp, x, y):
    return tuple(
        int(255*x)
        # Read about HLS to understand these 3 arguments
        for x in colorsys.hls_to_rgb(
            # The hue is determined based on position/time
            (timestamp/20 + x/20 + y/20)%1,
            # The luminance is constant
            0.6,
            # The saturation is constant
            0.6
        )
    )
```



The program only allows for 20 Layers to be defined at the same time. All layers in the program can be accessed by reading from the `ArrayR` object `from layer_util import get_layers`.

Layer Store

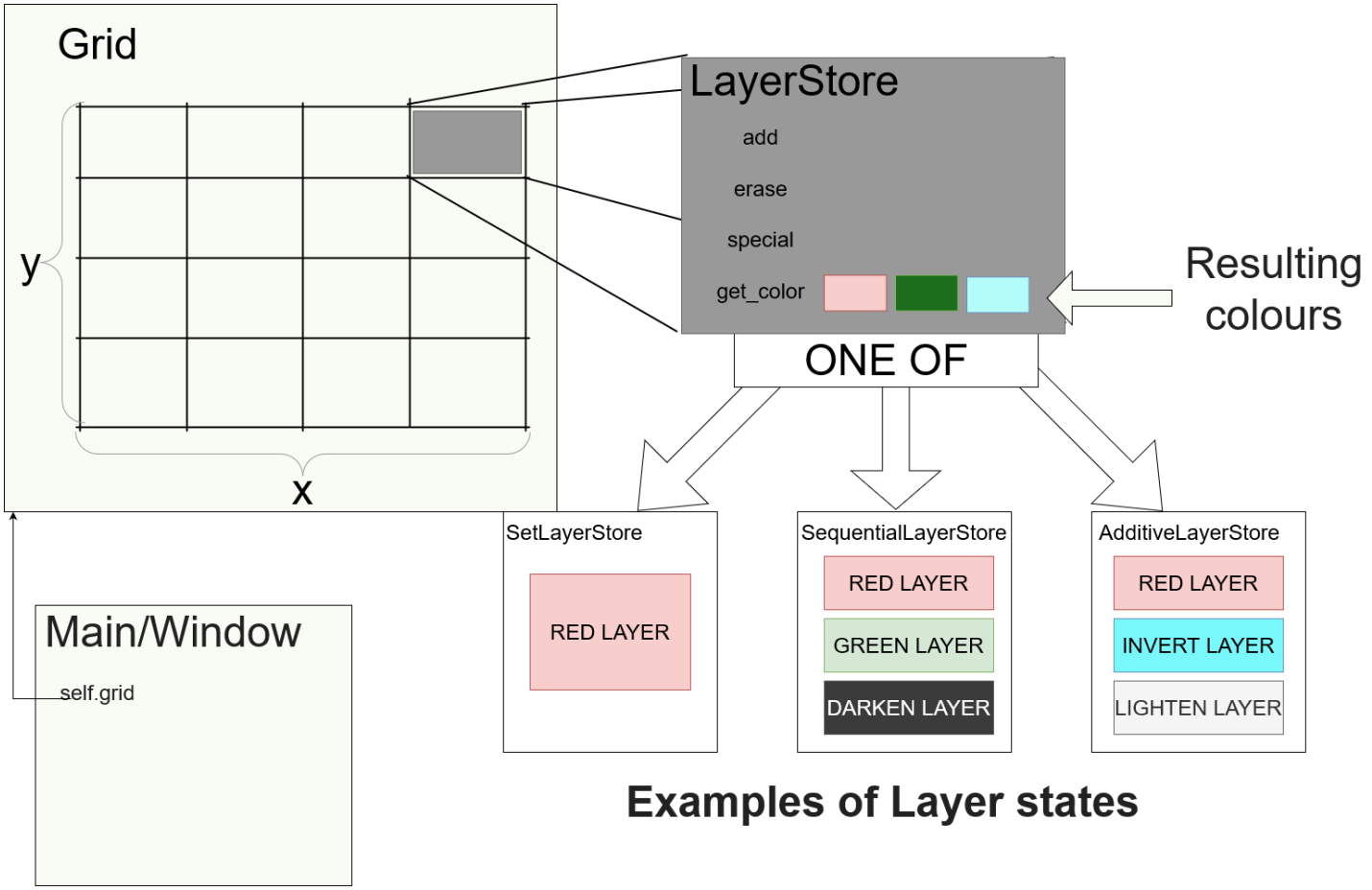
A Layer Store is something stored within each pixel of the grid, and keeps track of what layers are currently being applied

Paint Action

A Paint Action represents a single atomic action that affects the grid. For example, touching the brush at a single grid point, but having the brush add a layer to 14 different grid squares, is a single Paint Action comprising of 14 Paint Steps, where each Paint Step represents adding a layer to one of those grid squares.

