Game of Tic-Tac-Toe

This program implements this popular two-player game. You play against the computer.

Explore the game:

If you want to play with my final program to get a feel for this game, click the link given at the end of the article. Try not to peek at the code yet, since we want to design them ourselves below.

1. Run the program in a Python environment.
2. Follow the instructions.
3. The program decides randomly who will go first for each round.

Download all the files mentioned in this document by clicking here.

Python and CS Concepts Used

When we design this program, we will make use of the following Python and CS concepts. Learn these concepts if you don't know them before proceeding further.

- Algorithms
    - Designing new algorithms
- Arithmetic
    - Expressions
    - Basic operators (+, -, *, /)
- Conditional statements:
    - Conditions: YES/NO questions
    - Relational operators (==, <, >, etc.)
    - Conditionals (IF)
    - Conditionals (If-Else)
    - Conditionals (nested IF)
    - Boolean operators (and, or, not)
- Data structures – list
    - List operations
    - Using list as 2-D array
- Data types – basic

    - Integers
- Data types – strings
    - String operations
- Events
- Looping (iteration)
    - Looping - simple (for)
    - Looping - nested
    - Looping - conditional (while)
- Procedures
    - User defined (custom)
    - Procedures with parameters and return value
- Program output
    - Text
    - GUI (Tkinter)
- Random numbers
- Sequence
- User input
    - Text
    - Click buttons
- Variables
    - Simple
    - Local/global scope

Version 1 High Level Design

This "dumb" version of our program only supplies the logistics of the game, i.e. the board, and allows 2 human players play the game between them.

Let's consider how the various features of this program can be separated out as distinct pieces.

As usual, we can consider the backend and the frontend separately. The backend will hold the board in some variable(s) and will have all the logic to manipulate the contents of the board as the game proceeds. The frontend will display the board in a 3x3 grid format and pass user input (which cell was clicked) to the backend. The frontend will always show the current state of the board.

For the frontend, we will use a 3x3 grid of Tkinter labels to display the board in a 3x3 format.

The backend will internally represent the board as a 3-item list in which each item (represents a row and) is a list of 3 numbers. The backend logic needs to do the following:

(1) Keep track of the 2 players: let's call them Fred and Prem. Fred will use the Xs and Prem the Os.
(2) Keep track of whose turn it is. We can do this by using a variable that alternates between the two players.
(3) Whenever a blank cell is clicked, fill it with the current player's shape.
(4) Detect when the game is over and declare the winner. For this purpose, we will need an algorithm (procedure) to scan the entire board for a row, column, or diagonal filled with the same shape (X or O).

Objects:

We will distribute the code among the following objects:

- Frontend object will contain all logic related to the visible 3x3 grid.
- Backend logic will contain all code related to features described above.

Feature Idea # 1: The tic-tac-toe board

Draw a 3x3 grid of clickable cells.

Design:

Step 1: The visible board

For this, we can simply borrow an older Python program called "matrix" which provides just this functionality using clones. This uses a label tkinter widget in a 3x3 grid.

Each cell (i.e. label widget) will possess row and column numbers so that the backend can associate each visible cell with a location on the game board. When a cell is clicked, the clickLabel callback function will send these row/column numbers to the backend via variables.

Step 2: The board state

As discussed earlier, we will use a 3-item list of lists to represent the 3x3 board. Each item (i.e. sub-list) will represent a row.

When a cell is clicked, it can simply refer to its row/column to know which item in the list it belongs to and set it to X or O. For a reason that will become clear soon, we will use numbers to represent the shapes: 1 for X, -1 for O, and 0 for blank. When a cell is clicked, its state will be changed based on whose turn it is. And to ensure the visible board is in sync, the backend will send a "refresh" message every time the board state changes.

Feature Idea # 2: Show the Xs and the Os

When a player clicks on a blank cell his/her shape (X or O) is placed.

Design:

We will use a "player" variable to keep track of whose turn it is. Fred plays first and uses the Xs. Prem plays second and uses the Os. In order to display the shapes, we could allow each label to have 3 image properties: one blank, one with "X" and one with "O". Depending on the state of each cell (controlled by the backend), the appropriate image property can be displayed. The "refreshGrid" script in frontend will synchronize the display with the state of the actual board.

