Computing

Lesson 10: 2D Lists Challenge

Programming Part 5: Strings and Lists

Rebecca Franks

 Materials from the Teach Computing Curriculum created by the National Centre for Computing Education

Decompose noughts and crosses

Task

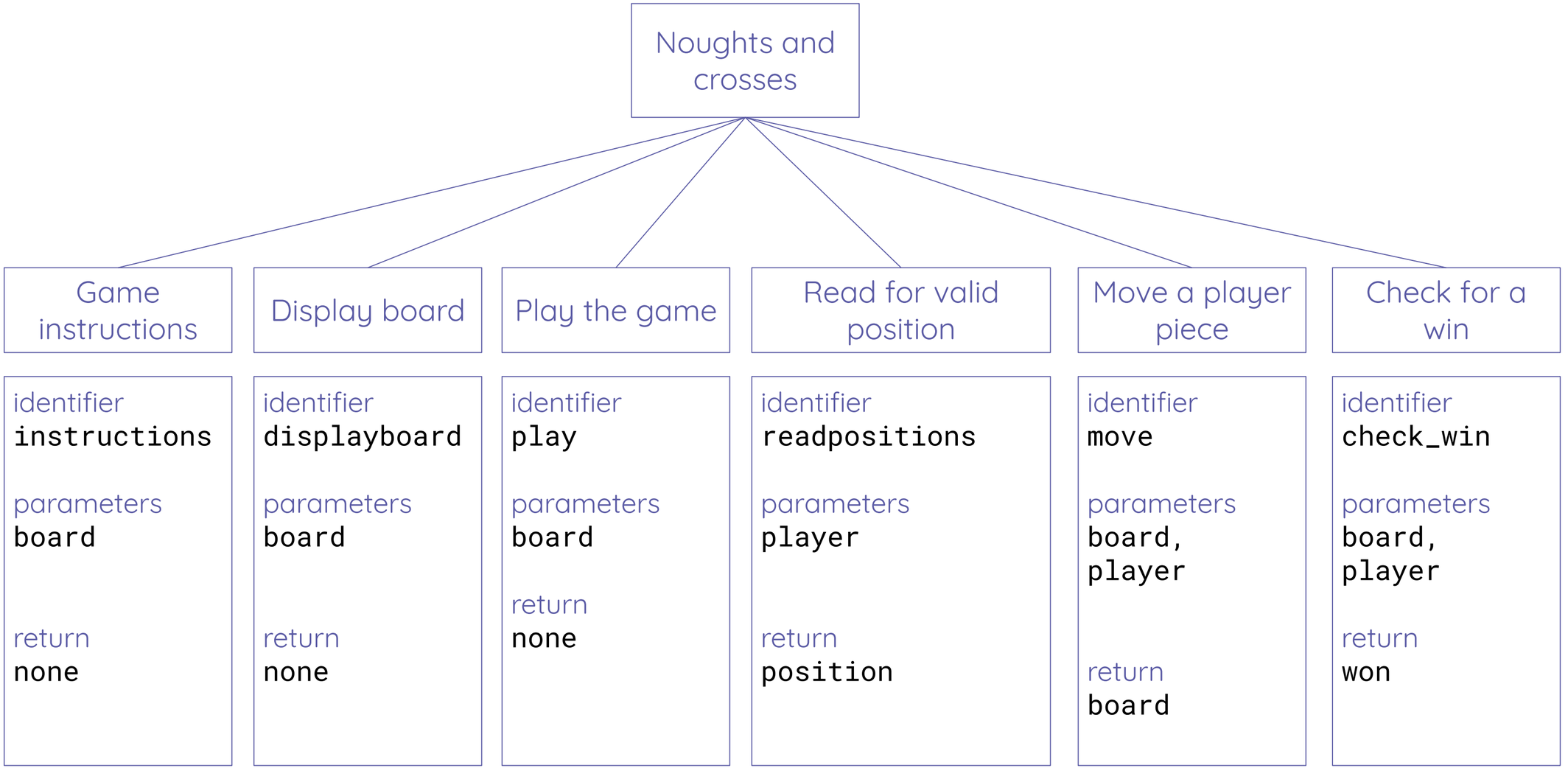
Think about all of the steps that are required for a player to get to a winning position in noughts and crosses.

