Girls' Programming Network

Tic-Tac-Toe

Challenge Workbook

Create an unbeatable Tic-Tac-Toe computer player!

This project was created by GPN Australia for GPN sites all around Australia!

This workbook and related materials were created by tutors at:

Sydney, Canberra and Perth

Girls' Programming Network

If you see any of the following tutors don't forget to thank them!!

Writers

Renee Noble
Jin Cong
Rowena Stewart
Kimberley Apted

Testers

Kimberley Apted
Courtney Ross

Part 0: Setting up

★ Before you start this workbook ★

Make sure you have already finished:

- The Base Workbook on making a two player tic-tac-toe game
- Extension 9: It's a tie

Task 0.1: Copy paste!

We might want to play our 2 player game later. Copy your code into a new file where we'll edit it to make our computer player version.

1. Make a new file and copy your code from your Base game and Extension 9 into a new file.

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 1:

☐ You should have a new file containing your code for the basic 2 player Tic-Tac-Toe with extension 9.

Part 1: Adding a basic computer player

Task 1.1: Set your mark

In our game, the computer player will always be X and the human player will always be O. We're going to store this setting so we don't have to remember which is which when coding human or computer player behaviour.

1. At the very top of your code, add two variables called `comp_symbol` and `human_symbol`, and give them the values `"X"` and `"O"` respectively.

Task 1.2: What spaces are free?

We'll need to know what places are available for the computer to play in.

Your board is represented by indexes shown to the right:

0	1	2
3	4	5
6	7	8

We'll create a list in our code shortly. It will store all the squares that are still empty.

1. List the available squares for the last examples?

X	O	X

free_squares =
[3,4,5,6,7,8]

O		
	X	
		O

free_squares =
[_____]

X	X	
O	O	

free_squares =
[_____]

Task 1.3: List the free squares

Let's make our code calculate the list of free squares from the state of the board! We'll use this to let the computer choose a move that is available.

1. Go to your code in the game loop, before you ask the user for input.
2. Create an empty list `free_squares` to store the empty squares.
3. Loop through the `board` list and add the indexes for each square that still has `" "` instead of a symbol.

Hint: A fun way to loop

One way to do this using the enumerate function:

```
free_squares = []
for index, current_square in enumerate(board):
    if current_square == " ":
        free_squares.append(index)
```

Task 1.4: Let the computer choose

Find the bit of your code where the player is asked to choose a square.

1. Add an `if` statement to make sure you are only asking for input when the current `symbol` matches `human_symbol`.
2. Then, use the `else` part of this `if` statement to let the computer choose a random square from the list of free squares.

Hint: Choose a random item from a list

We can use random choice like this:

```
import random
fruit_of_the_day = random.choice(["apple", "banana"])
```

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 2:

- ☐ You are storing the symbols for computer and human players.
- ☐ On each turn, you are checking whose turn it is.
- ☐ On each turn, you are calculating which squares are still free on the board and storing this in a list.
- ☐ If it is the computer's turn, you have selected a random place from the list of free places.
- ☐ You have played it and it's working.

★ Bonus 1.5: Stop the silly humans ★

At the moment the human can enter a square that doesn't exist, like 99, and break the game! Or they can go in a square that's already filled in. Let's fix it to prevent cheating and breaking!

Inside the `if` statement where you ask for the human move, check that it's allowed!

1. Go to below where you ask the human for their move.
2. Add an `if` statement to check if that square **is not** in free squares.
3. If it's not free, print out that the square is not allowed.
4. On the next line, add a `continue` statement to make the game skip back to the start of the loop so the human can try again.

Part 2: Making It Modular

To build a better computer player we're going to need more code! It's going to get messy if we leave it all in the main game loop. Let's start tidying it up by making some functions.

Task 2.1: Finding the free squares

1. Write a function called **`get_free_squares`**, it will have one argument which is the board list. As output, it will return a list of indexes for the empty squares.

2. Move your existing code for calculating the list of free squares into this function and delete it from its previous location.

3. Finally, write a statement to **`return`** the list of free squares as the last line of the function.

Don't worry about not calling the function yet. We'll call it later!

Task 2.2: Getting the computer's move

1. Create a function called **`get_comp_move`** to decide where the computer will move next. This function will have one input which is a list called **`board`**.

2. Before we select the computer's move, we need to know which squares are free. Luckily, we just made a function to do this for us! Call the **`get_free_squares`** function and store the result in a variable called **`free_squares`**.

3. Now it's time to get the computer's move. We already wrote the code for this in Task 1.4 so all we need to do is copy that code to the next line and delete it from our main loop.

