CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #4

due at 5pm, Tue 21 Sep 2021

1 Overview

In this program, you're going to track a person's path as they wander through a forest. As the person walks around, they will unroll a ball of twine so that they can find their way back - and you will keep track of where the twine is.

2 Background - Stacks

A stack is a data structure which only allows you to add and remove from one end. We often draw them vertically (which is why we call them "stacks"), but an array is a handy way to track one. Each time that we want to "push" on to the stack (that is, add a new element), we will simply append to the array. Each time that we want to "pop" (that is, remove one), we will take it off of the tail end of the array.

In fact, this is such a common thing to do that Python provides a method for it. Given an array of values, you can call `pop()` to remove one of the values. It will remove the last element from the array (that is, element `[-1]`), and return it:

```
data = ... a Python list, with at least one element ...
last = data.pop()      # data is now shorter by one element
```

There are a number of different ways to implement stacks - an array is only one of them. But it's the one that I want you to use for this project!

3 `twine.py`

In this program, you are wandering through the forest; you are lost! Happily, you've brought with you a ball of twine, that you unroll as you go. That way, you can always get back to where you started. (Sadly, there's no way out of the forest. Sorry about that!)

In this program, you will model this as a small interactive game. You will prompt the user with your current state, and then they will type a command; based on that command, you will update the state, print out some sort of response, and then prompt them for another command.

The commands that you must support include: `n,s,e,w` (move one square in that direction), `back` (retrace your steps by one space), `crossings` (tell the user how many times they've been at the current location), `map` (print a map of the space you've travelled), and `ranges` (give the x and y bounds of your path so far). Note that none of these commands have any parameters.

3.1 The History

You will model the twine using a stack, and the stack must be implemented as an array; each element in the stack must be an `(x,y)` tuple, giving your position at that point in the path. If the user gives the `back` command, you must pop the most recent value off of the stack, effectively moving you backwards in time. Your start position is always `(0,0)`, and it must always be included, in the first element of the stack; you must not pop it off.

3.2 The First Question

Before the very first prompt of the game (at the very beginning of your program) `print()` the following message.

```
Please give the name of the obstacles filename, or - for none:
```

Then, get one `input()` from the user (but ignore it completely). Once the rest of your program is working, you will update this, adding the "Obstacles" feature, which is documented at the bottom of this spec.

3.3 Prompt

The prompt, before every command, must be exactly three lines, like this:

```
Current position: (3,2)
Your history:     [(0,0), (1,0), (2,0), (3,0), (3,1), (3,2)]
What is your next command?
```

and after you have printed out the response to any command, you must add a blank line before the next prompt.

(To make the output from GradeScope look a little nicer, please print out the third line of the prompt using `print()` - and then use `input()`, with no argument, to read the command.)

3.4 Errors

If the user types any command which is not valid, print out an error message. For this program, error messages must begin with

```
ERROR:
```

and the autograder will ignore the rest of the message. However, your TA will inspect your code to see if you are printing out useful error messages, which would help the user. You must detect the following user errors:

- The user types more than one word on the command line. Since no commands take any parameters, this is an error.
- The user types a command which you don't recognize.

- (When you implement Obstacles): The obstacles filename is blank
- (When you implement Obstacles): The obstacles filename doesn't exist
- (When you implement Obstacles): The obstacles file has invalid data inside it

3.5 EOF

When the autograder runs your program, it will hook up a file to the input of your program. If you call `input()` when you are already at the end of that input file, Python will throw an `EOFError`. (EOF stands for "End Of File".)

While we are not going to learn about exceptions in depth yet, I would like you to use the following code snippet:

```
try:
    user_data = input()
except:
    do_something_here
```

This code protects you from `EOFError`. If you are able to read from the user, then the `user_data` variable gets set, and you just move on to the next block of code. But if an exception occurs, then it will jump you into the `except` block - and in there, you can do whatever you want to handle the error. (For instance, you might want to break the loop that you're in.)

4 `twine.py` - Command Details

4.1 Blank Lines

If, when you're expecting a command, the input instead is a blank line, then don't treat this as an error. Instead, print out the following message, then go back and prompt for another input.

```
You do nothing.
```

4.2 `n,s,e,w`

The four directional commands are fairly self-explanatory; move in that direction, pushing your new location onto the stack. (In this map, North is in the +y direction, and East is +x.)

It's perfectly OK if the user overlaps a place they've been before; this just means that the twine crosses over an older piece. In your program, this means that multiple entries in the stack will have the same value.

You don't have to print anything; just go ahead and print the blank line, and then print the new prompt.

(You will add obstacle checking to this, later.)

4.3 `back`

This command simply pops one value off of the stack - unless the stack only has one element.

Normally, you would simply print

```
You retrace your steps by one space
```

but, if there is only one remaining value on the stack, then print

```
Cannot move back, as you're at the start!
```

4.4 `crossings`

Count how many times your current position is in the stack (including your current position as 1), and print out the result, like this:

```
There have been 3 times in the history when you were at this point.
```

NOTE: The `crossings`, `map`, `ranges` commands should all print out information only based on the **current stack** - to not try to keep additional information. If you move backwards, your program should **completely forget** where you previously had been.

