Lab 7: Sketching a GUI

This week we will finally start building a GUI version of our 2048 game. The code will be very similar to the command line version of the game and several parts will be copied directly. We will focus mainly on developing the widget layout and connecting keyboard events to make the game work within the `tkinter` event loop.

Getting started

Start by creating a new file `gui.py`.

```
import tkinter as tk
from random import randint, choice

import core

class Game(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("py2048")
        self.configure(padx=50, pady=50)
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]

        self.set_random_empty_tile(2)
        self.set_random_empty_tile(2)

    def set_random_empty_tile(self, value):
        while(True):
            row = randint(0, 3)
            col = randint(0, 3)
            if not self.grid[row][col]:
                break
        self.grid[row][col] = value

if __name__ == "__main__":
    g = Game()
    g.mainloop()
```

The above code doesn't do much that is new. We are importing some things we know we will need.

In the `__init__` method we add a title and some padding. We set up an empty grid and add a few random `2`'s, just like before.

Adding some basic widgets

Now, we need to insert some widgets to build the user interface. We will start with the score indicator. This will be constructed from two `tk.Label` widgets, one with the fixed text "Score: " and another with a `textvariable` set to a `tk.IntVar` attribute which we will update with the score.

First, add the following to the top of the file, below any import statements and above the `Game` class definition.

```
normal = ("Helvetica", 24, "bold")
```

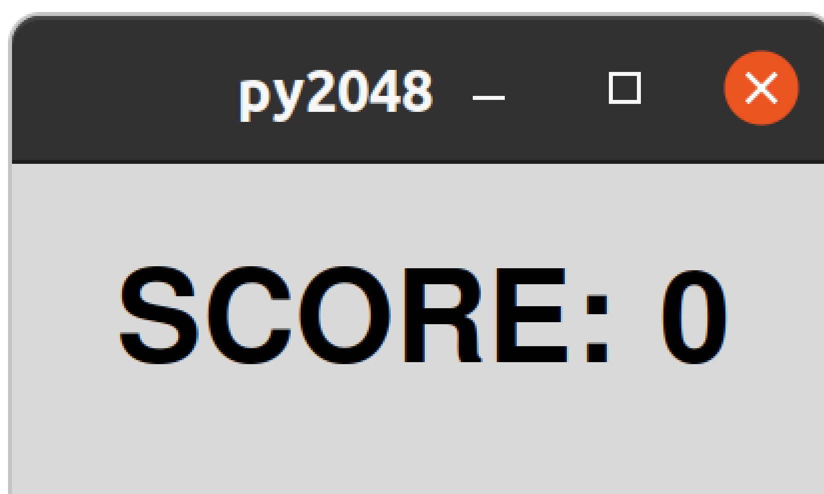
> We will use this font definition for all our labels.

Now add the following to the `__init__` method.

```
        self.score = tk.IntVar()
        tk.Label(text="SCORE: ", font=normal).grid(column=0, row=0)
        tk.Label(textvariable=self.score, font=normal).grid(column=1, row=0)
```

> The above defines a new attribute `self.score` as a `tk.IntVar` and creates two labels which are both inserted into the GUI using `grid`.

The result should look something like this.



We can set up a basic column configuration so the second column always takes the space.

```
        self.columnconfigure(1, weight=1)
```

and we can add `sticky="w"` as an argument to the score number label so it is always left aligned.

```
        tk.Label(textvariable=self.score, font=normal).grid(column=1, row=0, 
sticky="w")
```

> Note: This is an update to the existing line of code.

This should keep the text in the score labels together.

Creating the grid

The grid of tiles is now all we really need. To implement this, we will create a `tk.Frame` to hold all the tiles in four columns and rows. Add the following to the end of the `__init__` method.

