CSCI 1100 — Computer Science 1
Homework 5
Wandering trainer

Overview

This homework is worth **110 points** total toward your overall homework grade, and is due Thursday, October 20, 2016 at 11:59:59 pm. You will write and debug two versions of one program, with the first version solving a more restricted problem than the second.

See the HW 3 description for a discussion of academic integrity issues. Also review the grading criteria in HW 1 and Lecture 11. For the rest of the semester, these criteria will be a significant part of your grade. You may have noticed us beginning this process as part of the HW 3 grading.

The homework submission server URL is below for your convenience:

`https://submitty.cs.rpi.edu//index.php?semester=f16&course=csci1100`

The two versions of your program must be named.

```
hw5Part1.py
hw5Part2.py
```

This assignment builds heavily on material from HW 3, HW 4 and the random walk example from Lectures 11 and 12. Feel free to make use of your code from these assignments and the code we provided, but note that there is **one change in notation** from HW 3 and HW 4. Previously, we used `(x, y)` coordinates to mark positions on our grid. For this assignment, we will be using `(row, column)` coordinates instead. If you think about how we are storing and accessing our two dimensional lists, you should understand the change. If you are unsure how this will affect your program, work it out on some graph paper or come talk to a TA or instructor.

Part 1: Wandering Trainer

A pokemon trainer is placed in the middle of a grid that is M rows tall and N columns wide. These values are read into the program by asking the user. The user must also be asked for a probability p that must be great than 0.0 and less than 1.0, but will generally be relatively small. The reason for the probability is explained below. You may assume all input is correct. The upper left corner of the grid is location $(0, 0)$, while the bottom right corner is location $(M-1, N-1)$. The trainer starts out at location $(M//2, N//2)$. At each turn, the trainer will take a step in a random direction and look for a pokemon.

The program must simulate random movements of the trainer. As in HW 4, Part 3, the trainer can can only move in a straight line to the North (decreasing row), South (increasing row), East (increasing column), and West (decreasing column) one step per turn. After taking a step, the trainer will have a p probability of finding a pokemon on that spot. To manage this operations, we will use:

```
random.randint(0, 3)
random.random()
```

Each time it is called `random.randint(0, 3)` returns one of `0, 1, 2 or 3`. If the value return is:

- 0 - The trainer moves North one step
- 1 - The trainer moves South one step
- 2 - The trainer moves East one step
- 3 - The trainer moves West one step

Once the trainer moves, random.random() is called to generate a floating number between 0 and 1. If the value returned is less than *p* the trainer finds a pokemon on the current spot. Otherwise, no pokemon is found. Order is important, in every turn make sure you generate the direction **(randint(0, 3))** first and then the capture probability, **(random)**! The trainer should continue to make these random steps until 250 time steps (turns) have passed. As in HW 4, the trainer cannot go past the boundaries of the grid. It the trainer tries to move past the boundary, for example, by trying to go North when the *row* coordinate is 0, the trainer just stays at the same location and searches again for a pokemon.

To solve part 1 your program must report the position of the trainer and the number of pokemon collected. Once the simulation ends you must output the final position and the total number of pokemon collected.

Important Details:

1. In order for everyone to have the same output we must *seed* the random number generator. After you read the user input, but before the first time you call one of the `random` functions, you must include the following code

   ```
       seed_value = 10*M + N
       random.seed(seed_value)
   ```

   The seed ensures that the random number generator gives the same sequence of values for the random calls. For us, that means that we can compare your output to our output and expect to get the same sequence of movements and captures. To see more interesting behavior when testing your program, you might want to comment out the call to the `seed` function; just be sure to put it back in before you submit!

2. You **MUST** write and use

   ```
   def move_trainer( position, bounds, prob):
       '''
       position is the current (row, col) position of the trainer, passed as a tuple

       bounds is the max (rows, cols) on the grid, again as a tuple

       prob is the probability p that a pokemon will be found at the current position
       '''
   ```

   which moves the trainer for one time unit, checking for a pokemon at the resulting location. The function must return a tuple containing in order, the new position `(row, col)` and 1 if a pokemon was found this turn, or 0 otherwise. Your main code must then call this function in some kind of loop and use the results to track the pokemon and decide what to output. Do **not** call the `random.seed` function inside the `move_trainer` function.

3. In `move_trainer` you **must** make the calls to `random.randint(0, 3)` and `random.random()` in the correct order and exactly once per turn. Test the values returned to decide what move to make and if a pokemon is captured.

Additional Notes:

1. Turns should run from 1 to 250
2. In each turn:
   (a) Take a step and check for a caught pokemon using `move_trainer`
   (b) Every 20 turns print the trainer's position and number of pokemon caught since the last report
3. At the end, output the final number of time steps, the final position, and the total number of pokemon found. See examples below for details.

