CS 101
program 8

For the last assignment of the semester we're going to write a puzzle game, called Wumpus World. Your program will provide the user with feedback about what they can see; the user must use logic (or guesswork, if they prefer) to try to succeed.

Wumpus World is fairly simple:

- Wumpus World is a 5x5 grid, with cells numbered from 0,0 to 4,4.
- The player always starts in the lower left-hand corner (cell 0,0), and the starting cell is always safe.
- Somewhere (randomly placed anywhere but 0,0) is the Wumpus, a fierce beast that will eat the player if that cell is entered.
    - The Wumpus does not move.
- Somewhere is a pile of gold, which the player is trying to retrieve.
- About 20% of the squares (other than 0,0) are pits; stepping into a pit means the player dies (loses the game).
    - Yes, the gold can be in the same square as the Wumpus. It can be retrieved if the Wumpus is dead.
    - Yes, the gold can be at the bottom of a pit. In this case the gold cannot be retrieved.
    - About 1/5 to 1/4 of the time, the placement of pits will make the gold impossible to reach, either because it's in a pit or because all paths to it are blocked. (That's nothing your program has to worry about, but be aware of it during testing.)
    - The gold will never be at 0,0.
    - If there's a pit on the same square as the Wumpus, the Wumpus is too big to fall into the pit (so a pit and the Wumpus can both be in the same cell, and the Wumpus is still dangerous if so). Of course, it doesn't really matter if the Wumpus gets you or the pit does, it's bad either way.
- The player can move North, South, East, or West, but not diagonally. The player can only sense things about cells that can be moved to.
- Wumpus World is perpetually dark (this prevents Wumpuses from reproducing), but the player can sense some of the environment:
    - If there is a pit in an adjacent cell, the player will feel a *breeze*.
    - If the Wumpus is in an adjacent cell, the player will smell a *stench*.
    - If the gold is in the same cell as the player, the player will notice a *glint*.
    - If the player tries walking off the edge of the map, the player will hit a wall and feel a *bump*, but not actually move.
    - If the player successfully shoots the Wumpus (see below), the Wumpus will emit a *scream* that can be heard from anywhere in Wumpus World.
- The player has a crossbow, but only one arrow. If the player shoots, the arrow will travel in a straight line until it hits a wall, or the Wumpus. Trying to shoot again will have no effect (but counts as a move for scoring purposes).

The player starts at 0,0. On each turn the player enters their action at the keyboard. Players may take the following actions:

- `North`, `South`, `East`, `West`: Move one square in that direction. North means "to the row with the next-higher index, staying in the same column"; South is the opposite direction. East means to the column with the next-higher index, staying in the same row; West is the opposite direction.
- `Grab`. If the player is in the square with the gold, this picks up the gold. Once picked up, the gold will never be dropped. If this is not the square with the gold, this action has no effect. Grabbing the gold removes it from its square (since the player has it now); the player can't return to this square and get more gold later. This counts as a move for scoring purposes, regardless of whether it is successful in grabbing the gold.
- `Fire [direction]`, where `[direction]` is exactly one of North, South, East, West: Fire the arrow in the specified direction. See the above description for what happens if this action is taken.
- `Climb`: if the player is at 0,0, this takes the player out of Wumpus World, with or without the gold, and ends the game. If the player is anywhere else, this has no effect (but does count as a move for scoring purposes).

Your program should also offer:

- `Help`. This prints a short summary of what actions can be taken and the correct format. (For example, "`fire north`" is a valid command, but "`firenorth`" or "`fn`" aren't.)

A typical game might start something like this:

Player action	Program response	Player reasoning
(Start game)	`It is dark.`	There is no breeze or stench here. Therefore squares adjacent to 0,0, namely 1,0 and 0,1, must be safe. Explore a safe but unknown square: pick 0,1 arbitrarily
`West`	`You feel a bump as you walk into a wall.`	Oops, wrong direction.
`East`	`You feel a breeze.`	Danger! I know there's no pit at 0,0 (origin is always safe, and I was just there). There must be a pit north or east of this cell, at 1,1 or 0,2. No smells; the Wumpus isn't adjacent to this cell.
`West`	`It is dark.`	Back to origin to explore other safe cell.
`North`	`You smell a terrible stench.`	Now in 1,0. There is no breeze, so there is no pit next to this cell, so no pit in 1,1 or 2,0. Since there must be a pit in 1,1 or 0,2 and it's not in 1,1, there must be a pit in 0,2. I can smell something here, so the Wumpus must be in 1,1 or 2,0. Since there was no smell in 0,1, the Wumpus can't be in 1,1, so the Wumpus must be in 2,0, directly ahead.

`fire North`	`You shoot an arrow.` `You hear a horrible scream.`	The Wumpus is dead.
`North`	`You smell a terrible stench.` `There is a dead Wumpus here.`	No mention of breezes, so no pits in adjacent squares: cell 1,1, cell 3,0, and cell 2,1 are therefore known to be safe. The Wumpus is dead, so it's safe to ignore smells from now on.

And so it goes. The player must still avoid pits while searching for the gold, then returning to 0,0 via safe squares to climb out. Notice that *the player does not get feedback about their location;* they must track that on their own.

On each turn, your program will print one or more of the following messages, as appropriate:
`It is very dark.` (This is the "nothing specific to report" message.)

```
You feel a breeze.
You smell a terrible stench.
There is a live Wumpus here!
There is a dead Wumpus here.
You shoot an arrow.
You try to fire but are out of arrows.
You hear a horrible scream.
You see a glint of something that looks like gold.
You pick up a pile of gold.
You do not pick up anything.
You feel a bump as you walk into a wall.
You have fallen into a pit.
You cannot climb up from here.
You climb up out of Wumpus World.
```

Feel free to add appropriate descriptions if the player falls into a pit or is eaten by a Wumpus.