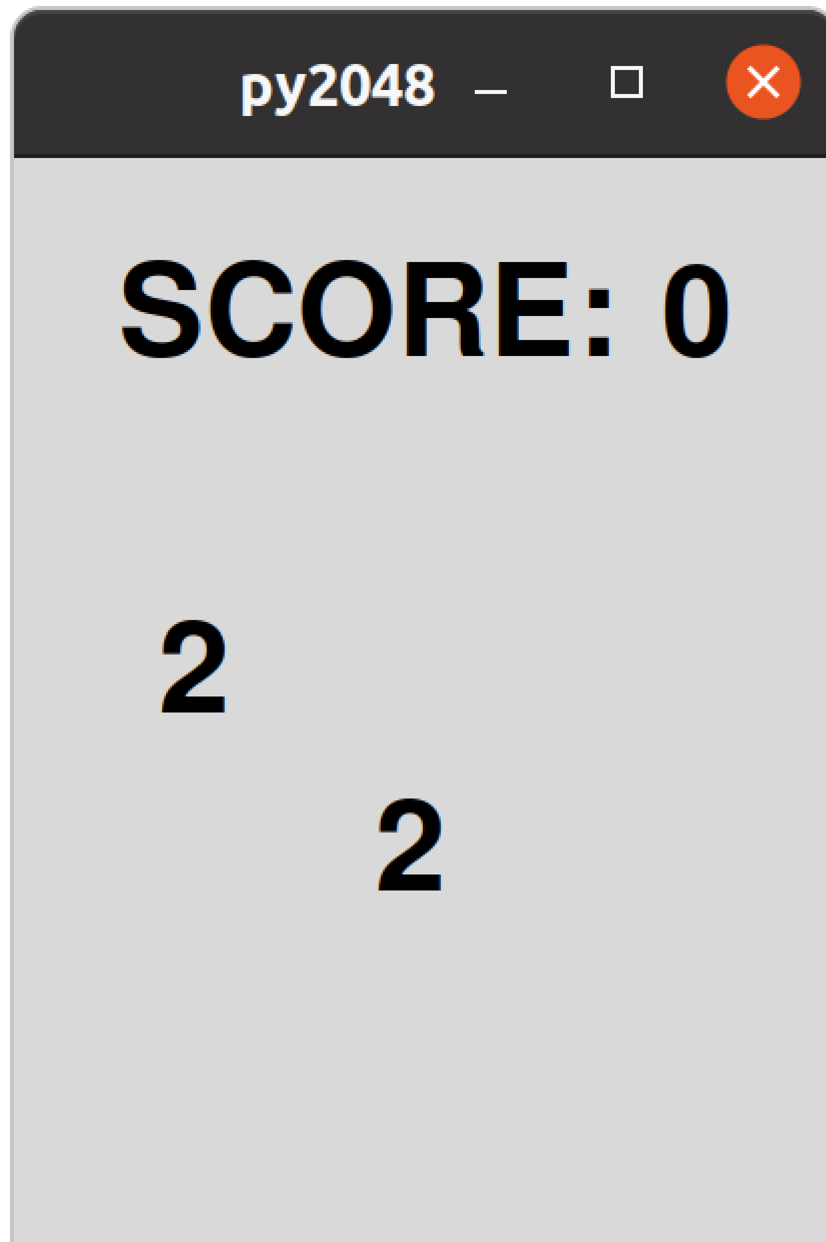
```
        frame = tk.Frame(padx=10, pady=10)
        frame.grid(column=0, row=1, columnspan=2, sticky="news")
```

> The above includes padding within the frame. The frame is also set to span our two columns and take up the full space within these two grid cells.

