CSCI 1100 — Computer Science 1
Homework 5
Wandering trainer

Overview

This homework is worth **100 points** total toward your overall homework grade, and is due Thursday, October 31, 2019 at 11:59:59 pm. You will write and debug three versions of one program, with the first version setting up a framework, the second program solving a restricted problem and the third completing the effort.

See the `Fair Warning about Excess Collaboration` documentation for a discussion of academic integrity issues. Also review the grading criteria in the `Submissions Guidelines` document and Lecture 11. For the rest of the semester, these criteria will be a significant part of your grade. You may have noticed these guidelines becoming worth more points over the past few assignments. This assignment they will be worth a significant part of the grade.

The homework submission server URL is below for your convenience:

```
https://submitty.cs.rpi.edu/f19/csci1100
```

The three versions of your program must be named:

```
hw5_part1.py
hw5_part2.py
hw5_part3.py
```

This assignment builds heavily on material from HW 3 and the random walk example from Lecture 11. Feel free to make use of your code from these assignments and the code we provided.

Overview

In this homework assignment, you will be writing a program that controls a wandering pokemon trainer in a quest to catch pokemon. The three parts of the assignment build on each other to complete the final program. Start out slowly and get the randomization functions working first (Part 1), then run a single simulation (Part 2) and finally gather statistical data by running a bunch of simulations (Part 3).

Part 1: Setting it up

In Part 2 and Part 3 you will be asked to put together a simulation of a wandering Pokemon trainer, searching the wilds for valuable and unique pokemon. In this part, we are going to address the framework that will hold it all together.

To complete this homework, you will need to define a main program and at least two functions called `move_trainer()` and `throw_pokeball()`. For part one, you do not have to fully flesh out either the main part of the program or the `move_trainer()` and `throw_pokeball()` functions. Instead, we want you to explore the use of the random functions `random.seed()`, `random.choice()` and `random.random()`.

Create the function `move_trainer()`. Do not worry about any return value from the function and do not worry about the parameters to the function. Instead set up directions as `['N', 'E', 'S', 'W']` and then in the body of the function use the `random.choice()` and `random.random()` functions to choose and print a direction and a value. The direction should be printed as a string and the value should be printed as a float accurate to 2 decimal places. Then create the function `throw_pokeball(num_false, num_true)`. This function should create a list of `num_false` boolean `False` values followed by `num_true` boolean `True` values, and then use `random.choice()` to choose and print one of the values from this boolean list.

Now add code to the program asking for an integer grid size `size`, an integer number of `False` values `F` and an integer number of `True` values. `T`. Calculate the random seed as `11 * size`, print it out and then use the `random.seed()` function to set the seed value. Now call `move_trainer` 5 times, followed by calling `throw_pokeball(F, T)` 5 times.

Two examples of the program run (how it will look when you run it using Spyder IDE) are provided in files hw5_part1_output_01.txt and hw5_part1_output_02.txt. (In order to access these files, you will need to download file `hw05_files.zip` from the `Course Materials` section of Submitty and unzip it into your directory for HW 5.)

Do not continue on before you have Part 1 working correctly. This part is not worth many points, but if this is not working, you will not be able to get the rest of the code working either. Once this works, you should not need to change any of the random calls as you work through Parts 2 and 3.

Part 2: Wandering Trainer

A pokemon trainer is placed in the middle of a grid that is `size` rows tall and `size` columns wide. This value is read into the program by asking the user. The user must also be asked for a probability p that must be greater 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 $(size-1, size-1)$. Again, the maximum limits of the grid are $(size-1, size-1)$. This is important both to get Part 2 correct and because it will cause an error accessing the tracking grid in Part 3 if you do not set this correctly. The trainer starts out at location $(size//2, size//2)$.

The program must simulate random movements of the trainer. The trainer can only move in a straight line to one of the four compass points to the North (decreasing row), East (increasing column), South (increasing row), or West (decreasing column) one step per turn. After taking a step, the trainer will have a p probability of seeing a pokemon on that spot and if they see one they will throw a pokeball for a chance to catch it. In Part 1 you put together the random calls that you need to manage moving and throwing the pokeball. In this part modify your functions to return the values you roll instead of just printing them out. `move_trainer` should return a tuple of `(direction, probability)` and `throw_pokeball` should return either `True` or `False`.

To proceed, set your trainer at the center point of the grid as described above, then move the trainer based on the results of a call to `move_trainer`. If the `probability` returned by `move_trainer` is less than or equal to the value `p`, the user sees a pokemon on the current spot and you need to call `throw_pokeball` to see if she catches it. Pokeballs are expensive. Only throw a pokeball if you see a pokemon. The first time you call `throw_pokeball` you should provide it with 3 `False` values and 1 `True` value. Thereafter, increase the number of `Trues` by one for every pokemon the trainer successfully catches. Nothing changes if the pokemon is not caught. The trainer should continue to make these random steps until she reaches the edge of the grid (row or column becomes 0 or

$size - 1$),

To solve Part 2 your program must report the position of the trainer and the number of pokemon seen, and the number actually caught.

