Pynopoly

Mar 2, 2018

Introduction

The objective of this homework is to write an interactive version of Monopoly in Python: *Pynopoly*. If done correctly, you will have a chance to fulfill your wildest dream: utilizing all of the material we've learned thus far.

Monopoly is a popular board game in which players buy and develop property, and charge various rates of rent for staying ("landing," really) on that property. The objective of the game is to drive all of your opponents into bankruptcy. Wikipedia provides a good outline.

Monopoly can be quite intricate; thus, we will make several simplifications. Our version of the game consists of players and a board. Players move around the board. For each space, they must either pay rent or decide to purchase, depending on whether the spot is available and who owns it. Some spaces are special: "Go," for example, allows a player to collect money; taxation spaces, on the other hand, require the player to pay some amount of money.

We will focus on implementing Monopoly's basic game play. Making it look pretty is secondary. Thus, developing an ASCII art version of the board, as we did with Battleship, won't be required.

Basic functionality (30 points)

Before programming the game interaction, there are several functions that must be implemented. These functions will make the interaction easier to build. To receive full credit for these functions, their definitions—the function name, arguments, and return type—must be implemented exactly as specified.

Create the game board (5 points)

```
def get_board(csv_file):
    # returns a list
```

Given a CSV file, this function should return a list of board spaces.

The specific CSV file, available for download here, provides a definition of a standard Monopoly game board. The first line in the file is a header, providing a description of what the given column specifies:

Column	Type	Description
name	`str`	Property name
cost	`int`	Property purchase price
owner	`str`	Property owner
type	`int`	The property's type, and, by extension, whether the property is purchasable. A value of zero means the property cannot be purchased; values greater than zero mean that it can, and further signify: 1) the property is a street, 2) the property is a railroad, or 3) the property is a utility
rent	`int`	Amount of rent to be paid to the owner

Each subsequent line of the file represents a tile on the original game board. You are free to represent the tile as you see fit. Your function must return a list of tiles with the ordering of the CSV file preserved. For example, "Go" is the first tile in the CSV file; thus, it must be the zeroth element in the list that is returned. Finally, your implementation should not assume a certain ordering of the columns.

Create the players (5 points)

```
def get_players(n)
    # returns a list
```

Given an integer, creates that number of players. Player names can be created dynamically; for example, if `i` is an integer in the range zero to `n`, a valid player name could be `'Player ' + str(i)`. You can be more creative with player names if you like, however, two things must be respected:

1. Player names must be created via programming logic. Asking for user input to generate names, or reading from a file, is not allowed.

2. Each player name in the returned list should be unique.

Along with a name, each player should have a position, and a wallet. The position is where the player currently resides on the board. Initially, this should point to the spot on the board named “Go.” The wallet represents the amount of money currently available to the player. As per the game rules, this should initially be $1,500. As the game progresses, this value should be incremented and decremented accordingly.

Notice that, as with the board, a player is actually a collection of attributes. Further, in this implementation, the *collection* of players is represented in the same way as tiles on the board: via a list. Thus, how you choose to represent the properties on the board will likely work for representing players in the game.

Print the game board (5 points)

```
def print_board(board, players)
    # no return value
```

Given a list of tiles (the return value of `get_board`) and a list of players, prints the name of each tile. The tiles “owner” should be printed if that tile is purchasable and has an owner other than the bank. If there are players currently residing on that tile, they should also be printed. As an example, consider a board consisting of the first eight spaces of the provided CSV, and four players:

```
>>> b = get_board('monopoly.csv')
>>> players = get_players(4)
>>> print_board(b, players)
Go
  Residents: Player 2, Player 3
Mediterranean Avenue
Community chest
  Owner: Player 4
Baltic Avenue
  Residents: Player 4
Income Tax
Reading Railroad
Oriental Avenue
  Residents: Player 1
  Owner: Player 2
Chance
```

Print a single player (5 points)

```
def print_player(player, board)
    # no return value
```

Given a player (an element from the list returned by `get_players`), and the board, this function should print information about that player. Specifically, the square on which the player resides, the amount of money the player currently has, and what properties the player owns.

```
>>> b = get_board('monopoly.csv')
>>> players = get_players(4)
>>> print_player(players[2], b)
Player 3
  Wallet: 1500
  Position: Go
  Properties: (None)
>>>
>>> # ... after some amount of game play...
>>> print_player(players[1], b)
Player 2
  Wallet: 825
  Position: St. Charles Place
  Properties:
    Indiana Avenue
    Ventnor Avenue
```

Move a player around the board (5 points)

```
def move_player(player, board, dice_roll)
    # no return value
```

Moves a player `dice_roll` positions away from their current position on the board; `dice_roll` can thus be thought of as the result of rolling the dice. Although traditional Monopoly is played with two six-sided die—and this implementation will stay true to that—this function should not assume a maximum value for `dice_roll`.

In Monopoly, when a player passes “Go,” that player collects some amount of money from the bank. The same should happen in this implementation. Thus, `move_player` should recognize whether such an event happens, and update the players wallet accordingly. The amount to increment should come from the “rent” attribute of the Go property. (Note: in the CSV file, Go’s rent has been specified using a negative number. This is done to denote the fact that this is a payment to the player from the bank, which is Go’s owner).