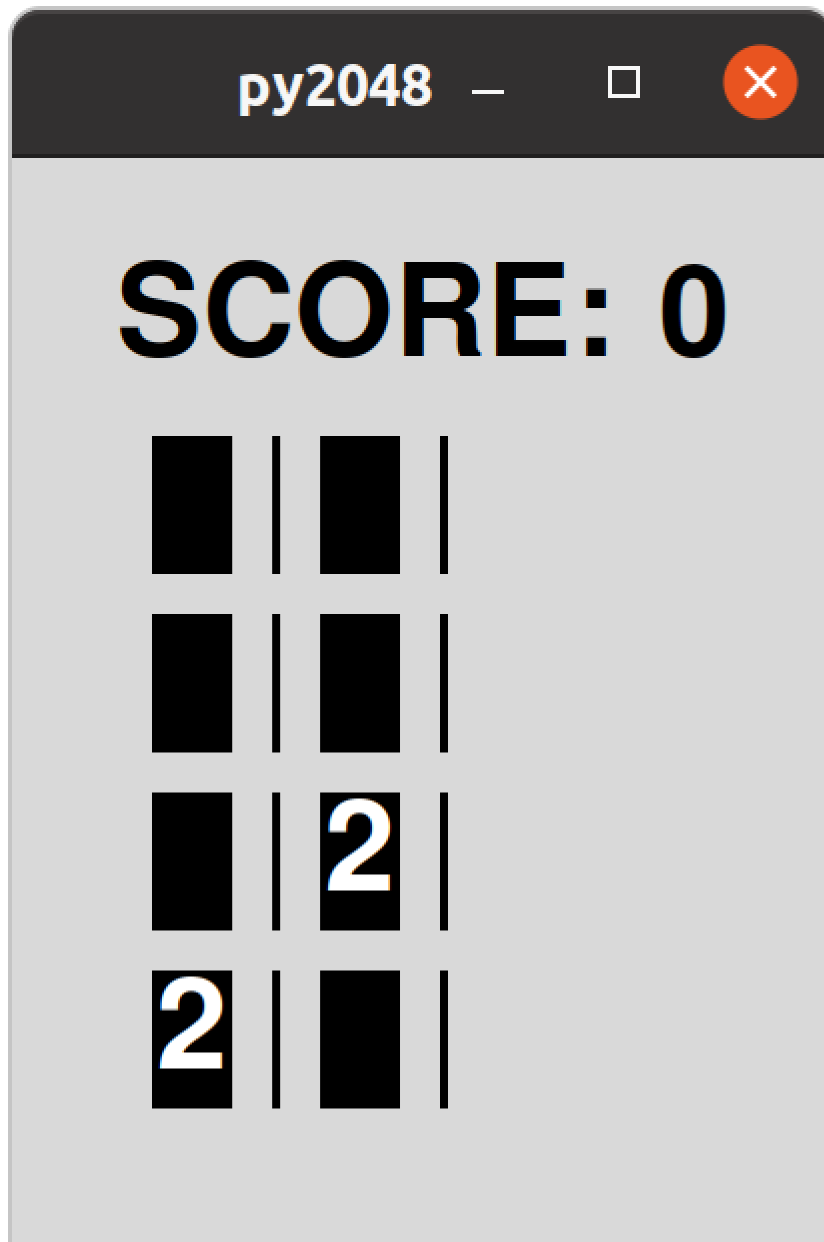
Each tile will be a label, within the frame widget, showing the value of the given tile. There are two challenges here. First, we need to create the labels in the right locations. To do this, we will loop over four rows and columns and set the `row` and `column` arguments to `grid` accordingly.

```
        for row in range(4):
            for col in range(4):
                tile = tk.Label(frame, text=self.grid[row][col], font=normal)
                tile.grid(row=row, column=col, sticky="news", padx=10, pady=10)
```

This should produce a result which looks something like this.



It's not clear in the figure, but the label sizes are being determined by the content. Try setting the label colours (using the `bg` and `fg` arguments) to highlight the problem.