Basic Anatomy of the Game

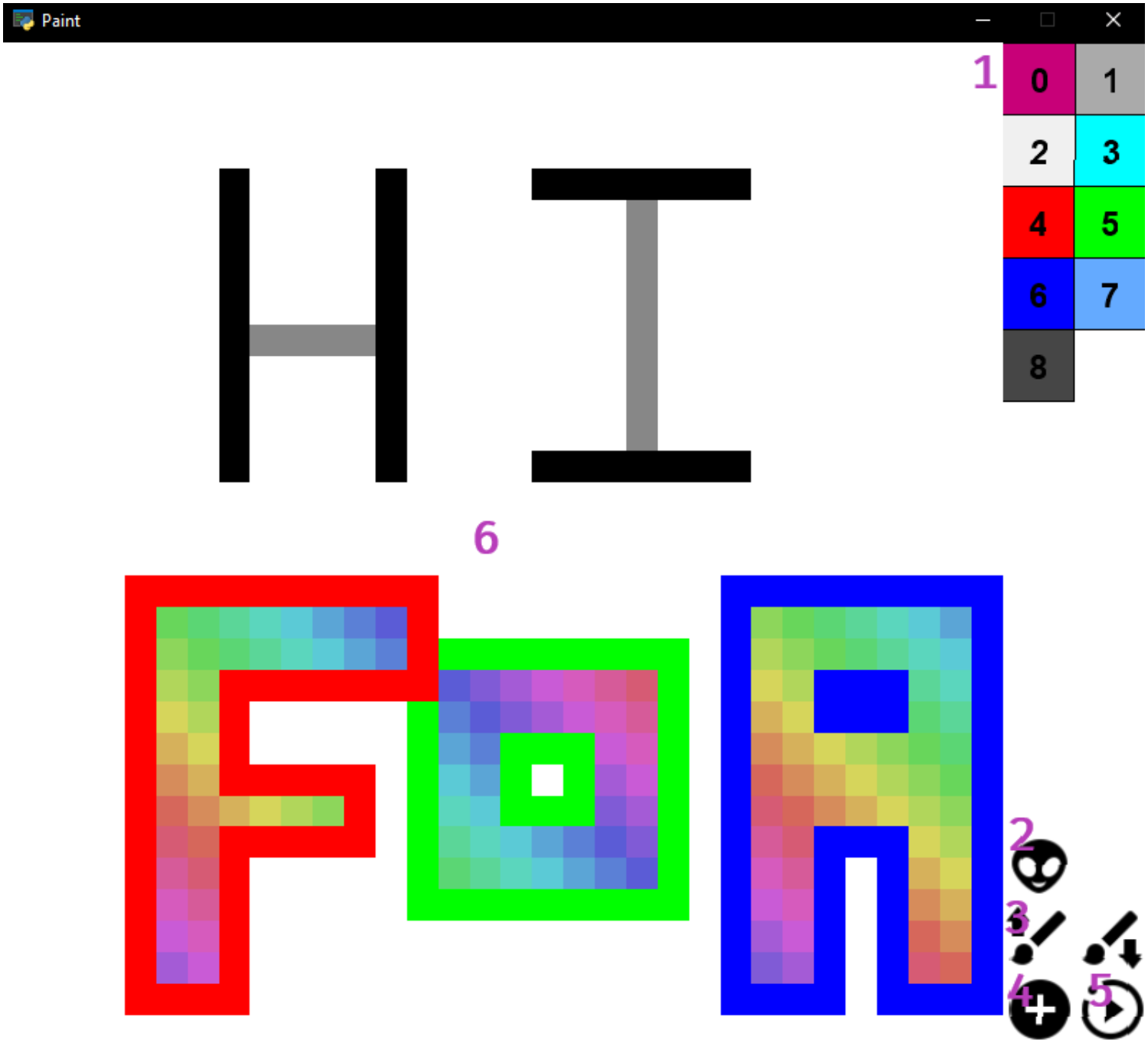
As part of Task 1, you'll be implementing the Grid class and it's components. Attached below is an image outlining the relationship between basic Game Objects (excluding Replay, Undo, and PaintAction objects, which are used in later tasks)



Using the visual editor

By running `python main.py`, provided you've made some headway on the first task, you'll be greeted with the visual editor. Note that using this editor is not a requirement to finish the assignment, but using it should make for a fun and simple way to test your changes as you go (in addition to the unit tests and visual tests provided)

Let's have a quick look at what each button does:



1. Layer selection

Click here to select a particular layer. Layers are defined in `layers.py`.

