**Explanation of Our Game of Life Code.**

The first part of the code is where we import the libraries we use in the rest of code. All modules, except matplotlib, are ones that follow along most default python installations, so we just assume the user has them. For matplotlib, however, we use a try/except block to a) import it if they have it and save a variable called has\_module with the value True, which tells the code later to run the game with an animation b) The code fails to import the module, because they don’t have it, but the except block catches this error and moves on. The has\_module variable is then stored as False, so only a text-based display of the game will be shown in the console.

Now follow a long list of functions, which we’ll attempt to explain one by one:

The build\_empty\_grid function takes one parameter, called dimension, and returns a grid (a list of lists) containing only 0s in each index. The way we create this grid is through a for-loop that iterates the number of times specified by the integer contained in the dimension-parameter. For each iteration it adds to the empty grid variable the number 0 multiplied by the dimension. So, if we designate the number 5 as the dimension parameter, the function will iterate 5 times, and in each iteration it will append the number 0 inside a list to the grid-variable of the type list, 5 times. It ends the function by returning the grid, which is now filled with 25 0s, with 5 in each row.

Next comes the build\_first\_grid function which takes two parameters: dimension and number\_of\_alive\_cells. This function returns a grid with the dimension specified in the parameter, containing as many 1s in the grid, as is specified by the second parameter, number\_of\_alive\_cells. We use a for-loop with the range of number\_of\_alive\_cells, and for each iteration it picks two random numbers within the numbers of 0 to the integer contained in dimension, and uses these two random numbers as index references for where it attempts to put a 1. The random numbers are picked using the random.choice() function, which is part of the random-library we imported earlier. If the position in the grid where it attempts to put a 1 is already a 1, then the while loop kicks in. The while loop basically repicks two random numbers until it lands on a point in the grid which isn’t yet a 1. One loop is now done, and the code continues with the next iteration of the for-loop. When the for-loop is done, the function returns the new grid which now has both 0s and 1s in random positions.

Now we’ll skip the transform\_to\_circular\_grid function right to the count\_neighbour\_cells function and refer to it later.

The count\_neighbour\_cells function is used to, for each number in the grid, count how many of its neighbours in a 3x3 square are 1s. It takes the inputs, x, y and some\_transformed\_grid. The inputs x and y are just index references. At the edges it becomes a bit hard to count all of a cell’s neighbours, as some don’t exist. This is where the transform\_to\_circular\_grid function comes in handy.

The transform\_to\_circular\_grid function is, first of all, a bit of a misnomer, as it actually turns the grid into a donut (a torus in technical language), such that all the edges are “connected”. Connecting these edges, is what the function is responsible for. The way it does this is by expanding the grid, essentially. All the sides at the different extremities, are copied and inserted anew in the grid. To further explain part of what it does, the bottom row is copied and inserted at the top. The same is done for the other sides. What one is left with, is the original grid, but one whose dimension in both axes is larger by 2. So, a 5x5 grid becomes a 7x7 grid. But for that 5x5 grid in the middle of the 7x7 grid using the count\_neighbour\_cells essentially allow us to treat the 5x5 grid as a torus. This brings us back to the count\_neighbour\_cells function.

Now, with the transform\_to\_circular\_grid function, counting a cell’s live neighbours becomes a breeze. We just have to adjust the x and y input by adding 1 to each, since we expand the grid when we transform it. Otherwise, the function is pretty intuitive: It looks at a certain cell in the grid, counts its live neighbours and returns the result of that count.

The next\_gen function is how the code can actually process how a population will look in the next generation, according to the rules of the game of life. It only takes the one input of the grid it has to “evolve”. Using the rules, there are two things that matter for the next state of a cell: Whether a cell was alive before or not, and how many live neighbours it had. The function starts by declaring a few variables. It first stores the number of alive cells for the original grid, the transformed grid so it can pass it as an argument when we count the neighbours, and an empty grid which is to become the next generation. It iterates through each cell in the grid using two for-loops, if there’s a 1, one is added to the number of alive cells variable for the last grid variable. For each cell in the grid, the cell at those same index references in the new grid, we convert the 0 to a list containing the value of the previous cell, and the number of live neighbours it had. So, a 1 with 3 neighbours becomes a [1, 3] – a list within a list within a list. Now the whole new grid is filled with a bunch of lists. Next, the function iterates through the entire new grid, again using two for-loops, applying the different rules of the game of life to each list, such that a [1, 3] becomes a 1, and so on for each different rule. This is further elaborated in the code’s comments. The function ends by returning this new grid containing the next generation and an integer representing the number of live cells there was in the previous grid. When returning two values from one function, it is by default in a list form, so we’ll have to keep that in mind for later. The reason we count the number of live cells in this function, is just to make the program a tad more efficient - we’re iterating through the grid anyway, so it seems like a waste to run a whole separate function going through it again.

The function display\_grid was just copied from the website, and we didn’t make any modifications to it, so we won’t be going over it here.