Property type counter (5 points)

```
def type_counter(player, board, tile_type)
    # returns an integer
```

Given a player, a board, and a tile type, returns the number of tiles of that type owned by the player. For example, if player `p` owned Pennsylvania Railroad and B+O Railroad,

```
>>> print(type_counter(p, board, 3))
2
>>>
```

Game play (40 points)

Playing the game consists of giving each player a turn. An implementation should first create the board and the players, then begin a loop allowing each player to decide on a particular action. As an example:

```
board = get_board('monopoly.csv')
players = get_players(4)
while True:
      # ...
```

This is merely an example! Your game should:

- work with any number of players; and
- use the type of loop, and the condition on which that loop should run, that you deem appropriate.

Once in the loop, however, there are three things that need to happen:

Interaction (10 points)

Prior to moving the game piece, you should give the user a few options:

```
>>> Player 4 [i: info, b: board, p: play, q: quit]
```

The prompt should include the player's name and a list of options. A user specifies their option of interest by typing the single letter prior to the colon. These options should do one of the following:

- **Information:** Provide information about the player (call `print_player`).
- **Board:** Print the current state of the board (call `print_board`).
- **Play** Commence with the turn! Essentially, roll the dice and make some decisions (both described later).
- **Quit** Leave the game play loop.

If the user types something outside of these four values, the prompt should be presented again. Printing player or board information should not result in a player loosing their turn: the next player gets a turn only after the current player selects "p" and plays.

Play (20 points)

Once a user elects to "play," the current player should roll the dice and the player should be advanced accordingly (`move_player`). You can simulate a dice roll using Python's random number generator. Once the player has been moved, the new position should be announced:

```
Player 4 [i: info, b: board, p: play, q: quit] p
Welcome to Marvin Gardens!
```

In the new position, there are effectively two that actions need to be handled: when a player has the option to purchase a property, and when a player is required to pay rent. You can determine either using the information in the CSV file.

1. If the square is purchasable, owned by the bank, and the player has enough money, give them the option to buy:

   ```
   >>> Would you like to buy? [y/n]
   ```

   If they decide to buy, you will need to commence with the transaction: deduct the purchase amount ("cost") from the player and update the property's "owner" to that player.

2. If purchasing is not an option, decide if the player has to pay rent. Specifically, if the square’s type is larger than zero, and the owner is someone other than the bank. If this is the case, then the rental amount must be deducted from the player and given to the owner. The amount of rent depends on the property type:

    1. If the property is a street, then rent is the “rent” as defined in the CSV file;

    2. If the property is a railroad, then rent is 25 times the number of railroads held by the current railroad owner;

    3. If the property is a utility, then rent is a function of the dice roll and the number of utilities held be the current owner. Specifically, the roll should be multipled by four if the owner owns a single utility; by 10 if they own both.

You may recognize that these two conditions do not cover all properties on the board—that is okay. All properties of the original game are specified in the CSV file for completeness; if you happen to finish this homework, you can add to the aforementioned decision structure to bring those properties into the playing fold. For now, however, if a player lands on a property that does not fit these two conditions you can think of it as that player merely advancing their position.

Once you have decided what to do with the square, you should check to see if the player is still eligible to play: if the player has less-than zero dollars, they should be removed from the game (or at least no longer be allowed to play). When a player is in this position all of their properties should be returned to the bank.

Note that it is okay if, for a moment, a player has a negative amount of money. This might be the case, for example, if they have to pay a tax or a rent that is more than the amount in their wallet. In such an instance, you can assume that the entire amount is paid to the reciever (and thus the final amount in the wallet is negative).

End of game (10 points)

Monopoly is over when a single player with more than zero dollars remains. When this is the case, that player should be declared the winner and the game should end.

Expectations (30 points)

1. The code that you submit should run. Even if you have not implemented all parts of the assignment, the subset of parts that are implemented should be free of Python errors.

2. There should be comments throughout the program:

    - A few lines at the top of the file specifying (at least) your name and net ID.

    - If you write a function, talk about, at a hi-level, what it does. What types does it take; what type does it return? What's the point?

    - If the block of a loop or a decision structure is longer than six or seven lines, a comment at the top outlining the point of the block is helpful.

    - If you use a concept that has not been covered in class, something you found online perhaps, you should add a comment explaining what is going on. Without an explanation, we can only assume you have cheated.

Overtime

For those that finish early and are looking for an additional challenge, Monopoly—especially this version—is ripe for added features:

- Note that some of the squares on the board are unused; notably, those squares that passed no test in our decision structure. Use those squares to add functionality to your game. “Chance,” for example, might actually be a guessing game where the user bets some amount of money.

- The actual game allows players to “develop” their property with houses and hotels—add this to your game.

- Fix the number of players by removing the interactive prompt. Have players make purchasing decisions based on some predefined heuristic. Play several iterations of the game (thousands) to see how the heuristics interact. For example, is purchasing every property a player can afford a good decision? Are only purchasing particular types of properties a good decision?

Refer to WikiBooks for the official rules of the game.

If you decide to implement these features, please make two submissions: one that implements the original specification, and another that implements the extra features.

Submission

A single Python file should be submitted via NYU Classes, against the corresponding homework assignment. There is no need to submit the CSV board file.