List them all in the box provided.

Don’t be too concerned over a specific order at this stage.

Getting started

Noughts and crosses structure chart



Task: Reading the board

Although there are lots of ways to set up a noughts and crosses board, you will be using a 2 Dimensional list. Initially, this will look like this:

```
board = [[" ", " ", " "],
         [" ", " ", " "],
         [" ", " ", " "]]
```

Each element in this 2 Dimensional list can be accessed using a variation of this code snippet:

```
    # r  c
board[0][0]
```

Task: Reading the board

The values in the square brackets on the previous slide are replaced with the row and column indexes. These are represented in the grid below:

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

A procedure will then be used to display the board based on the data from the 2D list. The start code for a noughts and crosses board can be viewed below or by accessing this link (**ncce.io/ks4-boardformat**):

```
def displayboard(board):
    print(" ", board[0][0], "|", board[0][1], "|", board[0][2])
    print(" ———┼———┼———")
```


Task: The board

Step 1

Use the information on page one to help you create a completed noughts and crosses board using the 2D list and the incomplete procedure.

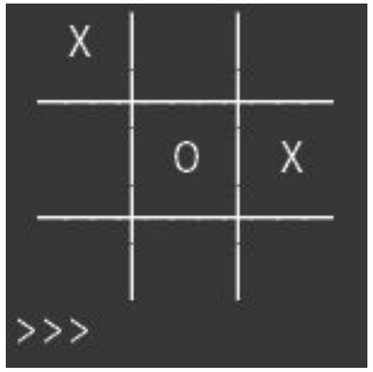
Step 2

Call the procedure and see how the board is displayed on the screen. You might want to try different board layouts to personalise it further.

Step 3

Modify the board 2D list by adding in O's and X's and check if your display board still looks like a workable board for your game.

An example output might be:



Task: Checking for a win

Step 1

Work out all of the possible ways that you can win at noughts and crosses. Write the total number down.

Tip: You might be able to work this out in your head but if not, get some scrap paper and keep drawing boards until you have drawn all of the possibilities and then count them.

Task: Checking for a win

Step 2

Using the grid below as a guide. Write down all of the combinations for a noughts and crosses win. The first one has been done for you.

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

1. [0][0] [0][1] [0][2]

Task: Checking for a win

Step 3

A function has been started that will check if the current 2D list contains a winning combination. Here is the start code:

```
def check_win(board, player):
    won = False
    if board[0][0] == player and board[0][1] == player and board[0][2] == player:
        won = True
```

Complete this function using the combinations that you provided in **Step 2.**

Task: Checking for a win

Step 4

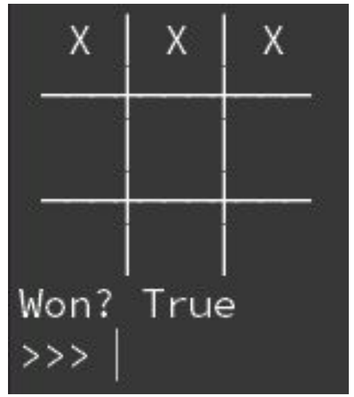
Test your function by editing your board so that X wins for each possible combination. For example, to test the first combination, set your board like this:

```
board = [["X", "X", "X"],
         [" ", " ", " "],
         [" ", " ", " "]]
```

Tip: you will need to create a variable for the player to ensure that this argument is passed to the check_win function. Make sure that the player holds the value X or your program will not work.

Tip: to make sure that your code works you will need to print won and see if it is True when a winning combination is used.

Here is an example of what the output for the first combination might look like:



Complete noughts and crosses

Task: Game instructions

Step 1

A procedure needs to be created to provide instructions to the user at the beginning of the game.