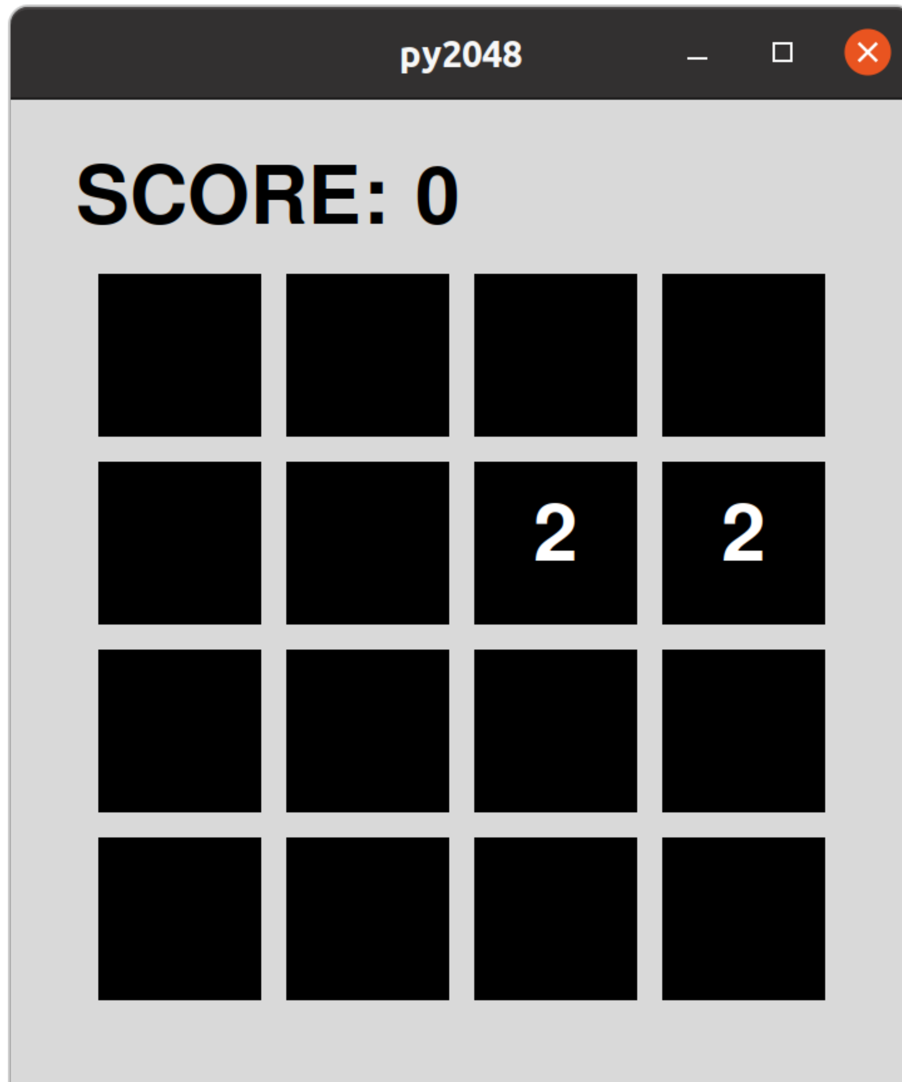


To fix this, we can configure the grid rows and columns within the frame.

```
        cell_size = 150
        for i in range(4):
            frame.rowconfigure(i, minsize=cell_size)
            frame.columnconfigure(i, minsize=cell_size)
```

> The `cell_size` variable (and font) may need to be adjusted for your screen size.

The result should be that each cell is expanded to fill the specified size.



Now, we will need to update the text in the labels as the game grid updates. To do this, we can create a collection of `tk.StringVar` objects that we can manipulate. We will use a dictionary with the keys set to two-tuples like this.

```
        self.tiles = {}
        for row in range(4):
            for col in range(4):
                self.tiles[(row, col)] = tk.StringVar()
                tile = tk.Label(frame, textvariable=self.tiles[(row, col)],
font=normal, bg="black", fg="white")
                tile.grid(row=row, column=col, sticky="news", padx=10, pady=10)
```

> This is an update of the existing code. Each label now has its textvariable argument set to the new `tk.StringVar` which is stored in our `self.tiles` dictionary so we can edit the text easily.

We should now see that the labels are now all empty. To update the labels we need to loop over the grid and set the value of each `tk.StringVar` object in `self.tiles`. We will use a method to do this, so we can call it after every move.