Example 1:

```
Enter the integer number of rows => 40
40
Enter the integer number of cols => 46
46
Enter the probability of finding a pokemon (<= 1.0) => 0.15
0.15
Time step 20: position (24, 23) pokemon caught since the last report 2
Time step 40: position (28, 15) pokemon caught since the last report 4
Time step 60: position (32, 17) pokemon caught since the last report 2
Time step 80: position (30, 21) pokemon caught since the last report 3
Time step 100: position (26, 17) pokemon caught since the last report 6
Time step 120: position (27, 18) pokemon caught since the last report 4
Time step 140: position (29, 20) pokemon caught since the last report 4
Time step 160: position (33, 18) pokemon caught since the last report 3
Time step 180: position (31, 22) pokemon caught since the last report 1
Time step 200: position (34, 25) pokemon caught since the last report 2
Time step 220: position (34, 23) pokemon caught since the last report 6
Time step 240: position (35, 26) pokemon caught since the last report 3
After 250 time steps the trainer ended at position (36, 25) with 40 pokemon.
```

Example 2:

```
Enter the integer number of rows => 100
100
Enter the integer number of cols => 300
300
Enter the probability of finding a pokemon (<= 1.0) => 0
0.0
Time step 20: position (51, 155) pokemon caught since the last report 0
Time step 40: position (48, 154) pokemon caught since the last report 0
Time step 60: position (56, 156) pokemon caught since the last report 0
Time step 80: position (55, 149) pokemon caught since the last report 0
Time step 100: position (53, 151) pokemon caught since the last report 0
Time step 120: position (54, 152) pokemon caught since the last report 0
Time step 140: position (57, 157) pokemon caught since the last report 0
Time step 160: position (53, 155) pokemon caught since the last report 0
Time step 180: position (51, 145) pokemon caught since the last report 0
Time step 200: position (54, 144) pokemon caught since the last report 0
Time step 220: position (57, 145) pokemon caught since the last report 0
Time step 240: position (58, 142) pokemon caught since the last report 0
After 250 time steps the trainer ended at position (58, 142) with 0 pokemon.
```

Example 3:

```
Enter the integer number of rows => 10
10
Enter the integer number of cols => 30
30
Enter the probability of finding a pokemon (<= 1.0) => 1
1.0
Time step 20: position (4, 14) pokemon caught since the last report 20
Time step 40: position (5, 9) pokemon caught since the last report 20
Time step 60: position (5, 11) pokemon caught since the last report 20
Time step 80: position (6, 10) pokemon caught since the last report 20
Time step 100: position (9, 1) pokemon caught since the last report 20
Time step 120: position (9, 0) pokemon caught since the last report 20
Time step 140: position (9, 2) pokemon caught since the last report 20
Time step 160: position (5, 3) pokemon caught since the last report 20
Time step 180: position (9, 6) pokemon caught since the last report 20
Time step 200: position (9, 3) pokemon caught since the last report 20
Time step 220: position (6, 8) pokemon caught since the last report 20
Time step 240: position (1, 9) pokemon caught since the last report 20
After 250 time steps the trainer ended at position (0, 8) with 250 pokemon.
```

Part 2: Gathering Data About the Wandering Trainer

Thus far, we have only considered a single case of running the pokemon simulation, but simulations are typically run over and over again and statistics are gathered about the results of the runs. So, in this second part of the assignment your program will need to repeat the simulation a user-specified number of times, and output several summary statistics:

1. An output of the grid showing the number of times that a pokemon was caught in each position.
2. The minimum and maximum number of pokemon caught in a single simulation and the simulation number (from 1 to the number of simulations run) at which these occurred.
3. The total number of pokemon caught across all simulations.
4. The maximum number of pokemon caught at *any* position as an integer and as a percentage of the total pokemon caught.
5. The minimum number of pokemon caught at *any* position as an integer and as a percentage of the total pokemon caught.

A Counting Grid: You must create a list of lists of counts for the number of times the trainer catches a pokemon in each cell of the grid. Here are two examples to help you. The first example shows an easy way to initialize the grid to have `M` rows and `N` columns:

```
    count_grid = []
    for i in range(M):
        count_grid.append( [0]*N )
```

The second example illustrates counting the number of occurrences of the numbers 0 through 9 using the random number generator (without using the `seed` function):

```
import random

num_trials = 2500
counts = [0]*10
for i in range(num_trials):
    digit = random.randint(0,9)
    counts[digit] += 1

print('Occurrences and percentages:')
for i in range(10):
    print("{:1d}: {:4d} {:4.1f}".format(i, counts[i], 100.0*counts[i]/num_trials))
```

Important details:

- You must **make** use of the `move_trainer` function from part 1. You can copy and paste it into your `hw5Part2.py` file.
- Next you **must** write and test a function called

  ```
  def run_one_simulation( grid, prob ):
      '''
      runs the simulation and keeps track of the number of pokemon caught
      on each space in the grid. prob is the probability a pokemon will be caught
      at each turn
      '''
  ```

  that starts the trainer in the center of the grid, and runs one full simulation of the trainer until it reaches the maximum number of time steps. At the end, it returns the number of pokemon caught during the simulation. Each time step has two phases. In the first phase, the trainer moves and during the second phase the trainer tries to catch a pokemon. This is exactly your `move_trainer` function from Part 1 and `run_one_simulation` should call `move_trainer` once per time step to move the trainer and capture pokemon. The results should be tracked and recorded in `grid`. For instance, suppose that `move_trainer` returns (row, col) as the trainer position and 1 as the number of pokemon caught that turn. Your code should increment `grid_count[row][col]` by one to account for the captured pokemon. You **must** have `run_one_simulation` in your program, but you may change its parameters as you wish. For example, you could pass `M` and `N` — the number of rows and columns — although you do not need to because you can compute the values from the `grid_count` list of lists.
- Do **not** call `random.seed` from inside `run_one_simulation`. It will cause the random number

generator to start over for each simulation and so your results will be the same for each simulation.

