Activities: Modules 29 - 30

In all files that you create, create test cases that run when you reload the file, (run / F5). You create an if statement at the bottom of the file that starts out with

```
if __name__ == '__main__':
    print('running tests')
```

The code in the if statement's body will only execute if the file itself is run, not when it is included. Since you run a file whenever you want its changes to become permanent, this tests all of the use cases that you written inside the if-statement, thus guaranteeing that you notice problems early as long as you create sufficient test coverage.

(1) Inside a file "geometry.py", create a class Geometry with a string dunder. The components of an object are just width and height.

(2) Inside a file "model.py", create a class Location. A Location has an x and a y coordinate. Create a string dunder and an equality dunder. Create a class method `def generate_random_locality(geometry)` that generates a random location.

(3) Inside the same file "model.py", create a class Direction. It starts out with

```
class Direction:
    codes = {'N', 'W', 'E', 'S', 'NW', 'NE', 'SW', 'SE', '0'}
    def __init__(self, code):
        if code in Direction.codes:
            self.code = code
        else:
            raise ValueError
```

(4) Add to the class Direction a string dunder.

(5) Add to Direction a class method `move_towards(mover, aim)` that determines the direction of a robot moving towards the avatar. For the time being, let robots only move left, right, up, and down. (Later we will change that to allow diagonal moves).

(6) Add to location an `update_location(self, direction, geometry)` method that updates the location according to the direction. Insure that the location never leaves the playing field given by geometry. Here is a code snipped from me:

```
elif direction.code == 'SE':
    self.y = min(geometry.height-1, self.y+1)
    self.x = min(geometry.width-1, self.x+1)
elif direction.code == 'SW':
    self.y = min(geometry.height-1, self.y+1)
    self.x = max(0, self.x-1)
```

(7) Inside "model.py" create a class Avatar. The Avatar class has a constructor, a string dunder, and an instance method move with parameters a direction and a geometry.

(8) Inside "model.py, create a class Robot. The robot class has a `move(self, avatar, geometry)` instance method that first calculates the direction towards the avatar and then moves the robot.

(9) Create a class Heap. Heaps do not move.

(10) Create a class Model that is instantiated by a geometry and a number of robots. The class has a geometry, a list of heaps, an avatar, a list of robots. Make sure to generate enough robots with random locations and an avatar with a random location, but such that no two items have the same location.

(11) Create a class view in a new file "view.py". This module needs to have access to model and geometry. The class View only contains two class methods, namely `draw_screen(my_model)` and `clear_screan()`. The code for the first is slightly tricky,

```
class View:
    def draw_screen(my_model):
        array = [[" " for i in range(my_model.geometry.width)] for
                     j in range(my_model.geometry.height)]
        for h in my_model.heaps:
            y, x = h.location.get()
            array[x][y] = 'H'
        for r in my_model.robots:
            y, x = r.location.get()
            try:
                array[x][y] = 'R'
            except:
                print(x, y, r)
        y, x = my_model.avatar.location.get()
        array[x][y] = 'A'
        print((my_model.geometry.width+2)*'-')
        for y in range(my_model.geometry.height):
            print("|", end = '')
            print ("".join(array[y]), end='')
            print('|')
        print((my_model.geometry.width+2)*'-')
```

since you need to figure out how to address in a two-dimensional list.

(12) Create a module "game.py" that runs the game, interacting with view and model. It has only one function, called `run_game()`. The function initializes a geometry and a model with a certain number of robots. It then enters into an infinite loop, where it clears the screen, displays the playing field and asks for input. The input is passed to the model for update. The update function will return a code if the avatar is dead (because it shared space with a robot or a heap) or if all robots are dead. At this point, you have an almost functioning game.