2. Special mode

Click here to apply the special mode to every grid square. Activates `on_special` in `main.py`.

3. Increase/Decrease brush size

Click here to increase or decrease your brush size. Activates `on_increase_brush_size` / `on_decrease_brush_size` in `main.py`

4. Change Draw Style

This clears the screen and changes the draw style for the grid. Automatically creates a new `grid` instance on `main.py` and Activates `on_reset`.

5. Replay

This replays the entire drawing session that was just done. Activates `on_replay` in `main.py`

6. Draw zone

Left click/drag to draw your currently applied layer on the grid.

Keyboard Shortcuts:

- CtrlZ/CtrlY - Undo and Redo an action. Activates `on_undo` / `on_redo` in `main.py` . Hold to continue undoing/redoing.

[TASK] Getting Started - The Grids & SetLayerStore

In this selection of Tasks, you cannot use inbuilt Python lists, dictionaries, sets, and other collections. You can use any data structures in the `data_structures` folder. The one exception to this is that you can use inbuilt lists when creating the `PaintAction` objects.

To get started, we'll add basic painting functionality to our app.

Grid

Start by implementing the 3 existing functions on the `Grid` class. The `Grid` class should create one instance of the `LayerStore` for each grid square, and these layer stores should be accessible by entering `grid[x][y]` (You might need to implement some magic methods on the grid class).

The Grid class should store all current grid information, including the individual Layer Stores.

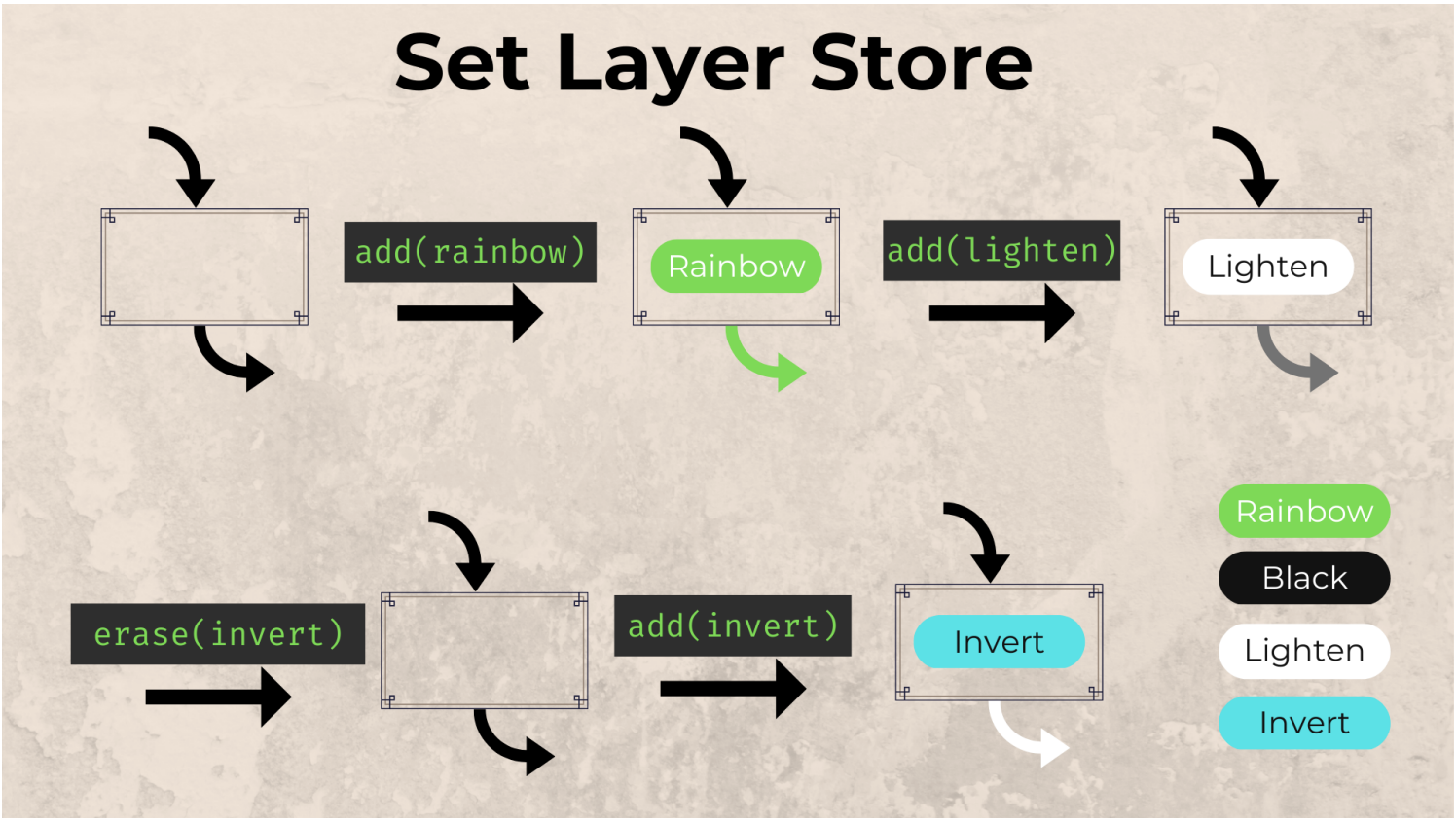
The Grid has a maximum and minimum brush size, which are set through Class variables `MAX_BRUSH` and `MIN_BRUSH`.
When `on_paint` is called in `main.py`, the `grid` object should paint all grid squares within `d` Manhattan distance of the grid square at `(px, py)` with the `layer`, where `d` is the current brush size (Which defaults to `DEFAULT_BRUSH_SIZE`).

So if the brush size is currently 1, then painting at `(0, 5)` should paint positions `(0, 4)`, `(0, 5)`, `(0, 6)` and `(1, 5)` with the specific layer (`(-1, 5)` is ignored because it is an invalid grid position)

Set Layer Store