Add the following method

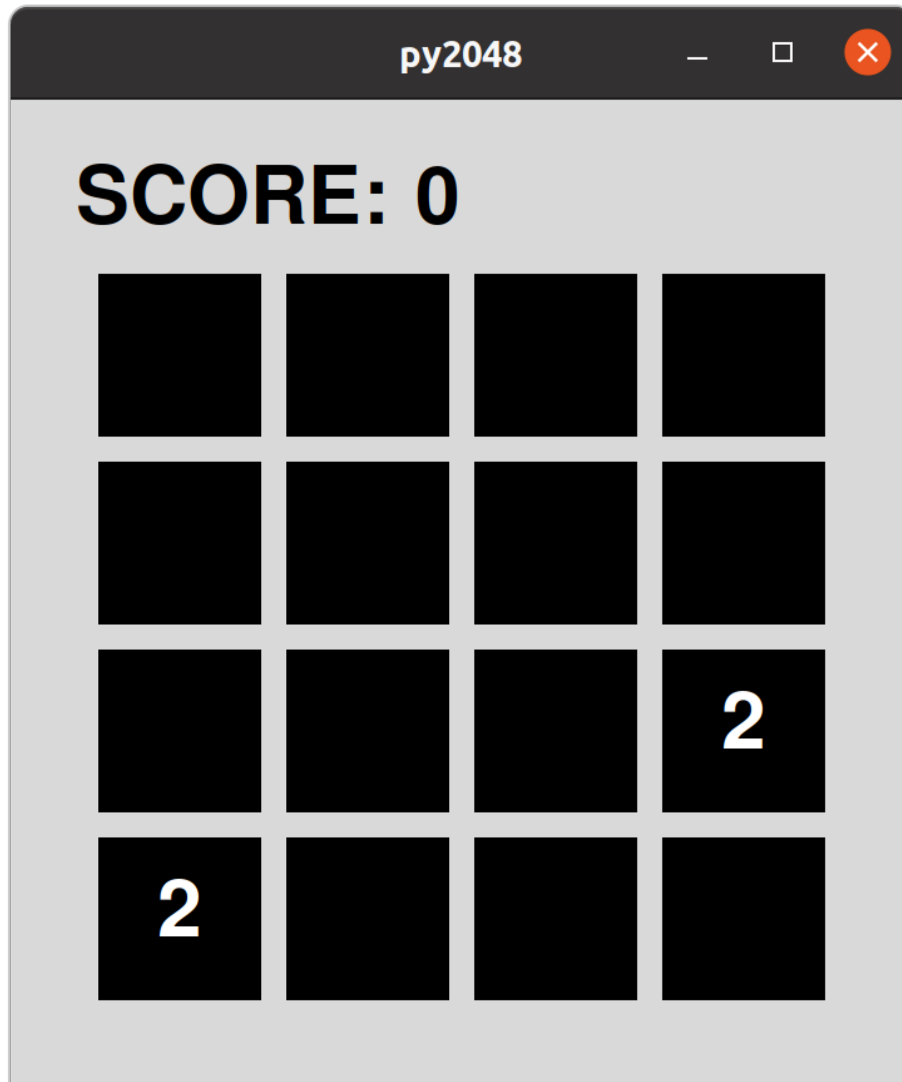
```
    def update(self):
        for row in range(4):
            for col in range(4):
                self.tiles[(row, col)].set(self.grid[row][col])
```

> Make sure you call the new `self.update()` method at the end of the `__init__` method

The result is encouraging, but not ideal.

We want to leave the `None` tiles empty, just like we did in the command line version. We can do this by simply using an `or` statement, so if the tile value os `None` the result will be an empty string.

```
    def update(self):
        for row in range(4):
            for col in range(4):
                self.tiles[(row, col)].set(self.grid[row][col] or "")
```



> Much better

Making a move

Now we have a properly initialised grid, we need to activate keyboard events to move the tiles. This is going to be somewhat familiar but will utilise the `tkinter` event system to trigger the move.

We will start by defining what to do when we process a move. Add a `process_command` method.

```
    def process_command(self, command):
        next_grid = self.moves[command](self.grid)
        if next_grid != self.grid:
            self.grid = next_grid
            new_tile = choice([2, 2, 2, 4])
            self.set_random_empty_tile(new_tile)
            self.game_over = core.is_game_over(self.grid)
```

> This is exactly the same as we had in the command line version, without the score calculation.