- Each time you call `run_one_simulation` the trainer should start in the center of the grid.
- We *suggest* that you write a function to extract and print the statistics of the grid in order to avoid cluttering the code in the main body of the program.
- At the end of the simulation, you may want to have `run_one_simulation` return the number of pokemon caught during that run. This will help with tracking the minimum and maximum number of pokemon caught per simulation.

Finally, here is an example that illustrates the output and formatting we are expecting.

```
Enter the integer number of rows => 18
18
Enter the integer number of cols => 16
16
Enter the probability of finding a pokemon (<= 1.0) => 0.4
0.4
Enter the number of simulations to run => 100
100

  20  21  18  17  21  28  21  34  26  22  27  32  20  21  26  37
  15  23  18  19  17  19  21  39  24  21  20  37  38  41  40  47
  24  29  27  19   8  25  24  32  23  37  38  36  30  30  36  33
   9  18  18  20  19  33  34  26  19  30  37  32  31  30  23  41
  12  18  14  24  31  38  31  22  22  28  49  36  31  28  36  36
  20  24  20  29  30  36  34  33  32  42  47  43  24  34  28  36
  20  19  18  26  29  19  32  44  40  37  37  46  35  28  33  46
  20  34  29  16  23  27  41  45  59  50  58  35  31  32  28  43
  23  26  20  31  25  29  43  60  83  61  47  36  36  28  25  34
  19  30  31  23  23  30  46  59  73  63  57  39  28  36  24  27
  25  37  19  22  34  31  39  64  71  66  61  36  29  35  37  19
  46  39  30  25  36  35  36  44  46  57  38  45  36  32  33  28
  35  38  27  30  35  51  47  47  48  54  54  58  42  41  59  45
  43  28  32  28  35  38  48  38  56  40  58  56  40  27  44  47
  27  28  32  34  45  54  54  38  39  30  32  35  43  45  50  46
  31  35  37  28  33  39  42  42  31  35  29  48  40  52  41  50
  31  29  36  34  35  36  36  27  22  26  39  44  40  40  48  49
  34  35  33  25  27  29  28  25  35  26  46  52  43  41  47  44

Total pokemon caught is 9951
Minimum pokemon caught was 81 in simulation 61
Maximum pokemon caught was 125 in simulation 52
Max number of pokemon caught on a space is 83 which was 0.83% of all pokemon caught
Min number of pokemon caught on a space is 8 which was 0.08% of all pokemon caught
```

And one more showing a potential failure case:

```
Enter the integer number of rows => 15
15
Enter the integer number of cols => 12
12
```

```
Enter the probability of finding a pokemon (<= 1.0) => 0
0.0
Enter the number of simulations to run => 10
10

   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0

Total pokemon caught is 0
Minimum pokemon caught was 0 in simulation 1
Maximum pokemon caught was 0 in simulation 1
Max number of pokemon caught on a space is 0
Min number of pokemon caught on a space is 0
```

In terms of formatting, each of the output values in the grid is formatted with `{:4d}`. You can either generate each row as a string and print it out, or you can use the `sep=` and `end=` values of the print statement to control the formatting.

Some notes on debugging

This is your most complex homework to date, so here are some suggestions for debugging:

- We are asking you to use large grids, 250 iterations per simulation, and for Part 2 a large number of iterations. This is too much to debug easily. Start small
  - Use Part 1 to get `move_trainer` working and to test it thoroughly. Step through it in the debugger, make sure all paths are selected and make sure that the positions of the trainer after each move are right. It should only take you a few times through the loop.
  - Use a small grid, eg. $5x6$ to check the behavior of the trainer as it walks up to the boundary. Does it go too far? Does it stop short? Make sure it works from all directions moving N, S, E, and W.
  - If you think your code is working correctly, but you are not matching our output, print out and check your seed first, and then check the number of random calls (`random` and `randint`) and the order you are making them. You will only need one `randint` call and one `random` call per iteration. The `seed` function should only be called **once** in any program.
  - Use special values of the probability to simplify testing. For example, a probability of 1 should capture a pokemon every turn. 0 should capture no pokemon

- In Part 2, use functions and debug them separately.
- Start out small. In Part 2 use a small grid, a small number of turns per simulation, and a small number of simulations. For example, with a 5x6 grid, 5 turns per simulation, and 2 simulations; you can easily trace your program execution using the debugger or print statements to verify execution.