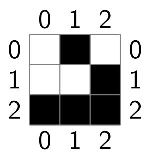
Exercises

Set 6

DM857 Introduction to Programming
DS830 Introduction to Programming

1 Small Simulation Projects

In this lab you will be asked to implement a series of zero-player games played on two dimensional grids of squares each of which can be presented as in one of two possible states. Below is a visualisation of one of such grids.



A way to encode 2D data in Python is using nested lists and treating each inner list as a row of the grid. For instance, the list below encodes the grid above.

```
[[0, 1, 0],
 [0, 0, 1],
 [1, 1, 1]]
```

Visualising 2D scalar data

Module `grid_visualisers` (available on itslearning together with its documentation) contains a number of classes for display 2D scalar data as a pseudocolors image on a 2D grid. Of these, `BinaryVisualiser` is for displaying binary data. The snippet blow illustrates its use to display some static data.

```
grid = [[0, 1, 0],             # some 2D binary data
        [0, 0, 1],
        [1, 1, 1]]
vis = BinaryVisualiser(grid) # create a visualiser for this data
vis.wait_close()             # wait until the window is closed by the user
```

The constructor of `BinaryVisualiser` takes as an argument the data to be displayed and raises a dedicated window (it takes a number of optional arguments to customise the display, see the documentation). Then, the instance method `wait_close(self)` is invoked to suspend the current execution until and wait for the visualiser window to be closed by the user. This class can also be used to animate a visualisation by providing it with fresh data using `set_data(self,data)` or `update(self,data,pause=0.1)`. The only difference is that the second pauses for the given number of seconds (default is `0.1` seconds). Both methods assume that the new data is of the same size of the old one (you can check the size of your data using the function `size` in the same module). Finally, the instance method `is_open(self)` returns whether the window managed by the visualiser is open or closed. The following function displays a grid with a black square (the dot) running across it horizontally until the window is closed.

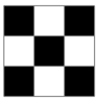
```
def running_dot(height:int,width:int)->None:
    '''Displays a window with a dot running across it.'''
    grid = [[0] * width  for i in range(height)] # a grid of all zeros
    x = width // 2
    y = height // 2
    grid[y][x] = 1                          # now with a dot at (x,y)
    vis = BinaryVisualiser(grid)            # visualiser
    while vis.is_open():
        grid[y][x] = 0                      # erase the current dot
        x = (x + 1) % width                 # move its coordinates right by 1 square
        grid[y][x] = 1                      # set the dot to its new position
        vis.update(grid,interval=0.2)       # update the visualiser and pause for 0.2seconds
```

Module `grid_visualisers` requires the package `matplotlib`[1] (available through PIP).

For some many of the exercises in this set you will have to use module `random`[2] which is included with the standard Python installation.

1.1 Exercises

1. Write the following functions for generating grids and test them with `BinaryVisualiser`.
   (a) `zeros(height,width)` that returns a `width` by `height` grid filled with `0`s.
   (b) `checkers(height,width)` that returns a `width` by `height` grid filled with alternating `0`s and `1`s such that it displays as a chessboard.



   (c) `uniform_noise(height,width,p=0.5)` that returns a `width` by `height` binary grid where each cell is randomly assigned `0` with probability `p` and `1` with probability `1-p`. (Hint: see `random.choice()`.)
   (d) `uniform_ones(height,width,ones)` that returns a `width` by `height` binary grid filled with exactly "ones" cells equal to `1` in randomly selected positions. (Hint: see `random.sample()`.)
   (e) `flip(grid)` takes a grid and turns `0`s into `1`s and vice versa.
   (f) `random_flip(grid,p=0.5)` takes a grid and turns `0`s into `1`s and vice versa with probability `p`.
   (g) `paste(source,target,offset_y,offset_x)` that copies the content of `source` into `target` applying the given offset (can be negative) and ignoring values that would end up outside the target grid. For instance, the value in `source[y][x]` is copied to `target[y+offset_y][x+offset_x]`, if there is such an cell in the target grid.
2. Using `BinaryVisualiser`, write the following functions.
   (a) `tv_noise(height,width,p=0.5,interval=0.1)` that displays uniform noise updated at regular intervals (see `uniform_noise`).
   (b) `jumpers(height,width,dots,interval=0.1)` that displays dots in random positions changed at regular intervals (see `uniform_ones`).

[1] `https://matplotlib.org/`
[2] https://docs.python.org/3/library/random.html

3. Conway's Game of Life is a zero-player game played on a (infinite) grid of squares each of which is in one of two possible states, inhabited or uninhabited (or live and dead, respectively). The state evolution of the grid happens in discrete time steps where the new status of each square is determined uniquely by its current status and that of his neighbours (squares horizontally, vertically, and diagonally adjacent to it) using the rules below.

   - Any inhabited square with less than two or more than three inhabited neighbouring squares becomes uninhabited.
   - Any uninhabited square with three inhabited neighbouring squares becomes inhabited.
   - In other case, a square maintains its current state.

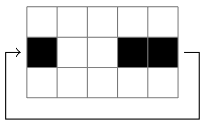
   In the remaining of this exercise you will implement a variant of this game where the infinite grid is replaced by a toroid (the left and right sides of the grid are "glued" together and likewise, the top and bottom sides). Thus the grid is finite in size while every square has 8 neighbours like on an infinite grid.

   (a) Define a function `next_state(current_state)` that takes a binary grid and returns a grid that represents the next state in the game.

   (b) Define a function `game_of_life(initial_state, interval = 0.2)` that displays an animation of the game being played starting from the given initial state. Test it using the functions defined during Exercise 1.

   (c) Define a function `noisy_game_of_life(initial_state, p=0.1, interval = 0.2)` behaves like `game_of_life` and in addition allows for each square to randomly flip its value with probability `p`.

4. In this exercise you will implement some zero-player variations of the snake game using `BinaryVisualiser`. In all exercises, snakes move in a "warp-around" space, a toroid, so a snakes that moves past one side of the grid reappears on the opposite side.



   (a) Identify the status of the simulation and reflect on what is the minimal information needed to represent its status, how to properly store and retrieve all the information (head, body and tail of one or multiple snakes).

   (b) Define a function `snake(height,width,length,interval=0.1)` that displays an animation of a single "snake" moving on a grid (big enough to host a snake of the given length). The movement will be implemented as one block appearing on one of the free spots near the head, while the tail block will disappear.

   At every interval, the snake randomly selects a direction free of obstacles and moves by one cell in that direction (left, right, up or down).

   The animation terminates when the snake finds itself at a dead end (it is no longer possible to pick a direction).

   (c) Define a function `snakes(height,width,lengths,interval=0.1)` that behaves like function `snake` but takes a list of lengths instead of a single one and displays a snake for each length.

(d) Modify your implementation of `snake` to bias the selection of the movement direction to make turns less likely. (Hint: `random.choice`)

(e) Define a function `hungry_snake(height,width,interval=0.1)` that behaves like function `snake` except that now the snake can eat food that randomly appears on the grid. When the snake head ends on a square with food it eats it and the snake grows by one square.

(f) Modify your implementation of `hungry_snake` to bias the selection of the movement direction to make the snake prioritise directions that bring it closer to a square with food. (Hint: computing the distance of two points in on a 2D toroid is very similar to computing the distance of two points on a closed line, like a circle.)

(g) Define a function `cannibal_snakes(height,width,lengths,interval=0.1)` that behaves like function `snakes` except that now snakes can eat each other. When a snake "hits" the body of another it eats the hit part and grows by one in length; the hit snake dies leaving behind its body for the other snakes to eat. The animation continues until there only one surviving snake.