Next, implement the `SetLayerStore` class, and hook this up to the Grid class. Once all of this is complete, you should be able to pass `tests/test_layer_stores/test_set_layer.py`.

The `SetLayerStore` simply remembers the last layer that was applied, and applies that. Any further additions to the store wipe the previous layers, and erasing a `SetLayerStore` just means having no layers applied.
The special mode on a `SetLayerStore` keeps the current layer, but always applies an inversion of the colours after the layer has been applied. So if previously your Layer output `(100, 100, 100)`, then it would now output `(155, 155, 155)`. See `tests/test_layer_stores/test_set_layer.py` for examples of this at play.



Edit some of the event functions in `main.py`, including `on_paint` and `on_special`, so that when running `main.py`, you can now:

- Draw on the screen with layers
- Click the special button to trigger the special action on all grid squares
- Click the increase / decrease brush size buttons to increase / decrease brush size.

When `on_paint` is called in `main.py`, you should paint all grid squares within d Manhattan distance of `self.grid[px][py]`, where d is the current brush size (which should be retrieved `self.grid`).

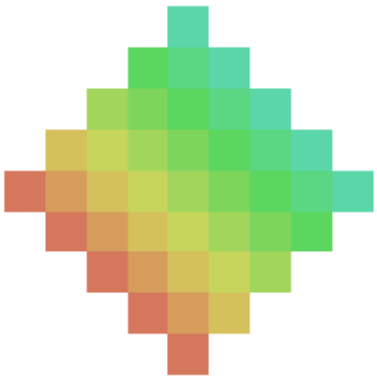
See `tests/test_misc/test_window.py` for an example of this at work.

You don't necessarily need to edit all functions in the STUDENT PART of `main.py` to get this to work. `self.grid` should be already initialised for you in main, see the implementation of `on_increase_brush_size`.

You may want to implement additional methods to existing classes to achieve the intended functionality. Don't be afraid to modularise!

Confirming the visual program works can be done by running `python -m visuals.basic`. Output should look as follows:

0	1
2	3



[TASK] More LayerStores - Additive & Sequence Layer Store

In this selection of Tasks, you cannot use inbuilt Python lists, dictionaries, sets, and other collections. You can use any data structures in the `data_structures` folder, except for `ArrayR`. You can use other data structures which use `ArrayR`, such as `ArrayStack`, but not `ArrayR` explicitly.

Next, we'll add two more layer stores types, allowing for different drawing styles in the app.