At the end of the game (when the player climbs out or dies), report their score:
+1000 Climbed out with gold
+100 Climbed out without gold
-10 Fired weapon
-1 per move

Score 0 if the player died.

Design notes:

- Your program knows the map, so the primary problem here is tracking the player's location and deciding on appropriate feedback based on what the user does.
- Your code must respond appropriately if the user enters an invalid command. For example, `fire` (with no direction) is not valid.
- The user should be able to enter their input in upper case or lower case, in any combination.

- Write a Cell class. This class will model one square in the map. A Cell simply has a few boolean variables; for example, hasPit, hasGold, isBreezy, etc.
- Write a Map class. A Map class contains a 2-dimensional list of Cells. It also has the following methods:
  - isBreezy(self, row, col): returns True if the specified cell is breezy, False otherwise.
  - isSmelly(self, row, col): returns True if the specified cell has a stench, False otherwise
  - isGlinty(self, row, col): returns True if the gold is in that square, false otherwise.
  - hasWumpus(self, row, col): returns True if the Wumpus is there, false otherwise.
  - hasPit(self, row, col): returns True if the cell has a pit, False otherwise.
  - Any reference to a row or column off the map will raise an `OffMapError`, which is an exception class that you will write.
  - You may need additional methods as well; for example, if the player grabs the gold, then the gold should be removed from the map and any glittery cells updated to reflect that.
  - reset(self). Clear all data from old map and make a new map:
    - The Wumpus is randomly placed anywhere except 0, 0
    - The gold is randomly placed anywhere except 0,0
    - Each Cell other than 0, 0 has a 20% chance of containing a pit; these are placed as well.
- Write a WumpusWorld class. A WumpusWorld class will play a game of WumpusWorld. It will have a Map as part of its data. It will also track the player's location and keep track of information such as whether the player has fired the crossbow or grabbed the gold. It will also be able to call the Map's reset method if the player wants to go again.
- Your main program will make a WumpusWorld object. The program will handle user interaction, getting input and using it to call methods on the WumpusWorld object, and reporting the WumpusWorld's responses back to the user.
- ***Extra credit:*** For 5 points extra credit, allow the player to enter commands using only the first letter. So movement commands would be N, S, E, W; grab would be G; shoot north would be F N; etc. To get the credit, the user must be able to enter the regular or one-letter commands in any combination. (For shooting, the 2 components of the command should be separated by a space character.)
  - This is NOT just looking at the first letter and ignoring any other input. This code should still recognize `F norbert` or `fire why` or `eat` as invalid commands.
- ***Extra credit:*** For 5 points extra credit, give the program the ability to read an input file describing a map and use that, and to save a map in the same format. The map is just a text file with a list of features followed by their grid locations, one per line, like this:

  GOLD 3 2
  PIT 0 2
  PIT 4 1
  WUMPUS 1 2

  and so on. Items are not listed in any particular order. Your saved file should be in the same format. You should verify that all locations are valid, that there is one and only one square with a Wumpus, one and only one square with gold, and that the origin 0,0 is safe (no pit, no Wumpus, but also no gold).

    - The format for these additional commands are: `Load` *`filename`* and `Save` *`filename`*. These should *not* have 1-letter abbreviations (due to the conflict between `Save` and `South`; the command `S aname.txt` becomes ambiguous; is it a Save command, or a badly formatted command to go South?
    - Yes, you can do both extra-credit features. This makes it possible to earn 70/60 on this program.
- Your program does not need to do anything in the "Player Reasoning" column above; that's just to give an idea of what game play is like.
- An earlier version of the game offered a score of +100 for killing the Wumpus, but this was removed after complaints from People for the Ethical Treatment of Wumpuses and the Wumpus Preservation Society.

Historical note: Wumpus World has a long and colorful history in artificial intelligence research. In this case, it is presented to a program to solve as the player. Using only the information presented, the program must update what it knows about the world, deduce what it can from that, and formulate a plan of action. Knowledge representation, logical inference, and planning are all key areas of AI research. Wumpus World is often used as a simple testing environment for trying new ideas or demonstrating techniques.

APPENDIX: Data dictionary & Class Map

class OffMapError, class Cell, class Map should all be in source file Map.py

class OffMapError(Exception): raised if attempt to go off map.
Methods:
    __init__(self)

class Cell(object): Models one cell (square) in the map.
Data:
    self.hasWumpus
    self.hasGold
    self.hasPit
    self.hasBreeze
    self.hasStench
Methods:
    __init__(self)

class Map: manages the entire map, assigns values to cells, reports status of map cells
Data:
    self.grid
Methods:
    __init__(self)
    onGrid(self, r, c)
    offGrid(self, r, c)
    reset(self)
    isBreezy(self, r, c)

isSmelly(self, r, c)
isGlinty(self, r, c)
hasWumpus(self, r, c)
hasPit(self, r, c)

class WumpusWorld should be in source file Wumpus.py

class WumpusWorld: contains a map, records player movement & current game state
Data:
self.worldmap
self.WumpusAlive
self.playerAlive
self.playerHasGold
self.playerHasArrow
self.playerMoves
self.playerRow
self.playerCol

Methods:
__init__(self)
# most of these use current player location
stepEast(self)
stepWest(self)
stepSouth(self)
stepNorth(self)
grab(self, r, c)
grabGold(self) # checks current r, c, calls grab(self, r, c)
fire(self, direction)
canClimb(self)
feelBreeze(self)
smellStench(self)
seeGlint(self)
hasWumpus(self)
hasPit(self)

main program should be in source file Program8.py

Main program: Handles all user interactions, gets input, checks it for validity, passes valid input to methods of WumpusWorld object, provides user feedback based on results, computes final score.