4.5 `map`

Print out a grid of characters, which shows a map of the area near your starting point. It will show the origin $(0, 0)$, your path, and the obstacles (once you've implemented that). Sometimes, the user may walk far enough away from the origin that part of their path is off the map; that's OK; we simply won't see that part of it.

The map should always be exactly 11 characters wide, and 11 characters tall: it represents the world from $(-5, -5)$ to $(5, 5)$. Draw a box around it, as I'll demonstrate in the example below.

Print out the following characters:

- Print a `+` (plus) at the player's current location - even if it is also the origin.
- Print a `*` (asterisk) at the origin (that is, $(0, 0)$), unless the player is there.
- Print a `.` at any location where the player has been, but which is not the origin or the player's current position.
- Print a `X` at any obstacle. (If you haven't implemented obstacles yet, then this doesn't doesn't apply to you yet.)
- Print a blank space otherwise.

For example, if your stack contains `[(0,0), (1,0), (2,0), (2,1), (2,2)]`, and the only obstacle is at (1,2), then print the following:

```
+-----------+
|           |
|           |
|           |
|      X+   |
|       .   |
|     *..   |
|           |
|           |
|           |
|           |
|           |
+-----------+
```

NOTE: See the NOTE in `crossings` above. If the player has backtracked their path, only show the **current** path.

4.6 `ranges`

This command scans through the current history (that is, the unrolled twine) and gives the furthest West, East, South, and North that the player has walked.

```
The furthest West your twine goes is 0
The furthest East your twine goes is 2
The furthest South your twine goes is 0
The furthest North your twine goes is 2
```

NOTE: See the NOTE in `crossings` above. If the player has backtracked their path, only show the statistics for the **current** path.

(spec continues on the next page)

5 Obstacles

You can get most of the credit for this assignment without this feature. So, for your first version of this program, simply print out the prompt shown below, but ignore the first line typed by the user, and ignore the existence of obstacles. Add this feature later.

Before the first prompt to the user, you must get the name of an “obstacles” file and read it. It gives a list of locations which are blocked, and the player cannot enter.

5.1 Code to Add to Your Movement Commands

To implement obstacles, you will need to add a new check to your movement commands: `n,e,s,w`. You have only one thing to check for: if the user attempts to move into a location that has an obstacle, then print

```
You could not move in that direction, because there is an obstacle in the way.
You stay where you are.
```

and then don’t push a new location onto the stack.

If you succeed in moving, however, you don’t have to print anything; just go ahead and print the blank line, and then the new prompt.

5.2 How to Read the Obstacles File

First, give the user a prompt:

```
Please give the name of the obstacles filename, or - for none:
```

and then, on the next line, read the filename. If the user gives you `"-"`, then you must not attempt to open any file; instead, just record that there are no obstacles to worry about in your variables, and start the game.

If the user gives you anything else, then attempt to open the file given - if it works, then read the file to get the obstacles list. If it fails, then report an error message to the user (see below), give them the prompt again, and then ask again - loop until either the user gives you `"-"`, or a file that you can read.

How do you detect the condition that the file does not exist? That’s easy! Python will throw an exception if you try to open a file, and it doesn’t exist. Look at this little example snippet (it’s very much like the `EOFError` handler above):

```
try:
    my_file_obj = open("some_filename.txt")
    # if the file existed, then your code will reach this comment
except:
    # if the file didn't exist, then the code will go here.
    # make sure to write some code to do something in this case.
```

5.3 The Format of the Obstacles File

The obstacles file simply contains a set of (x,y) pairs, one pair per line. Blank lines are allowed, although this time comments are not. The obstacles file can be of any size (including zero valid lines).

A valid line in the obstacles file is simply two integers, separated by whitespace, like this:

```
1 2
-3 10
0 1
```

So long as the values are both integers (and there's nothing else on the line), you don't have to do any more error checking. In particular, you may ignore strange things like having duplicate entries, having a (0,0) entry, or having exceptionally large entries - I won't test any of these.

5.4 How to Store the Obstacles?

There are a variety of ways that you could store obstacles as you read them from the file, and you are allowed to use whatever data structure you want.

The simplest way to do this, from your perspective, is simply to use an array of tuples: each time that you read a new obstacle from the file, you add it with `append()`. This makes for simple, handy code - since you can use the `in` operator to search the list of obstacles, to see if it contains a given location.

But arrays are pretty poor, when it comes to searching. You have to look at all of the elements, one at time, to figure out whether or not the array contains what you're looking for. (Sure, the `in` operators means that you don't have to write much code - but Python is doing **a lot** of work under the covers.)

A better data structure, if you'd like to try it out, is a `set`. Down in the guts, it's basically a dictionary - meaning that the `in` operator is lightning fast. But, unlike a dictionary, it doesn't have any values; it only stores the keys.

6 Turning in Your Solution

You must turn in your code using GradeScope.