Checkpoints

Checkpoint 1 (Due Monday, Nov 1):

- `parse_data`: Implement the parse data function such that it reads a csv file into a list of lists where each sublist contains integer values. These values will be either 1 or 0 to indicate whether a dead or alive cell is at that location. The data function will read the file given by the second argument of `sys.argv` when the program is run.
- `sum_across_rows`: For the later logic of the game of life we will need to determine how many cells in a given subsection are alive. This function should take a numpy array, sum all of the alive cells, then return the total. Recall the sum/total pattern for this exercise and adapt it for a 2d array.
- `copy_pattern`: This is the trickeiest function in this lab and will likely require the most time. As such we only expect that you will have *started* this function by the end of the first checkpoint. This function takes four arguments: (1) a numpy array of the pattern we want to render as read-in from parse_data and (2) the full grid onto which we want to place our pattern, (3) the amount of room we want between the start of the pattern and the top of the grid, and (4) the same as three but between the pattern and the left side of the grid. You will:
    - (1) Want to check if the size of the grid is large enough to fit the of the pattern plus the padding.
    - (2) Iterate over the pattern one cell at a time and copy the value from that cell to the same location on the grid plus the x and the y padding.

Checkpoint 2 (Due Monday, Nov 8):

- `copy_pattern` : If you did not finish this function in the last checkpoint it should be finished in this checkpoint.
- `init` : Please follow the instructions in the file itself. Here's an exanded template that you might find useful

```
 # Step 1
grid = #use numpy zeros to create a grid of zeroes that is y_dim by x_dim sized
pattern = #pass your parse data function sys.argv[1] to get the file as a np.array()

# Step 2
#Check to make sure that the size of the pattern plus it's padding on both dimensions
#is small enough to fit on the background. If not exit.

# Step 3/4
# Copy the pattern onto the grid using your copy pattern function and return the resulting grid
```

- `update` : This is the heart of operation. Conways game of life is a rule based game where, with each update, we look at the value of each cell in the grid and:
    - Any life cell with fewer than two live neighbors dies, as if by underpopulation. Keep the cell in the new grid we're making 0 and update the color to be about to die.
    - Any live cell with two or three live neighors lives on to the next generation. Update the cell in the new grid to be 1 and update the color to be alive.
    - Any live cell with more than three live neighbors dies, as if by overpopulation. Keep the cell in the new grid and update the color to be about to die.
    - Any dead cell with eactly three live neighbors becomes a live cell, as if by reproduction. Update the cell in the new grid to be 1 and update the color to be alive. Complete this function so that for every update, each cell is examined and updated based on these rules. Here's some example code. You won't be able to copy-paste it because I used different variable names but do read through it to get a sense of how the update works.

```
 #Step 1:
new_grid = # another grid of zeros that's the same dimensions as cur

#Step 2: Iterate over all the cells locations
for r in range(cur.shape[0]):
  for c in range(cur.shape[1]):
      #Step 2.1
      num_neighbors_alive = # Use sum across rows to get the number of cells alive around the cell at curr[r, c].
                            # You will want to use slicing to get the region that's passed to sum_across rows adn
                            # you will have to deal with subtracting 1 in the instance that the current cell is alive

      # Step 2.2
      # This is the core game functionality. You will be updating colors
      if #die by under popuation:
         color = #set the color so that the cell looks like it's about to die
      elif #die by over population
         color = #again, set the color so that the cell looks like it's about to die
      elif #become alive due to voverpopulation
         #set cell in same location on the new grid to 1
         #set the color to make it appear alive
      else:
         #set color to be background

      # Step 2.3
      if #cell at current location is dead:
        # set color to background

      # Step 2.4
      # This is already taken care for you but pygame will draw some stuff

# Step 3: Return the grid you just built
```

- There are some tuples declared at the top of the `life.py` file. Modify these to make a color scheme of your own choosing.
- Create a new pattern called `mylife.csv` and place it in the `data/` directory.

Running the program

The program will be run in one of the two following ways * `python life.py <path/to/pattern.csv>` : In this version the user only provides the file and the padding along the x and y axis should be set to default to 0. * `python life.py <path/to/pattern.csv> <y-pad> <x-pad>` : In this version, the user not only provides the pattern, but also provides the amount of padding they want between the start of the pattern and the y and x axis of the grid.

Further Information:

- Wikipedia Description:
- Intro to Conway's Game of Life
- Numberphile - Inventing Game of Life
- Some cool examples with epic music
- Why is Game of Life Turing Complete
- The Game of Life Inside the Game of Life

Ruberic

- 30 Points - Participation
- 10 Points - `parse_data`
- 10 Points - `sum_across_rows`
- 5 Points - `copy_pattern` started by checkpoint 1
- 10 Points - `copy_pattern` completed by checkpoint 2
- 25 Points - `update`
- 5 Points - Change color scheme
- 5 Points - Produce pattern

Acknowledgements

This repository is a modified version of beltoforion's implementation. It was modified for an introductory computer science course at University Laboratory Highschool at UIUC.