We will call this method whenever the user presses an arrow key. To catch the keypress events, we need to call `self.bind()`. Add this to the `__init__` method.

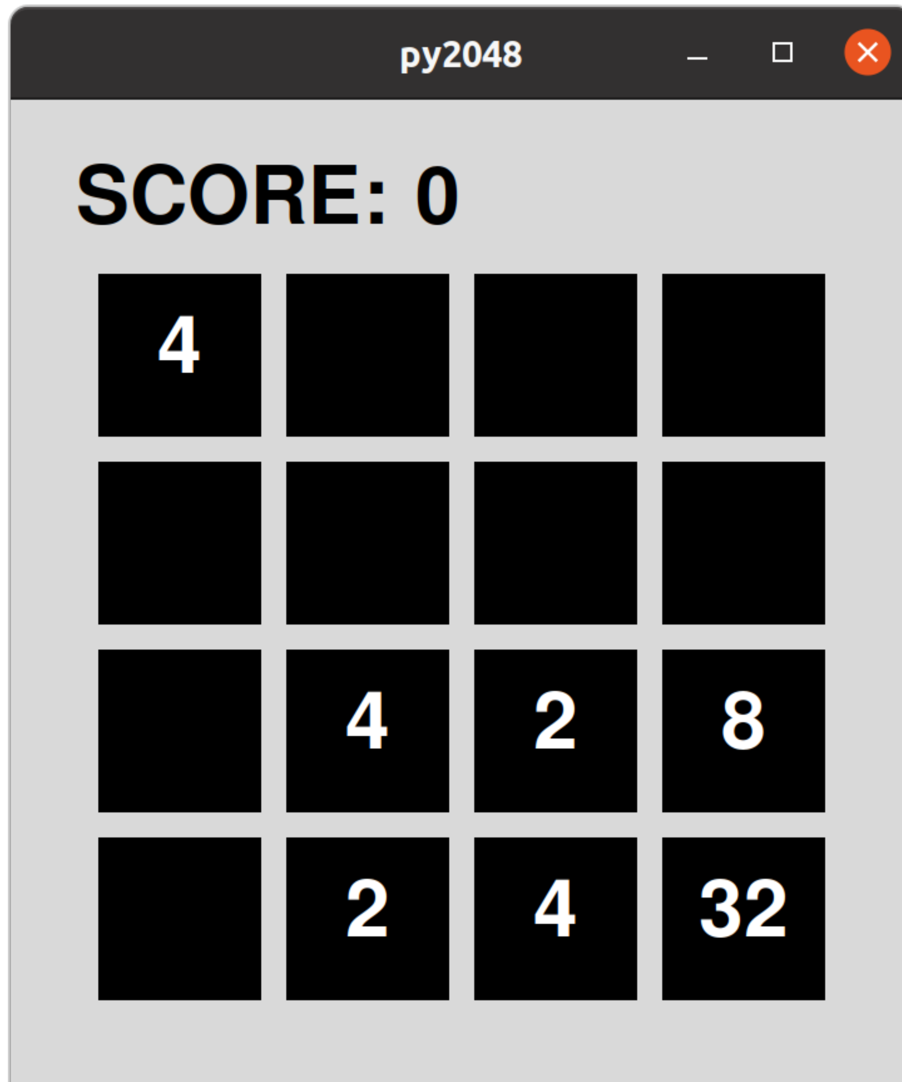
```
        self.moves = {
            "Left": core.move_left,
            "Right": core.move_right,
            "Up": core.move_up,
            "Down": core.move_down,
        }
        for key in self.moves:
            self.bind(f"<{key}>", self.move_handler)
```

We are passing key codes in the form `<Left>` and `<Right>` to represent the keys we want to listen for. In each case, we will trigger the event handler, `self.move_handler` which we will now add.

```
    def move_handler(self, ev):
        self.process_command(ev.keysym)
        self.update()
```

The `move_handler` method calls our `process_command` method, passing the symbol which is extracts from the event data. This updates our grid data as necessary. Then it calls `self.update()` to reflect the changes to the GUI.

The result should be that you can now play the game using the arrow keys!