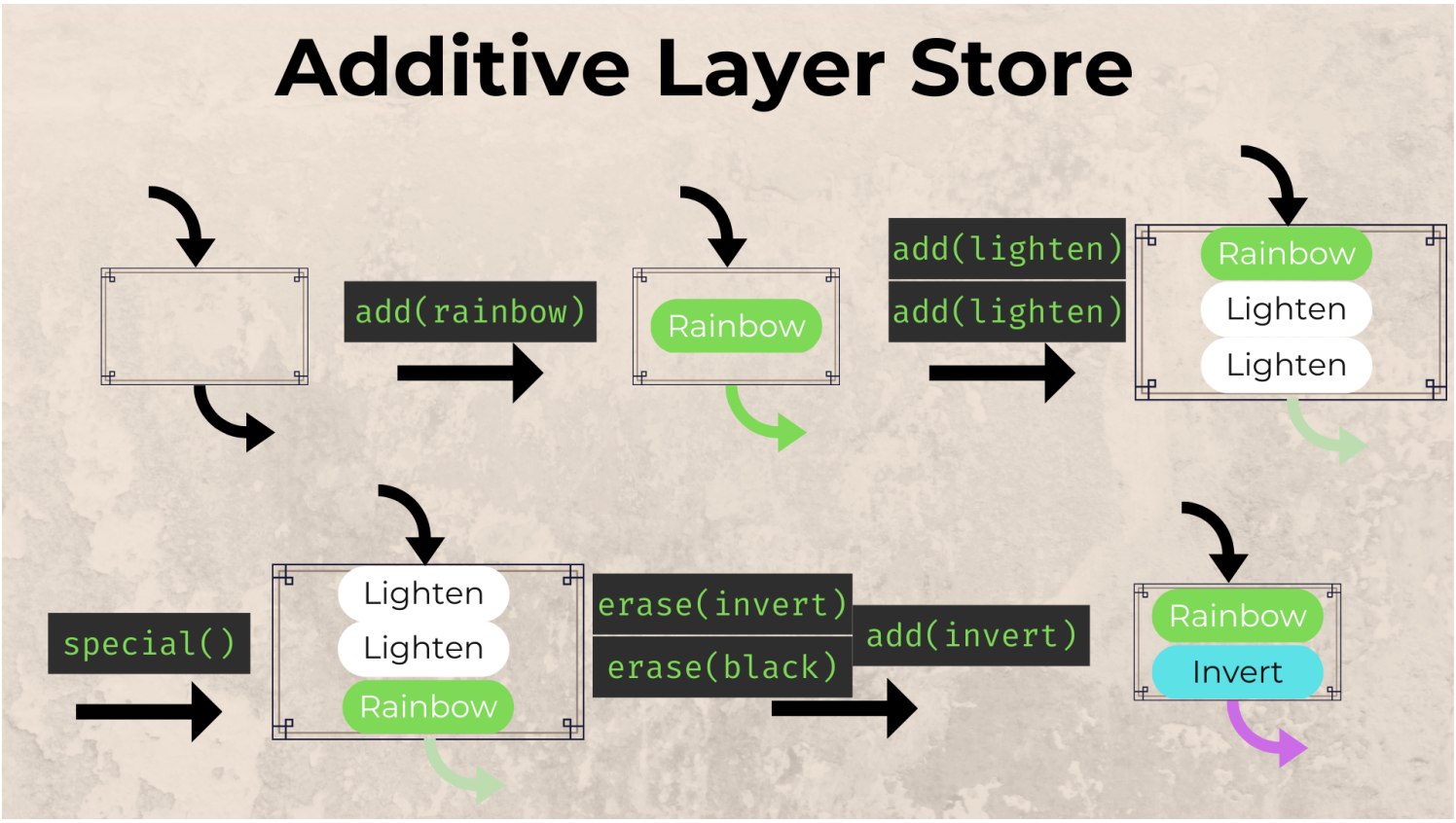
Additive Layer Store

The Additive Layer Store simply applies layers consecutively. Whenever a store has a collection of layers, these layers are applied one-by-one, from earliest added to latest added.

Erasing from an Additive Layer always removes the oldest remaining layer.

The special mode on an additive layer reverses the "ages" of each layer, so the oldest layer is now the youngest layer, and so on.

Since multiple copies of the same layer can be used in this mode, make the capacity of the store at least 100 times the number of layers.

