Game of Life in Python

As an exercise in graphics, we are going to implement the Game of Life cellular automata. Later, we will look at modelling scenarios using cellular automata approaches.

The basics of Game of Life are similar to the work we did drawing a square on a `canvas`, so we start with the same packages. We're also going to use `randint` from `random` to generate random integers.

```
from tkinter import *
from tkinter.ttk import *
from time import sleep
from random import randint
```

We'll store the current position in a list and then draw this onto the screen each time we update it. First let's set up the list. We'll use 70 rows (`r`) and 50 columns (`r`). Will a cell size of 20px, this will mean a 1400x1000 canvas.

In our list, we'll store `1` if the cell is alive and `0` if not. Let's start with a list of random values as our initial grid.

```
grid = []
for r in range(50):
    thisrow = []
    for c in range(70):
        thisrow.append(randint(0,1))
    grid.append(thisrow)
```

To see what the result of these loops looks like, run

```
for r in range(len(grid)):
    for c in range(len(grid[0])):
        print(grid[r][c],end="")
    print()
```

Let's draw this onto a window. First we need a window and a `canvas` to draw on.

```
window = Tk()
window.geometry('1400x1000')
window.title("Game of Life")
canvas = Canvas(window, width=1400, height=1000, bg='white')
canvas.pack(anchor=CENTER, expand=True)
```

Now we can use a loop to draw our grid, and an `if` statement to decide whether each cell is alive (coloured) or not (white). Similar to what we did previously, we'll put all this in a `while run` loop that we'll need in a second when we come to animate our canvas. And we use a `window.mainloop()` to keep the window on screen.

```
run = True

while run:
    canvas.delete("all")

    for r in range(len(grid)):
        for c in range(len(grid[0])):
            if grid[r][c] == 1:
                colour = "blue"
            else:
                colour = "white"
            canvas.create_rectangle((20*c,20*r), (20*(c+1), 20*(r+1)), fill=co

    window.update()
    sleep(0.5)

window.mainloop()
```

If you run this, you should be looking at something like this.

If you close the window showing the grid, you may get a `_tkinter.TclError` error. This is because the loop is still trying to update the grid but the `canvas` isn't there any more. To handle this, place this code before the loop.

```
def handler():
    global run
```

```
    run = False
    window.destroy()
window.protocol("WM_DELETE_WINDOW", handler)
```

Now we need to update our loop to implement the Game of Life rules. What happens to a cell in the next loop is based on its status and that of its neighbours in the previous loop, so we can't update our list as we go - we need a second list. After we have drawn our grid, make a new list variable.

```
nextgrid = []
```

A cell on our grid is indexed as `grid[r][c]` where `r` is the row number and `c` is the column number. We are interested in all the cells around this one and want to count how many are currently alive. The neighbouring cells we are interested in are:

- `grid[r-1][c-1]`
- `grid[r-1][c]`
- `grid[r-1][c+1]`
- `grid[r][c-1]`
- `grid[r][c+1]`
- `grid[r+1][c-1]`
- `grid[r+1][c]`
- `grid[r+1][c+1]`

However, we need to be wary of the way lists work in Python.

- If we ask for `grid[-1]`, Python will happily loop around and get the last item on the end of our list. We can deal with this by checking whether `r-1` and `c-1` are `>=0`.
- If we have 70 items in a row as `grid[0][0]` to `grid[0][69]` and we ask for `grid[0][70]` we'll get an `IndexError` because we have asked for an index (`70`) that doesn't exist - we have left the grid. We can deal with this by checking whether `c+1` is less than 70.
- The same happens if we ask for `grid[50][0]` because rows run from `grid[0]` to `grid[49]`. We can deal with this by checking whether `r+1` is less than 50.

Assuming our `r` and `c` are within the grid, we need to add a cell to `nextgrid` which implements the Game of Life rules, that is

1. Any coloured cell with fewer than 2 coloured neighbours is emptied (loneliness);
2. Any coloured cell with more than 3 coloured neighbours is emptied (overcrowding);
3. Any coloured cell with 2 or 3 coloured neighbours remains;
4. Any empty cell with exactly 3 live neighbours is coloured.

We do this by adding up the values of the neighbouring cells as `thissum`. To find the value of the current cell to add to `nextgrid`, we start a new variable `thiscell` with the same value as the current grid, and updating this depending on the value of `thiscell` and the neighbourhood sum `thissum`.

```
for r in range(len(grid)):
    thisrow = []
```

```
    for c in range(len(grid[0])):
        thiscell = grid[r][c]
        if r-1>=0 and c-1>=0 and r+1<50 and c+1<70:
            thissum = grid[r-1][c-1] + grid[r-1][c] + grid[r-1][c+1] + grid[r]

            if thiscell == 1 and (thissum < 2 or thissum > 3):
                thiscell = 0
            elif thissum == 3:
                thiscell = 1
        else:
            thiscell = 0
        thisrow.append(thiscell)
    nextgrid.append(thisrow)
```

Once this process has finished, we have drawn `grid` to the screen and finished using it to calculate the `nextgrid`. Finally, then, we update `grid` to take the values of `nextgrid` ready for the next iteration of the loop.

```
for r in range(len(grid)):
    for c in range(len(grid[0])):
        grid[r][c] = nextgrid[r][c]
```

Now we can do `window.update()` and `sleep(0.5)` and let our loop iterate.

If you run this, you should see a Game of Life simulation running.

Experiment with the starting configuration. Instead of filling the initial grid with random cells, try starting your grid with some of these configurations - what happens?