# Tic-Tac-Toe (Monte Carlo)

Help Center

## Overview

Tic-Tac-Toe is a simple children's game played on a $3 \times 3$ grid. Players alternate turns placing an "X" or an "O" on an empty grid square. The first player to get three-in-a-row wins. If you know the appropriate strategy and your opponent does not, you cannot lose the game. Further, if both players understand the appropriate strategy the game will always end in a tie. An interesting variant of the game is "reverse" Tic-Tac-Toe in which you lose if you get three-in-a-row. The game is also more interesting if you play on larger square grids.

For this assignment, your task is to implement a machine player for Tic-Tac-Toe. Specifically, your machine player will use a Monte Carlo simulation to decide its next move. We will provide both a console-based interface to the game where your machine player will play against itself and a graphical user interface where you can play against your machine player. Although the game is played on a $3 \times 3$ grid, your version should be able to handle any square grid. We will continue to use the same grid conventions that we have used previously.

For this mini-project, we will provide you with a complete implementation of a Tic-Tac-Toe Board class. However, for your part of the mini-project, we will provide only a very minimal amount of starting code. We will also dispense with the phased description of the implementation process so that your coding task for this mini-project is a more realistic example of the software development process.

## Provided Code

We have provided a `TTTBoard` class for you to use. This class keeps track of the current state of the game board. You should familiarize yourself with the interface to the `TTTBoard` class in the `poc_ttt_provided` module. The provided module also has a `switch_player(player)` function that returns the other player ( `PLAYERX` or `PLAYERO` ), and a `play_game(mc_move_function, ntrials, reverse)` function that uses the `mc_move_function` you provide to play a game with two machine players on a $3 \times 3$ board. The `play_game` function will print the moves in the game to the console. Finally, the provided module defines the constants `EMPTY` , `PLAYERX` , `PLAYERO` , and `DRAW` for you to use in your code. The provided `TTTBoard` class and GUI use these same constants, so you will need to use them in your code, as well.

At the bottom of the template, there are example calls to the GUI and console game player. You may uncomment and modify these during the development of your machine player to actually use it in the game. The `run_gui` function takes five arguments: the dimension of the board, which player the machine player will be, a move function, the number of trials per move, and a reverse argument indicating whether or not you want to play the normal ( `False` ) or reverse ( `True` ) game.

## Testing your mini-project

As always, testing is a critical part of the process of building your mini-project. Remember you should be testing each function as you write it. Don't try to implement all of the functions and then test. You will have lots of errors that all interact in strange ways that make your program very hard to debug.

- As you implement your machine player, we suggest that you build your own collection of tests using the `poc_simpletest` module that we have provided. Please review this page for an overview of the capabilities of this module. These tests can be organized into a separate test suite that you can import and run in your program as we demonstrated for Solitaire Mancala. To facilitate testing on the first few mini-projects, we will create a thread in the forums where students may share and refine their test suites for each mini-project.
- Finally, submit your code (with the calls to `play_game` and `run_gui` commented out) to this Owltest page. This page will automatically test your mini-project. It will run faster if you comment out the calls to `play_game` and `run_gui` before submitting. Note that trying to debug your mini-project using the tests in OwlTest can be very tedious since they are slow and give limited feedback. Instead, we *strongly* suggest that you first test your program using your own test suite and the provided GUI. Programs that pass these tests are much more likely to pass the OwlTest tests.

Remember that OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums.

When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this mini-project that is linked on the main assignment page.

## Machine Player Strategy

Your machine player should use a Monte Carlo simulation to choose the next move from a given Tic-Tac-Toe board position. The general idea is to play a collection of games with random moves starting from the position, and then use the results of these games to compute a good move. When you win one of these random games, you want to favor the squares in which you played (in hope of choosing a winning move) and avoid the squares in which your opponent played. Conversely, when you lose one of these random games, you want to favor the squares in which your opponent played (to block your opponent) and avoid the squares in which you played. In short, squares in which the winning player played in these random games should be favored over squares in which the losing player played.

Here is an outline of this simulation:

1. Start with the current board you are given (let's call it `board`).
2. Repeat for the desired number of trials:
   A. Set the current board to be `board`.
   B. Play an entire game on this board by just randomly choosing a move for each player.
   C. Score the resulting board.
   D. Add the scores to a running total across all trials.
3. To select a move, randomly choose one of the empty squares on the board that has the maximum score.

These steps should be relatively straight-forward except for step 2C, scoring the board. Scores are kept for each square on the board. The way you assign a score to a square depends on who won the game. If the game was a tie, then all squares should receive a score of 0, since that game will not help you determine a winning strategy. If the current player (the player for which your code is currently selecting a move) won the game, each square that matches the current player should get a positive score (corresponding to `SCORE_CURRENT` in the template, which is the scoring value for the current player) and

each square that matches the other player should get a negative score (corresponding to `-SCORE_OTHER` in the template, which is the scoring value for the other player). Conversely, if the current player lost the game, each square that matches the current player should get a negative score ( `-SCORE_CURRENT` ) and and each square that matches the other player should get a positive score ( `SCORE_OTHER` ). All empty squares should get a score of 0.

Note that you want to select a final move from the total scores across all trials. So, in step 2D, you are adding the scores from 2C to the running total of all scores. Higher scores indicate squares that are more likely to be played by the current player in winning games and lower scores indicate squares that are more likely to be played in losing games.

## Implementation

Your task is to implement the following four functions: `mc_trial` , `mc_update_scores` , `get_best_move` , and `mc_move` . These four core functions should do the following:

- `mc_trial(board, player):` This function takes a current board and the next player to move. The function should play a game starting with the given player by making random moves, alternating between players. The function should return when the game is over. The modified board will contain the state of the game, so the function does not return anything. In other words, the function should modify the `board` input.
- `mc_update_scores(scores, board, player):` This function takes a grid of scores (a list of lists) with the same dimensions as the Tic-Tac-Toe board, a board from a completed game, and which player the machine player is. The function should score the completed board and update the scores grid. As the function updates the scores grid directly, it does not return anything,
- `get_best_move(board, scores):` This function takes a current board and a grid of scores. The function should find all of the empty squares with the maximum score and randomly return one of them as a `(row, column)` tuple. It is an error to call this function with a board that has no empty squares (there is no possible next move), so your function may do whatever it wants in that case. The case where the board is full will not be tested.
- `mc_move(board, player, trials):` This function takes a current board, which player the machine player is, and the number of trials to run. The function should use the Monte Carlo simulation described above to return a move for the machine player in the form of a `(row, column)` tuple. Be sure to use the other functions you have written!

You should start from this code that imports the Tic-Tac-Toe class and defines several useful constants. You may add extra helper functions if so desired. However, the signature of the four functions above must match the provided description as they will be tested by the machine grader.

Once you have working code, you will want to experiment with the values of `NTRIALS` , `SCORE_CURRENT` , and `SCORE_OTHER` to get a good machine player. You must use these constant names, as the machine grader will assume they are defined in your file. Further, the final test will be whether your player selects obvious good next moves.

---

Created Thu 29 May 2014 10:41 PM CEST

Last Modified Mon 16 Feb 2015 5:29 AM CET