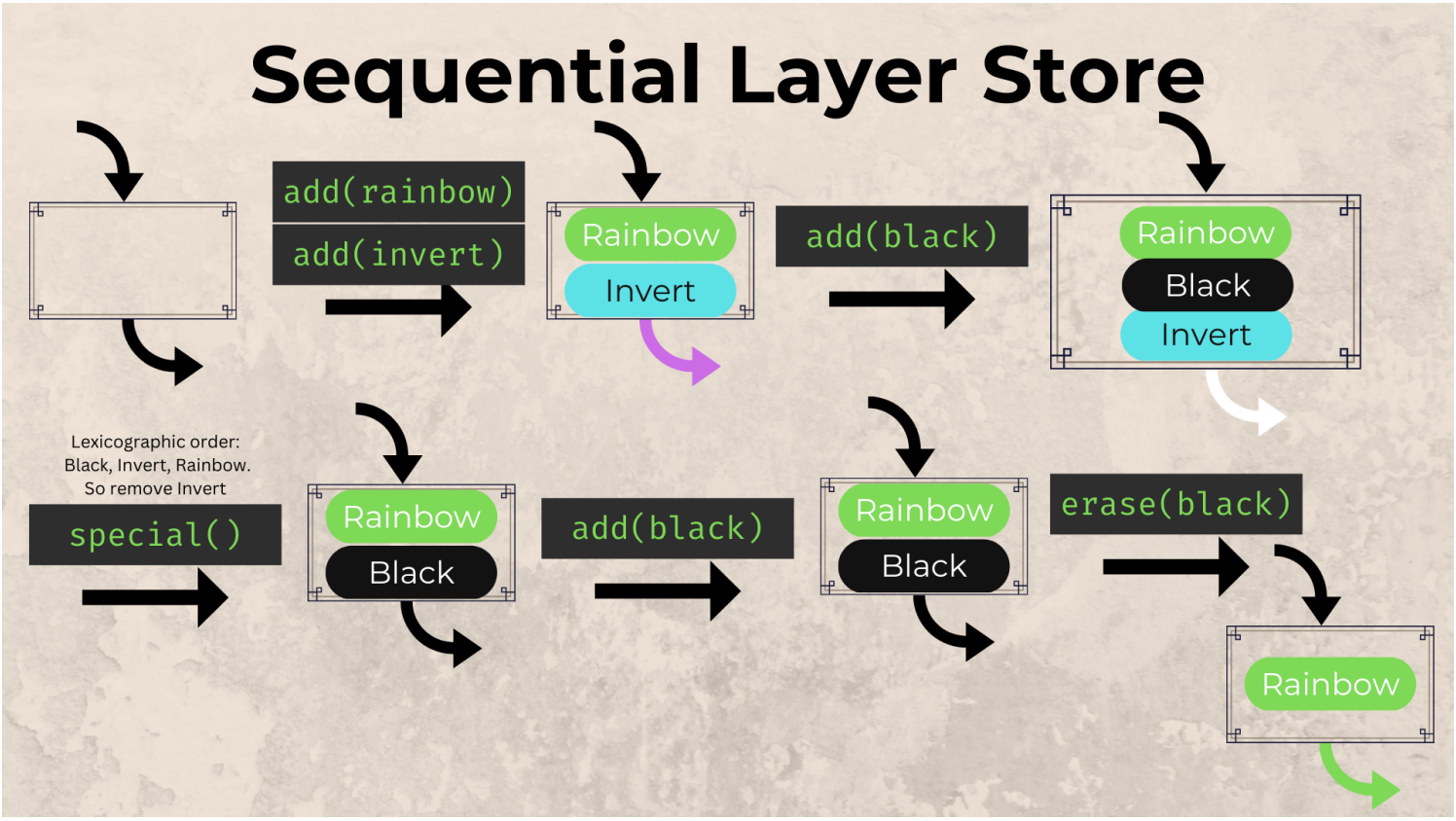


Sequential Layer Store

The Sequential Layer Store simply keeps tracking of every layer as either "applying" or "not applying".

`add` makes the Layer "applying", while `erase`ing a Layer makes it "not applying". The colour of the Layer store is calculated by applying each Layer which is currently "applying", in order based on their index (accessible through `layer.index`. The index is a unique integer from `0` to `len(get_layers())-1`, which is defined when the layer is registered).

The special mode on a sequential store removes the median "applying" layer based on its name, lexicographically ordered, in the case of an even number of applying layers, select the lexicographically smaller of the two names.