4. On the last line of your function, make sure you have a return statement to output our selected move.

5. In the main loop, call the **`get_comp_move`** function and store the result in the variable for our next move. This variable should be the same one for the human move and the computer move.

Task 2.3: Switching turns

While we're tidying up let's also make a function for changing to the opposite player symbol; from naughts to crosses and vice versa.

1. Make a function called `get_opposite_symbol`, it should take in one input which is the symbol for the last player.

2. It should return the opposite symbol from the given argument, for example `get_opposite_symbol("X")` should return `"O"`.

3. In your game loop call your function in the place where you switched whose turn was next.
   ```
   symbol = get_opposite_symbol(symbol)
   ```

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 3:

- ☐ You have a function that gives you a list of free squares on the board.
- ☐ You have a function that gives you the next square for the computer's move.
- ☐ You have a function to switch to the other player at the end of each turn.
- ☐ You can still play a game of tic-tac-toe against the computer.

Part 3: Winning on the next turn

At the moment our computer player just makes random moves. Let's make the computer a bit more competitive.

If the computer can win on the next move, it should make that move instead of a random move.

Task 3.1: Check all possible moves

Inside our **`get_comp_move`** function, we want to check all the free squares on the board to see if we can win by going there.

1. Make an empty list call **`winning_moves`** that will store any squares that will win the game.

2. Create a `for` loop that goes through all **`free_squares`** on the board.

3. For the current square in the loop, update your **`board`** with the computer symbol and test if there is a winner. You can use you **`check_winner`** function.

4. If your **`check_winner`** function has returned **`True`** then it must have been a winning move for the computer! Add the current square to the list of winning moves.

5. Remove the move you just checked from your **`board`** to reset for the next iteration.

Hint: Trying and undoing a move

You can remove a move that the computer tries out by setting the square back to `" "`:

```
# trying out a move...
board[0] = "X"
outcome = check_winner(board)
# ...undoing the move
board[0] = " "
```

Task 3.2: Play to win

Once the computer has checked all the free squares, we should check if it found any winning moves. If it has found any winning moves, we will get the computer to play one of those.

1. Check if **`winning_moves`** contains any squares.

2. If **`winning_moves`** has 1 or more items, **`get_comp_move`** should return the first item in the list.

3. If **`winning_moves`** has 0 items, **`get_comp_move`** should return a random choice as before.

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 4:

- ☐ You have a list in which to store all the winning moves.
- ☐ You check all free squares for a winning move for the computer.
- ☐ The computer will choose a winning move instead of a random move, if there is at least one to take.

★ BONUS 3.3: Test it out! ★

Your computer should be playing smarter now, but how do you know?

1. Try playing it a few times and setting it up to win. Does it win? Which human moves make it more likely for the computer to make good moves?

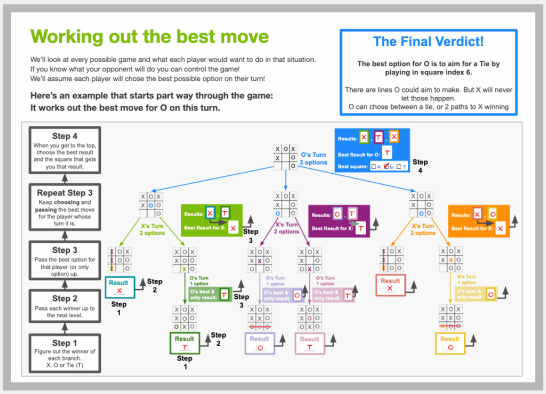
Part 4: The best move for me!

We're going to need a function that tries all possible ways the game could go, and figures out what the player should do on that turn to make the game go as well as possible for them.

Task 4.1: Predicting the future of the game

Before we can code how to predict the game, we need to be able to do the algorithm by hand!

1. Complete the A3 worksheet hand out to predict the last 3 moves of a tic-tac-toe game. Figure out what the best move is for this turn.
   Go to your A3 worksheet now!
   Read the example and complete the "Do it yourself" page!



2. Think about how we could do this not just for the last 3 moves, but from any point in the game!

Task 4.2: Picking a winner… if you can

In the last activity you saw that each stage we got a list of results that we would get from all potential games, based on what you and your opponent would do.

We'll write a function that will help us with making that choice. But first some examples.

1. **What's the best result for the player in these examples:**

Whose turn	Options	Best result
X	`["X", "O", "T", "O", "T", "T"]`	
X	`["O", "O", "O"]`	
O	`["T", "T", "T", "X", "T"]`	

O	`["T", "T", "O", "T", "T"]`	

Task 4.3: My best outcome

Let's write a function that tells us our best outcome from a list of possible outcomes.

