Generate World

Implementation of the Game of Life began with designing the generate_world function. This function creates the 2d grid of 1s and 0s using a list of lists. Each list within the world list is a row in the grid and each element is a 1 or 0. Thus, the number of columns would be determined by looking at the length of a list in world. If an element is 1 that represents a cell that is currently alive and if the element is a 0 that represents a dead cell.

There are two types of worlds that can created: empty or random. The type is determined by the user via a command line argument. The empty world is just an M x N list of 0s where M is the columns and N is the rows. The columns and rows are also determined by the user via command line arguments. Similarly, the random world is also an M x N list of zeros. However, a random coordinate is generated by picking two random numbers in the range 0 to M and 0 to N. The element in the world that corresponds to this coordinate is assigned a 1. A list of the coordinates that have already been assigned a 1 is kept ensuring that 10% of the area (M x N) will be randomly assigned a 1 without repetition. If the user supplies dimensions that will not be able to support at least 10% living cells, the program will complain and exit promptly. Once the world is created, empty or random, the function will return the world list.

Update Frame

The core of the program was built in the update_frame function which examines the current state of the game and determines what the next state will be and returns the plot image of what that should look like. The first step to this function is to create of copy of the world which will be manipulated while the original world will be list that will be examined. To determine the next state, the function looks at every element in the grid and for each element looks around at its neighbors. For each neighbor, if that neighbor is alive, increment the count of alive neighbors (which will be used later). The point of interest here is the modulo operator that enabled the wrap around effect. More information is included in the comments in lines 85-88. Accessing the elements can be seen here:

```
>>> l = [1,2,3]
>>> l[0 % len(l)]
1
>>> l[1 % len(l)]
2
>>> l[2 % len(l)]
3
>>> l[3 % len(l)]
1
```

When the iterator reaches len(l) + 1 the element accessed is the first element of the list. This concept was applied to the 2d grid list of lists. Finally, once the alive neighbors are counted, the logic provided in the game rules are applied to the current cell—killing it, bringing it to life, or neither (assigning the current element to 0, 1, or do nothing, respectively). The only caveat was that the current cell is counted as a neighbor (yes, perhaps misleading code). However, this was accounted for by adjusting the rules to the living neighbors (i.e. if a cell is alive and has less than 2 living neighbors the code will check for less than 3 living neighbors). The last step of the update_frame function is to apply the modified world to world.

Blit

The blit function places a pattern into the world grid at position x, y. x, y is the top left most position of the pattern/first list, first element of pattern. The code is pretty straight forward: loop through each element of pattern and at position x,y of world change the current x, y element to whatever current pattern element the loop is on. Once the loop reaches the end of the row of the pattern, move on to the next row in pattern. If (x, y) in world doesn't exist the IndexError will be caught and alert the user that the pattern might be either too large to fit in the currently specified bounds or the x, y position given might not exist in the currently specified bounds.

Get Command Line Options

One extra command line arg was added to the arguments given in the skeleton code. I thought having an easy way to quickly add a blit without modifying the actual code would be useful and easier to test different patterns. I wanted the user to be able to specify the name of the pattern and provide a coordinate as one argument. I initially thought it would make sense to do a group_argument, but having the argument be a mixed type (string and two integers) complicated things a bit. I decided to add a normal argument to the parser with 3 required args and my own custom action that will validate the input and modify the namespace accordingly. The action is called ValidateBlit which is a class that inherits the Action method from the argparse module and modifies the __call__ method. The method first assigns the first input after -b to the pattern variable, and the next two to x and y respectively. Then the method validates the input by checking if the pattern matches any of the supported patterns and casts x and y to ints. If the pattern isn't in the list of patterns or x or y cannot be cast to an int, the program will complain to the user. To access the pattern and coordinates separately, I created a named tuple from the collections module called Blit. Blit is an object disguised as a tuple containing a pattern and coordinate. Here's an example of Blit:

```
F:\Users\Daniel\OneDrive - drexel.edu\ECEC 301>python gameoflife.py -b gosper_gun 10 10
Blit(pattern='gosper_gun', coord=[10, 10])
```

This way, each item in Blit can be accessed like an object via Blit.pattern or Blit.coord. Finally, the attributes pattern and coord are set to opts.blit using the settattr method (setattr(object, name, value)). opts now looks like this:

```
F:\Users\Daniel\OneDrive - drexel.edu\ECEC 301>python gameoflife.py -b gosper_gun 10 10
Namespace(blit=Blit(pattern='gosper_gun', coord=[10, 10]), cols=50, framedelay=100, rows=50, world_type='empty')
```

and to access the pattern, for example:

```
opts.blit.pattern
```

I also added a custom metavar to the argument to provide clarity to the help message:

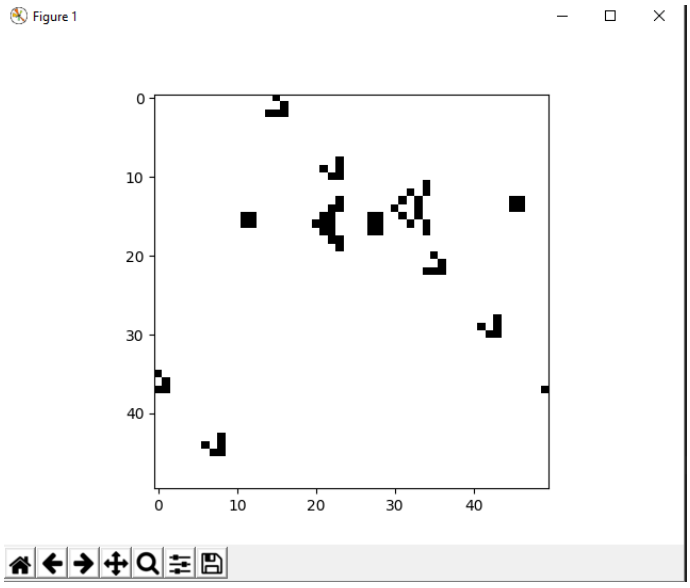
```
F:\Users\Daniel\OneDrive - drexel.edu\ECEC 301>python gameoflife.py -h
usage: gameoflife.py [-h] [-r ROWS] [-c COLS] [-w {random,empty}]
                     [-d FRAMEDELAY] [-b BLIT X Y]

optional arguments:
  -h, --help            show this help message and exit
  -r ROWS, --rows ROWS  set # of rows in the world
  -c COLS, --columns COLS
                        set # of columns in the world
  -w {random,empty}, --world {random,empty}
                        type of world to generate
  -d FRAMEDELAY, --framedelay FRAMEDELAY
                        time (in milliseconds) between frames
  -b BLIT X Y, --blit BLIT X Y
                        blit pattern followed by x y coordinate
```

Main

The modifications to main are due to putting the new command lines options into practice. Now, the blit function is only called if the user specifies a blit and the arguments supplied to the blit function are now dynamically created. The pattern argument is accessed via opts.blit.pattern by using the gettattr method. Using patterns.opts.blit.pattern would not work because that would be interpreted as accessing the pattern opts from the pattern module. The x coordinate is contained in the first element of the list opts.blit.coord list and the y coordinate is the second element.

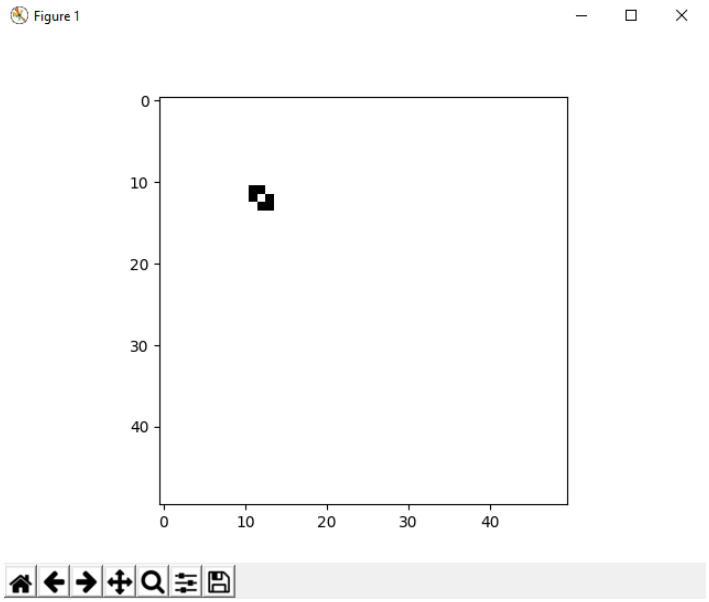
Here is an example of adding the gosper_gun to position (10, 10) and running for a few seconds:

```
F:\Users\Daniel\OneDrive - drexel.edu\ECEC 301>python gameoflife.py -b gosper_gun 10 10
Conway's Game of Life
=====================
   World Size: 50 x 50
   World Type: empty
   Frame Delay: 100 (ms)
   Blit Pattern: gosper_gun
```

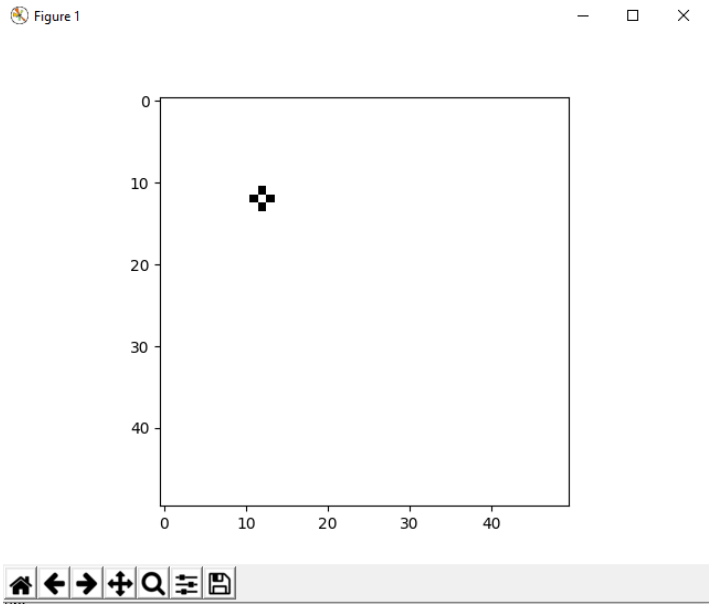


I also added the ship and tub still life patterns along with the beacon pattern for fun.

Ship:



Tub:



I'll leave the reader to check out the beacon oscillator.