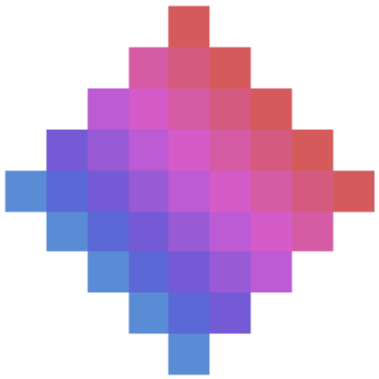


Edit your `Grid` implementation so that the layer stores created depends on the `draw_style` passed in.

Once all of this is complete, you should be able to pass `tests/test_layer_stores/test_add_layer.py`, `tests/test_layer_stores/test_seq_layer.py`.

Now, your paint program should support editing in all three layer modes. Running `python -m visuals.styles` should look like this:

0	1
2	3



[TASK] Undo & Replay Features

In this selection of Tasks, you cannot use inbuilt Python lists, dictionaries, sets, and other collections. You can use any data structures in the `data_structures` folder, except for `ArrayR`. You can use other data structures which use `ArrayR`, such as `ArrayStack`, but not `ArrayR` explicitly.

You can assume that in both of these features, the maximum number of actions does not exceed 10000.

Now that all drawing modes are functional, we want to add some extra features to the paint program.

Before we can do that though, we need to slightly modify our existing LayerStore functionality.

Edit the `add` and `erase` methods for each of your Layer Stores so that they return a boolean representing whether the state of the store was changed.

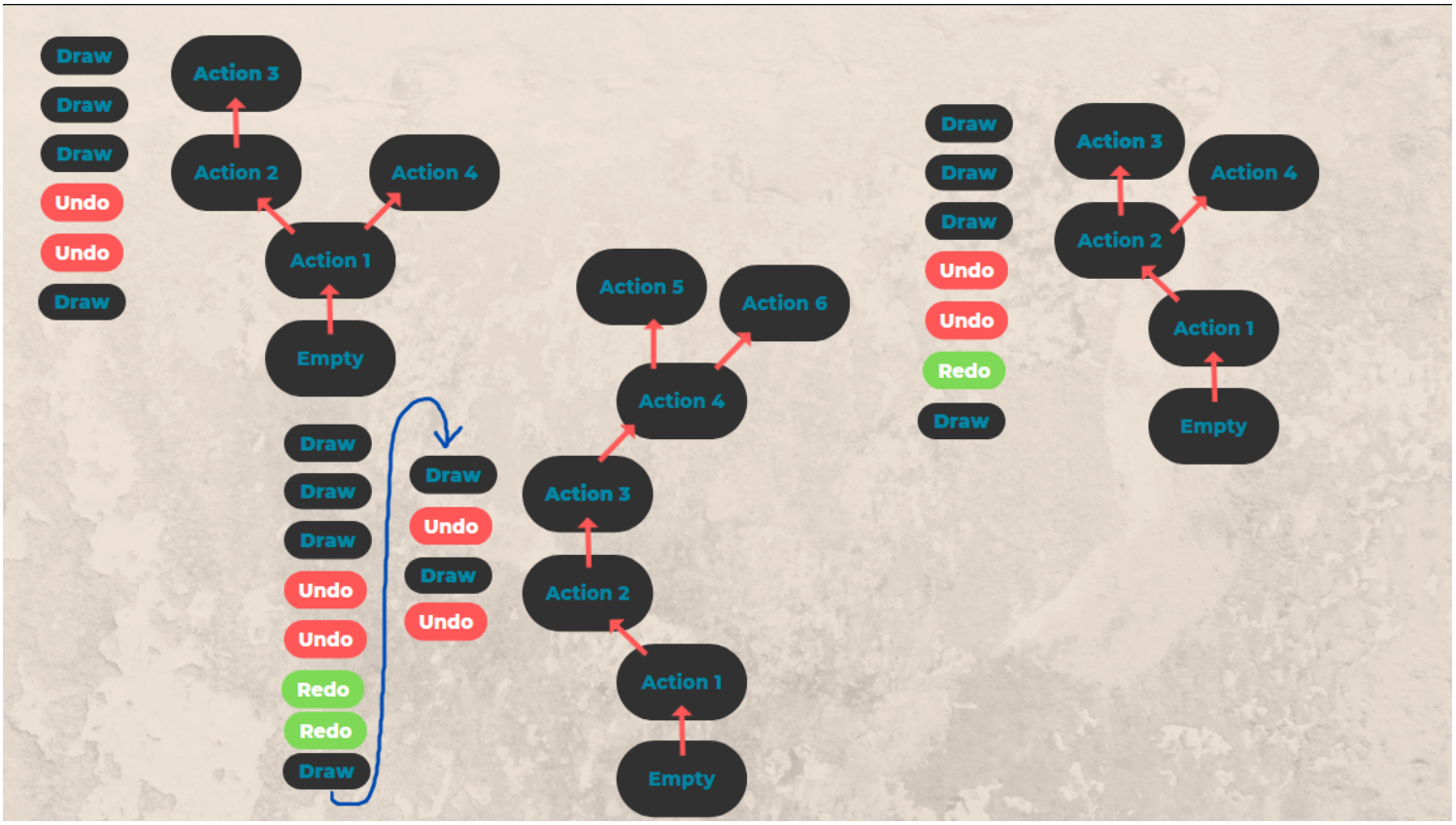
Undo

We want to be able to undo and redo draw actions on the grid. Whenever `on_paint` is called, this should count as a new action which is added to the `UndoTracker`.