1. Create a function called **`best_outcome_for_symbol`**.

2. Your function should take in two arguments:
    1. The symbol whose turn it is; and
    2. A list containing some number of `"X"`, `"O"` and `"T"` strings. These represent the different outcomes from different possible game moves.

3. Your function should loop through the list of outcomes and decide what is the best outcome for the symbol whose turn it currently is. The ranking of best is:
    1. **Best** - The current symbol (they want to win of course)
    2. **Middle** - `"T"` for tie. If you can't win go for the tie!
    3. **Worst** - The opponent's symbol. Only if there's nothing else!

4. Return the best option in the list.

Hint: Use your functions

The opponent's symbol will change depending on the player symbol argument given to **`best_outcome_for_symbol`**, and we created a function that does exactly that in Task 2.3!

Hint: Returning early

One of the nice features of a function is the `return` statement. Each function can have multiple `return` statements, and the function will finish once the *first* one is reached.

Check out this example on calculating ticket prices.
Entry is free for kids under 15 and members. Concession is $8. Full price is $15.

One `return` statement:

```
def admission(age, member, concession):
    result = 15

    if age <= 15 or member:
        result = 0
    elif concession:
        result = 8

    return result
```

Multiple `return` statements:

```
def admission(age, member, concession):
    if age <= 15 or member:
        return 0
    elif concession:
        return 8
    return 15
```

For this example, both functions do exactly the same thing, so you could choose whichever method makes more sense to you.

Task 4.4: Check your work!

Our game won't use the `best_outcome_for_symbol` function until a later step, but there's no reason why you can't test it now.

1. Go to the bottom of your code and call your function on the following examples:
    a. Best outcome for X with the list `["X", "O", "X", "T", "T"]`
    b. Best outcome for O with the list `["X", "X", "T", "T"]`
    c. Best outcome for X with the list `["T", "T"]`
    d. Best outcome for O with the list `["X"]`

2. You can delete the print statements once you're happy your function is working correctly.

Hint: Call your own function

Here's an example of how you can test a function with some simple print statements:

```
def add(num1, num2):
    return num1 + num2

print("Expected: 6, Actual:", add(2, 4))
print("Expected: 1110, Actual:", add(243, 867))
```

The printed output will look like this:

```
Expected: 6, Actual: 6
Expected: 1110, Actual: 1110
```

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 5:

- ☐ You've worked out the best options for the game by hand.
- ☐ You've written a function that finds the best outcome in the list for the symbol whose turn it is.
- ☐ You've checked that your function works on some examples

Part 5: The next, next, next moves

The best way to choose a computer move is to work out all the possible games that can happen from this place on the board. To do this we'll look at what all the next possible moves are. Then look at all the possible moves are for those possible games. And look at the possibilities after that! All the way till the end of the game! We'll use recursion!

Task 5.1: Starting the never ending function

1. Start by creating a new function called **`get_move_outcomes.`**

2. Your function should take in two arguments:
    1. The symbol whose turn it is; and
    2. The list containing the board configuration we want to test possible future moves for.

3. Inside the function use your **`get_free_squares`** function to get the list of free squares for board that was passed into the function.

Task 5.2: All out of spaces

If there's no free squares left it means all 9 spots are already full. We can't go and the game is a tie. The result of a full board is always a tie!

1. Inside the function create check if there are no items in **`free_squares`**.

2. If there's no spaces left on the board return `"T"` from the function.

Hint: Base Case

This is the base case for our recursive function!

Task 5.3: Space to place

If there's still space on the board we want to see what would happen in all the possible games starting with the options for the next possible move.

1. Inside the function, after the code from Task 5.2, create an empty list to store all the results of the possible games. *We'll add things to this later once we work out how all the potential games would go if everyone played perfectly.*

2. Next, create a `for` loop that loops over all the free squares.

3. For the current square of the for loop set it to the current symbol. We're going to test the outcome of this possible game.

4. Use the **`check_winner`** function to see if this would result in the game being won.

5. Use an `if` statement to see if this results in a win, add the current symbol to the list of results if it does.

6. After the `if` statement set the current square back to `" "`.

Task 5.4: Giving back the best

1. Use the **`best_outcome_for_symbol`** function you wrote earlier to take the list of results and find the best outcome for the current symbol in the list.

2. Return the best outcome. *The symbol will always chose to go down the path that will get them the best option.*

Task 5.5: Space to place, but no way to win!

So far we've taken care of the cases where there were no more places, or if there was a place that made the current symbol win this turn. But what about if there's more places but no one wins this turn?

1. Add an `else` statement to the if statement you wrote in Task 5.3.

2. Use your **`get_opposite_symbol`** function to get the symbol that will play next.

3. Do the recursion!! This is where we call the **`get_move_outcomes`** inside of itself. Pass in the current configuration of the board, and the symbol that is going to go next that you got in the last step. Store the result.