Feature Idea # 3: Winner or stalemate

After every move check the board for a winner or stalemate.

Design:

This is the most challenging part so far. We need to scan rows, columns, and diagonals to detect a winner. This is where our earlier decision to denote an X with 1 and an O with -1 comes handy.

Before checking for stalemate (i.e. the grid is full) we should check for a winner. Here is the high-level algorithm:

```
Algorithm Check Winner
Check if any of the rows contains all Xs or Os and if yes,
      Declare winner and restart.
```

```
      Stop
Do the same for columns.
Do the same for diagonals.
Check if the grid is full and if yes, declare stalemate and restart.
```

To check if a row, column, or diagonal is filled with the same shape, we can simply add up the numbers. If the sum is 3, Fred is the winner. If the sum is -3, Prem is the winner. All other cases do not matter.

Now, let's design the algorithms for the scanning task. While we are at it, we should also check the condition when the grid becomes full (see the last algorithm).

Algorithm for scanning rows:

There are 3 rows starting at item (0,0), (1,0), and (2,0) respectively. Each row consists of 3 consecutive items. Using this information we can come up with the algorithm as below:

```
I = 0
Repeat 3 times
      Add up items of row I
      If sum is 3, return "Fred" as the winner
      If sum is -3, return "Prem" as the winner
      I = I + 1
End repeat
Return "None"
```

Algorithm for scanning columns:

For each column the row index varies continuously. For example, first column consists of items (0, 0), (1, 0), and (2, 0). Using this idea we can come up with the algorithm as below:

```
I = 0
Repeat 3 times
      Add up items in column I
      If sum is 3, return "Fred" as the winner
      If sum is -3, return "Prem" as the winner
      I = I + 1
End repeat
Return "None"
```

Algorithm for scanning diagonals:

There are 2 diagonals: one consists of items (0, 0), (1, 1), and (2, 2), and the other consists of (0, 2), (1, 1), and (2, 0). Here is the algorithm:

```
Add up items (0, 0), (1, 1), and (2, 2)
      If sum is 3, return "Fred" as the winner
      If sum is -3, return "Prem" as the winner
Add up items (0, 2), (1, 1), and (2, 0)
      If sum is 3, return "Fred" as the winner
      If sum is -3, return "Prem" as the winner
Return "None"
```

Algorithm for grid full:

This is a simple algorithm in which we just need to check if there isn't a single item which is 0.

```
I = 0
Repeat 3
      J = 0
      Repeat 3
            If element (I, J) is 0
                  Return False
            Increment J by 1
      End repeat
      Increment I by 1
End repeat
Return True
```

Save as Program Version 1

Congratulations! You have completed all the features of Version 1. Compare your program with my program at the link below.

File: tic-tac-toe-1.py

Version 2 High Level Design

In this "intelligent" version, we will allow the computer to be one of the players. The computer won't really use any intelligence for the moves, but will pick moves randomly from the available blank cells.

Most of the logic of Version 1 can be used as is. In order to allow the computer to play, we will need to "simulate" a click, i.e. run the code under "clickLabel" when the computer picks a cell. And then, the computer will play in place of "Prem".

Feature Idea # 4: Simulate click

Arrange the code such that a cell can be clicked from inside the program.

Design:

Right now, the "clickLabel" function doesn't do any work; it just sends the cell's row/column info to the backend. So, the "computer" player can do the same, i.e. when it is its turn to play, it can pick its cell and message its row/column info to the backend.

Feature Idea # 5: Replace Prem by Computer

Have the computer play in place of Prem.

Design:

Right now, whenever it is Prem's turn to play, the program just waits for him to click. Instead, the program will "simulate" a click by calling the cellClicked procedure (as made possible by Feature idea #4 above).

Step 1: Call cellClicked when it is Prem's turn.

At the end of the "click processing" script we can check the "player" variable and if it is "Prem" – indicating that it is Prem's turn – we can initiate this call. Replace "Prem" by "Computer" everywhere. Also introduce a variable called "turn" which will indicate whose turn it is to play. Finally, instead of using "Fred" ask the user for his/her name.