Since some layer stores aren't easily reverted to their previous state, we've decided that simply calling `erase` with the same layer that was originally drawn is a fine undo action (Note that in the additive layer mode, this does not actually always undo the action).

An undo action should only be done on a grid square if the action we are undoing actually changed the state of the Layer Store. This is why `add` and `erase` return booleans.

The undo feature can be though of as a "tree" of actions, where undoing an action takes you down the tree and drawing on the screen adds a new branch to the tree at your current position.



The undo tracker only ever needs to move down the tree or back up the tree on the branch you just came down from, and so most of this tree is unused and doesn't necessarily need to be stored in memory.

Hint: You *might* (but also might not) need multiple structures to represent this functionality.

Your task is to implement the `UndoTracker` class methods given in the scaffold, and hook these up to

your `main.py` / `grid.py` code so that pressing Ctrl+Z/Ctrl+Y undoes and redoes your paint jobs.

Replay

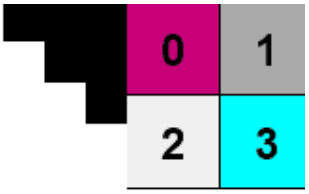
Now that we can paint, undo and redo, we want to be able to play of this information back so that we can show others how we made our masterpiece (Much like the feature in Chicory: A Colorful Tale's home screen)

The replay feature is generally a simpler version of the `UndoTracker`, we just want to keep track of every action taken (Including Undos, Redos, Drawing, and Specials), and then, when playback has started, feed all of these actions back to the game so that it can be re-run. Your task is to implement `ReplayTracker`, and then hook these up to `on_replay_start` and `on_replay_next_step` so that the replay feature is functional.

Once the replay is finished, the replay should clear its memory, so that older actions are not included in the previous replay.

Once all of this is complete, you should be able to pass `tests/test_undo.py,tests/test_replay.py`.

Now, your paint program should support editing, undo, redo and replay. Running `python -m visuals.complex` should look like this:


0
1
2
3

Code Submission

Submit all of the code in this box. This is required for your submission to be marked.

The git repository can be found here.

Week 1 - Check In

Post the link to your Pull Request for Week 1 submission.

没有回应

Week 2 - Check In

Post the link to your Pull Request for Week 2 submission.

没有回应

Post the link to your team members Pull Request for Week 1 submission, which you commented on.

没有回应

Week 3 - Check In

Post the link to your Pull Request for Week 3 submission.

没有回应

Post the link to your team members Pull Request for Week 2 submission, which you commented on.

没有回应

Week 4 - Check In

Post the link to your Pull Request for Week 4 submission.

没有回应

Post the link to your team members Pull Request for Week 3 submission, which you commented on.

没有回应

[OPTIONAL] Make your own features!

Now that your paint program is fully functional, you can come up with some of your own features!

Some possible features might be:

- Custom shaped paint brushes
- Screenshot button
- Export replay to file
- Change it into a level editor for a 2d platformer!
- Add a layer which chroma keys in a video file
- Add a paint bucket tool that searches for the largest contiguous area which currently does not have this layer and applies it
- Add an eraser that clears the grid at certain pixels, whose actions can also be undone.

Just make sure that your additions do not break previous test for the actual assignment.

If you do make something cool, be sure to share it on the forums!

Changelog

9 March 2023 [17:54] - Corrected the name of the test file.