4. Add the result to the list of results.

Task 5.6: Test it out

1. Think up a some different board configurations.
    - Some that are all full.
    - Some with only 1 spot left.
    - Some with a move that wins next turn.

2. Test your `get_move_outcomes` function, for either `"X"`, or `"O"`. See if the potential results are what you expect by printing out all the results.

3. See if it returns the best option for the symbol you passed in. Is it what you expected?

Hint: What did you expect?

Here's an example of how you can call the `get_move_outcomes` function to test it:

```
board = [" ", " ", " ", " ", " ", " ", " ", " ", " "]
print("Expected: T, Actual:", get_move_outcomes("X", board))
print("Expected: T, Actual:", get_move_outcomes("O", board))
```

Try some different values for `board` and see if your function is working as expected!

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 6:

- ☐ You have a `get_move_outcomes` function.
- ☐ Your function takes care of tied games.
- ☐ Your function takes care of winning moves.
- ☐ Your function takes care of cases where the game is not over at the end of the turn and you keep testing how the game goes using recursion.

Part 6: Computer can't be beat

Now we have a recursive function we can work out the best move for the computer to make on this turn.

Task 6.1: More to life than winning

We already keep track of the winning squares, we need to do the same for ties and loses!

1. Go back to your **`get_comp_move`** function.

2. Add 2 more empty lists at start of the **`get_comp_move`** function:
    1. A **`tied_moves`** list; and
    2. A **`losing_moves`** list.

Task 6.2: Testing all the options

When we can't win on the next move we currently make a random move. But now we have our **`get_move_outcomes`** function we can do better!

1. Go to the line in the **`get_comp_move`** function where we return a random move, and remove the random line of code.

2. Where your random line used to be, create a `for` loop that loops over all the free squares.

3. For the current square of the for loop set it to the computer's symbol.

4. Use the **`get_move_outcomes`** function to see how the game turns out based on how the *human* might play next (and how the rest of the game will go). Store the result.

5. Next, create an `if` statement to decide which list to add the square to:
    a. If the result is the computer symbol, store the square in the winning list.
    b. If it's a tie, store the square in the tied list.
    c. If it's the human symbol, store it in the losing list.

6. After the `if` statement set the current square back to `" "`.

Hint: Who's turn is it anyway?

Think carefully about the arguments you're passing into the `get_move_outcomes` function. Remember:

- In this for loop we are trying out a possible *computer* move; and
- When we call the `get_move_outcomes` function we want to find the best outcome arising from the next *human* move.

Task 6.3: Choosing a favourite

We have now looped through all the possible squares to play in and calculated how every possible game can go!

We have a list of all the squares that will result in wins, losses and ties for the computer (assuming the human plays the best moves possible). Time to pick our favourite!

1. If there's any winning games, return one of those.
2. If not, return a tied game if there is one.
3. Finally, if there's no other choice, return a lost game.

Task 6.4: Play time

Test your game, see if the computer can keep up!

1. Does the computer player win if you "let it" by trying to lose? What happens if you go in squares 1, 3, 5, 6 on your turns? If you can get it to tie, make sure that your computer player is going for any immediate wins it can!
2. We said that this would make the computer player unbeatable. What do you think? Can you beat it??

☑ CHECKPOINT ☑

If you can tick all of these off you are done!

- ☐ You have lists for storing squares that you can play in that will make you win, lose and tie.
- ☐ You call you recursive `get_move_outcomes` function.
- ☐ Your return a square from the list of winning moves if there is one. If not, return a tie. As a last resort, play a losing move!
- ☐ You played against your computer player!

★ BONUS 6.5: Try using a dictionary! ★

We didn't cover dictionaries in the lectures for this workshop, but if you've used them before or think you can figure them out from the cheat sheet, try this bonus task!

1. Replace the three lists `winning_moves`, `tied_moves`, and `losing_moves` with a dictionary containing an empty list mapping to each of the three outcomes:
   ```
   outcomes = {comp_symbol: [], "T": [], human_symbol: []}
   ```

2. Anywhere you're adding to the three lists you've just removed, you can add the move to the corresponding list like this:
   ```
   outcomes[result].append(square)
   ```

3. If you haven't already done so, you can clean up your `if` statement from Task 6.2. Since all three options are repeating the same code, you don't need it anymore!

4. Now update the part of the function where we return the best move.
   - You can check the number of winning moves like this:
     ```
     len(outcomes[comp_symbol])
     ```
   - And you can get the first winning move like this:
     ```
     outcomes[comp_symbol][0]
     ```