The nb\_times\_list\_in\_list is a function we later use to determine whether a population’s state is infinitely repeating. The function is very simple: it only takes two parameters, namely, a list and a list containing several lists. This function iterates through the list containing more lists, and checks if the list is in it, and if so, how many times. It returns an integer representing the number of times, the list was inside the list of lists.

Now comes the function that is responsible for running the entire game. It takes three inputs: the delay in seconds one wishes between each iteration; some grid it uses as a starting point; and a last parameter that is, by default, set to the Boolean value True. Some variables are declared in the start of the function including an empty list which will store all the previous generations; an empty list which is to contain the grid of the next generation; and an integer set to 1 representing what generation the grid is on. Now comes the part that runs the part that runs the game until either, a) the grid that is about to be showed has already been there 5 times, or b) if the next generation is the same as the current generation. The former condition is tested by the while loop, whereas the latter condition is tested by an if block within the while block using the command “break”, which just continues the code past whatever loop contains the argument. Getting to the while loop, it essentially just displays the generation, printing the number of live cells in it and what generation it is, and initializes the next grid following the delay which we later set to 1 second. This just repeats until the conditions stated previously. This function is also used to determine how many generations it takes until the population has reached a stable state. The function only does this if we set the display parameter to be false, in which case the function does not display the different generations of the grid, but returns an integer representing how many generations it takes before a population/initial grid will go before it reaches a stable state. The way we toggle these parts of the function is just through the use if statements.

Now we’ve essentially made all the functions required to run the game, at least in its text-based form. We ask the user using input() about the various attributes of the grid he desires, and using an if-block combined with a try/except-block to test if the user’s inputs are valid. The code proceeds by building a grid according to the user’s wishes, with inputs the code now has determined are valid. Now the game uses the has\_module variable we previously set to either False or True, based on whether they had the matplotlib module. We’ll start with the case where the user doesn’t have the matplotlib module.

In this case, the program just passes the grid that has been built, as a parameter into our game\_loop function. The first parameter is just the delay between each generation which we just set to 1 second. The delay is done using the .sleep() function from the time library, and is just there to make the game more comprehensible, such that all generations don’t just run and print out all at once.

If the user has the matplotlib module, the code becomes a bit more advanced. All of the documentation for the module is provided here <https://matplotlib.org/3.3.3/contents.html>. More specifically, I had to look up the specific documentation for the functions .imshow() found here, <https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.imshow.html>, and also the documentation for the FuncAnimation found here, <https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.animation.FuncAnimation.html>. The rest of the code relating to the matplotlib is quite common and unsurprising given a basic familiarity with the library.

First the “fig” and “ax” variables are declared, basically initializing the plot; the thing we’ll display our game on. The first “image” we display on the plot is given in the variable “img” which we declare by passing our grid and the colour map we want to use into the function .imshow().

Next comes the function that is responsible for updating the graphic, appropriately called animate(). This function takes three parameters: “i”, which is a standard parameter used for all animations using matplotlib telling what frame the code is on, or, for our purposes, what generation the grid is at. Next parameter is the grid, which is what we’ll update, and the last is the current “image” on the graphic.

The function starts by generating the next grid using the next\_gen function. We don’t care about how many cells were alive in the last grid, so we only store the first value the function returns. Next, we update the data used for the ax.imshow() function, used in the “img” variable. This line is essentially the part that updates the graphic. Lastly, the grid is set to be equal to the new\_grid such that the function works, when the FuncAnimation() function repeats it. This also ensures that our grid actually progresses and doesn’t just show the same image all the time. We return the img variable which has just been given new data, and therefore has been updated, so the function, FuncAnimation(), can load it.

The reason behind the display parameter in the game\_loop function should now become apparent. For the animation, so it doesn’t just run infinitely consuming excessive computing power, we set the amount of frames the animation should run to exactly how many generations it takes before a population reaches a stable/infinitely repeating state. This number is found by running the game\_loop with the grid and the parameter display set to false, so it doesn’t output in the console unnecessarily, and so we can get the grid’s “lifespan”.

Now comes the hard function, namely the FuncAnimation from the matplotlibrary. As the first parameter, we pass the variable that contains the graphic in which the animation should run. This is the variable “fig”. Next is the function that updates the graphic which is animate(); one I’ve already described in some detail above. The fargs parameter is kind of a catch-all parameter. This input just contains the additional variables, beside i, we have to pass in the animate() function. The interval is just the amount of milliseconds between each frame, which we’ve arbitrarily set to 100ms. The number in frames just sets how many frames the animation should run for, which we set to the variable we found previously containing the number of generations a population will run for. The repeat parameter we’ve set to False, just asks whether once the animation is done, if it should repeat. The final line of the if-block is just plt.show(), and this is the line that tells our code to actually display the graphic on the screen.

(I’ve also added a gif file showing what the game looks like with the module in the assignment)