Important Details:

1. The first time the trainer moves is turn 1
2. In each turn:
   (a) Take a step and check for a seen pokemon using `move_trainer`
   (b) If the trainer sees a pokmon, throw a pokeball using `throw_pokeball` and record if you catch it.
   (c) Report all pokemon seen and if you catch them or not.
3. At the end, output the final number of time steps, the final position, and the total number of pokemon seen and caught. See examples below for details.
4. 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 * size + size
   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. Part of the function of Part 1 of this homework is to help you verify that you are setting the seed correctly and making the correct sequence of calls to match our values. To see more interesting behavior when playing with or 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!
5. You **MUST** modify and use the following routines from Part 1

   ```
   def move_trainer():
       '''
       return a tuple, (direction, seen), where direction is one of
       'N', 'E', 'S', or 'W' and seen is a random  number between
       0.0 and 1.0 that determines if a pokemon is seen.
       '''

   def throw_pokeball(num_false, num_true):
       '''
       num_false is the number of False entries at the start of a catch list
       num_true is the number of True entries at the end of a catch list

       return a True or a False randomly chosen from the list.
       '''
   ```

   `move_trainer` determines the direction the trainer moves and whether they see a pokemon. The function must return a tuple containing in order, the direction and a probability.

`throw_pokeball` determines if the trainer catches the pokemon they saw. Your main code must call these functions 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.

6. In `move_trainer` and in `throw_pokeball` you **must** make the calls to `random.choice(direction)` and `random.random()` in the correct order and only as described above. You set this up in the routines we wrote in Part 1 and modified above. Test the values returned by those routines to decide what move to make and if a pokemon is seen or captured.

Three examples of the program run (how it will look when you run it using Spyder IDE) are provided in files hw5_part2_output_01.txt, hw5_part2_output_02.txt, and hw5_part2_output_03.txt from `hw05_files.zip`.

Part 3: 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 last 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 likelihood of catching a pokemon on each space in the grid. This should be calculated as the number of pokemon $caught - missed$.
2. The average number of turns used over all simulations.
3. The minimum and maximum number of turns in a single simulation and the simulation number (from 1 to the number of simulations run) at which these occurred.
4. The maximum pokemon likelihood, $caught - missed$, in the grid.
5. The minimum pokemon likelihood, $caught - missed$, in the grid.

A Counting Grid: You must create a list of lists of counts that maintains the difference between the number of pokemon caught on a space and the number seen but missed. Here are two examples to help you. The first example shows an easy way to initialize the grid to have `size` rows and `size` columns:

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

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` and `throw_pokeball` functions from part 2. You can copy and paste them into your `hw5_part3.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 versus the number seen but missed. prob is
      the probability a pokemon will be seen at each turn

      returns the number of turns required to reach the edge of the grid
      '''
  ```

  that starts the trainer in the center of the grid, and runs one full simulation of the trainer until the trainer reaches the edge of the grid. At the end, it returns the number of turns taken. Each time step has three phases. In the first phase, the trainer moves, during the second phase the trainer looks for a pokemon, and during the third phase the trainer tries to catch any pokemon seen phase 2. This is exactly your `move_trainer` and `throw_pokeball` functions from Part 1. `run_one_simulation` should call `move_trainer` once per time step to move the trainer and look for pokemon. If it sees a pokemon, it should call `throw_pokeball` to try and capture it. The results should be tracked and recorded in `grid`. For instance, suppose that the trainer moves to (row, col) as a result of a `move_trainer` call. If the probability returned by `move_trainer <= p`, you should call `throw_pokeball` and increment `grid_count[row][col]` by one if the throw was successful, or decrement it if the capture failed. You **must** have `run_one_simulation` in your program, but you may change its parameters as you wish. For example, you could pass in `size` — the number of rows and columns of the grid.
- 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. Just like in Part 2, each simulation starts with 3 `False`s and 1 `True` value being passed to `throw_pokemon` and the number of `True`s go up by 1 for every pokemon caught.
- 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 each simulation, you may want to have `run_one_simulation` return the number of turns used during that run. This will help with tracking the minimum and maximum number of turns.

Finally, an example of the program run provided in hw5_part3_output_01.txt illustrates the output and formatting we are expecting. A simple example showing a single iteration can be found in hw5_part3_output_02.txt from `hw05_files.zip`.

In terms of formatting, each of the output values in the grid is formatted with `{:5d}` and no spaces between entries. 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. Average should use our normal `{:.2f}` format.

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 3 a large number of iterations. This is too much to debug easily. Start small
  - Use Part 1 to get the random functions working and use Part 2 to get `move_trainer` and `throw_pokeball` fully operational and to test them thoroughly. Step through them 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.
  - In Part 2, Use a small grid, eg. 5×6 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, E, S, 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 `choice`) and the order you are making them. You will only need one `choice` call and one `random` call per call to `move_trainer` and one `choice` call per call to `throw_pokeball`. 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 see a pokemon every turn. 0 should see no pokemon
  - Use functions and debug them separately.
  - In Part 3, start out small all over again. Use the same small grid as in Part 2 and limit it to a small number of turns per simulation, and a small number of simulations. For example, with a 5×6 grid, 5 turns per simulation, and 2 simulations; you can easily trace your program execution using the debugger or print statements to verify correct execution.