> Who would have thought it would be so easy?

Updating the score

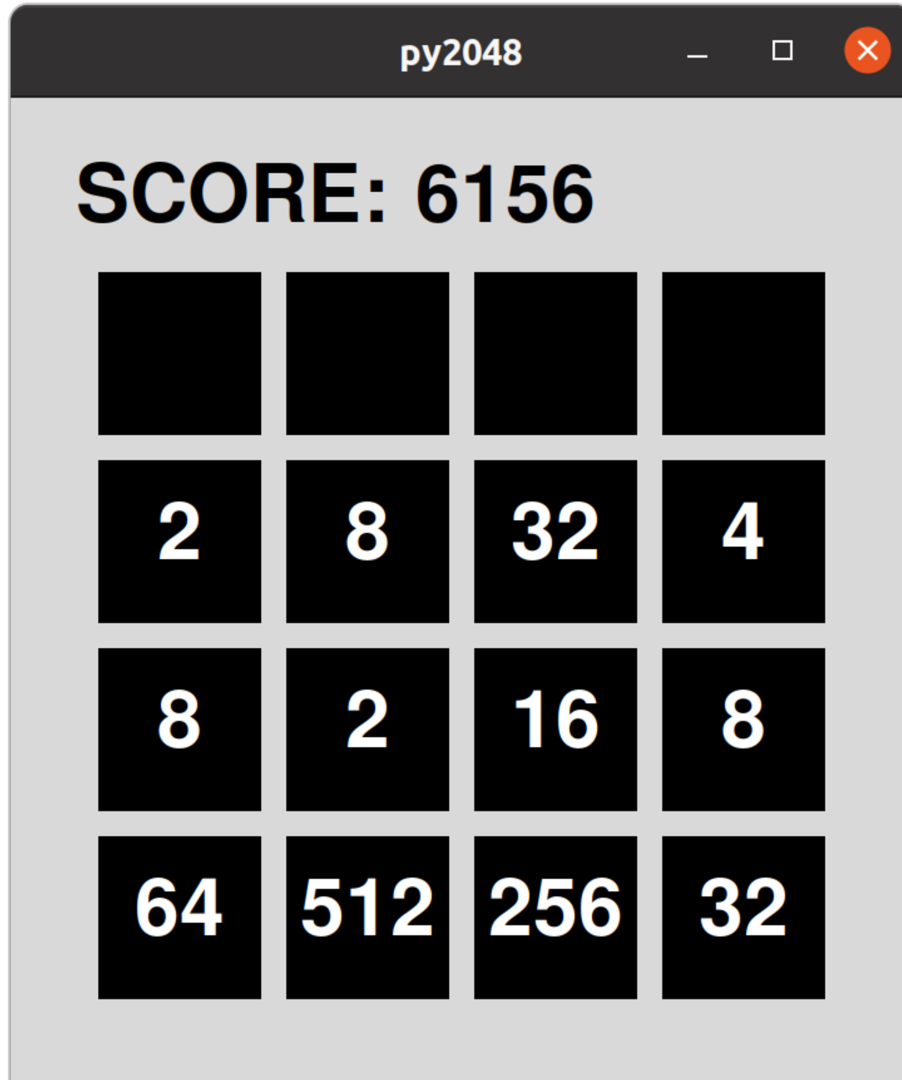
The final move is to calculate and update the score. The first part of this is familiar. We simply copy this from the previous version of the game into the `__init__` method.

```
        self.point_functions = {
            "Left": core.horizontal_points,
            "Right": core.horizontal_points,
            "Up": core.vertical_points,
            "Down": core.vertical_points,
        }
```

We can now simply update the `self.score` property. However, we cannot simply add the integer points to the `tk.IntVar`, we need to call `get()` and `set()` like this.

```
self.score.set(self.score.get() + self.point_functions[command](self.grid))
```

> Add the above line into the correct place in the `process_command` method.



Now you should have a fully working game.... almost.

Challenges

There are a few missing features such as a "game over" message and the ability to restart. Try to implement these features.

Can you match the colour scheme of the online version of the game?