Step 2: Pick a blank cell at random.

Of course, before sending the message to the backend the computer needs to pick a cell at random from the available empty cells. The list can tell us which cells are empty. We could count the empty cells and then use the pick random operator to pick one of them. See the algorithm below:

```
Count = number of empty cells (items in "board" that are 0)
R = pick a random number between 1 and Count
The R'th empty cell is our cell.
```

Save as Program Version 2

Congratulations! You have completed all the features of Version 2. Compare your program with my program at the link below.

File: tic-tac-toe-2.py

Final version High Level Design

In this version, we will allow make the computer a bit more intelligent. We will use some simple rules to avoid completely dumb moves. For example, if the opponent is about to win (when only one blank cell is remaining to complete a row, column, or diagonal), the computer should thwart that possibility. Conversely, if the computer sees an opportunity to win (when a row, column, or diagonal is already filled with 2 of its own shapes) it should grab it. Since the basic work for both these rules is the same, we can combine it in a single feature idea as explained below.

Feature Idea # 6: Fill the 3rd cell

Check if the 3rd cell of a row, column, or diagonal helps you win or prevent a loss.

Design:

We will once again have to scan all rows, columns, and diagonals and check for this state. The above algorithms (used in Feature #3 for scanning) can be adapted for this purpose. We should first look for "win" before worrying about "loss", right? This can cause duplication of work, because we will need to scan all rows (or columns or diagonals) for "win" and then scan again for a "loss". One workaround to avoid this duplication is to give priority to a win, but remember a "lossy" row. See the algorithms below to understand this better.

Algorithm for scanning rows:

There are 3 rows starting at item (0,0), (1,0), and (2,0) respectively. Each row consists of 3 consecutive items. Using this information we can come up with the algorithm as below:

```
Algorithm: ScanRowsForWinLoss
Return value: cell row/column to use or (-1,-1)
retVal = (-1,-1)
I = 0
Repeat 3 times
      Add up items of row I
      If sum is -2 or -2
            Locate the empty cell
            retVal = row/column of this cell
      If sum is -2  // chance to win!
            Return retVal
```

```
      // do nothing if sum is 2
      I = I + 1  // go to the next row
End repeat
Return retVal
```

Algorithm for scanning columns:

For each column the row index varies continuously. For example, first column consists of items (0, 0), (1, 0), and (2, 0). Using this idea we can come up with the algorithm as below:

```
Algorithm: ScanColumnsForWinLoss
Return value: cell row/column to use or (-1,-1)
retVal = (-1,-1)
I = 0
Repeat 3 times
      Add up items in this column (items I, I+3, and I+6)
      If sum is -2 or -2
            Locate the empty cell
            retVal = row/column of empty cell
      If sum is -2  // chance to win!
            Return retVal
      // do nothing if sum is 2
      I = I + 1  // go to the next column
End repeat
Return retVal
```

Algorithm for scanning diagonals:

There are 2 diagonals: one consists of items (0, 0), (1, 1), and (2, 2), and the other consists of (0, 2), (1, 1), and (2, 0). Here is the algorithm:

```
Algorithm: ScanDiagonalsForWinLoss
Return value: cell row/column to use or (-1,-1)
retVal = (-1,-1)
Add up items of first diagonal
      If sum is -2 or -2
            Locate the empty cell
            retVal = row/column of this cell
      If sum is -2  // chance to win!
            Return retVal
      // do nothing if sum is 2
Add up items of second diagonal
      If sum is -2 or -2
            Locate the empty cell
```

```
            retVal = row/column of this cell
      If sum is -2  // chance to win!
            Return retVal
      // do nothing if sum is 2
Return retVal
```

Save as Program Version "Final"

Congratulations! You have completed all the main features of the game. Compare your program with my program at the link below.

File: tic-tac-toe-final.py

How to play the game:

1. Run the program in a Python environment.
2. Follow the instructions.
3. The program decides randomly who will go first for each round.

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 4 February 2020