During game play the user will see the board with the numbers 1 to 9 to show the locations where they can place their X or O. They enter the number and then their X or O will be placed on the board. The first player will be X and the second player will be O.

Create a procedure called instructions that provides these instructions to the user.

Task: Game instructions

Step 2

Test your program by calling the procedure to see how user friendly it is. You might decide that it needs to appear more slowly and introduce some delays to the program to slow the instructions down.

An example output can be seen on the next slide.

Task: Game instructions

```
Welcome to noughts and crosses
------------------------------
Instructions:

This is a 2 player game
The first player will be noughts
The second player will be crosses
The game is presented in a grid...

 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9

To choose a position for your piece, enter the location number
```

Task: Move a player piece

Step 1

A function needs to be created that moves the player piece onto the board. Define a function using the following interface:

Identifier: `move`

Parameters: `board, player`

Return values: `board`

Tip: all subroutines should be defined before the main program begins.

Task: Move a player piece

Step 2

Complete the function ensuring that it:

- Asks the current player to enter their desired `position`
- Uses the `position` entered to add either a O or X to the `board` at the desired location
- Returns the `board`

Task: Move a player piece

Step 3

For testing purposes make sure that player holds the value X. Test your program by entering each number from 1 to 9 and making sure that it correctly adds an X to the correct position on the board.

An example output can be seen on the next slide.

Tip: when testing your code you can # tag out the `instructions()` procedure to make testing quicker. Just place a # before the procedure call and it will be ignored during execution.

Task: Move a player piece

```
X's turn. Where would you like to place your piece?
7
  1 | 2 | 3
 ———————————
  4 | 5 | 6
 ———————————
  X | 8 | 9
Won? False
>>> |
```


Task: Play

Step 1

A procedure needs to be created that will run the playing of the game. This will be built up over several tasks. Define a procedure using the following interface:

Identifier: `play`
Parameters: `board`
Return values: `none`

Step 2

At this stage the play procedure should:

- Only work for X
- Continue to ask the user to enter a position until X wins
- End the program when X wins

Complete the play procedure so that it performs the above tasks.

Task: Play

Step 3

Test your program for all winning moves that X can have to make sure that it works correctly. Remember to call the procedure at the bottom of your program.

An example output can be seen on the next slide.

Task: Play

```
X's turn. Where would you like to place your piece?
1
  X  |  2  |  3
 ____|_____|____
  4  |  5  |  6
 ____|_____|____
  7  |  8  |  9
Won? False
X's turn. Where would you like to place your piece?
4
  X  |  2  |  3
 ____|_____|____
  X  |  5  |  6
 ____|_____|____
  7  |  8  |  9
Won? False
X's turn. Where would you like to place your piece?
7
  X  |  2  |  3
 ____|_____|____
  X  |  5  |  6
 ____|_____|____
  X  |  8  |  9
Won? True
>>>
```


Task: Switching between players

Step 1

For this task you will still be working in the `play` procedure.

Introduce a new variable called `currentplayer` and hold the value X. This should be declared outside of the `while` loop initially.

Step 2

Introduce selection to your `while` loop so that if the current player is X it switches to O and if the current player is O it switches to X.

Task: Switching between players

Step 3

Ensure that player now holds `currentplayer` instead of X.

Step 4

Test your program to see if it switches between players during game play.

An example output can be seen on the next slide.

Task: Switching between players

```
X's turn. Where would you like to place your piece?
1
  X  |  2  |  3
 ————————————————
  4  |  5  |  6
 ————————————————
  7  |  8  |  9
O's turn. Where would you like to place your piece?
2
  X  |  O  |  3
 ————————————————
  4  |  5  |  6
 ————————————————
  7  |  8  |  9
X's turn. Where would you like to place your piece?
4
  X  |  O  |  3
 ————————————————
  X  |  5  |  6
 ————————————————
  7  |  8  |  9
O's turn. Where would you like to place your piece?
```

Task: Announce the winner!

Step 1

Introduce a new variable called `winner`. The new variable should hold the `currentplayer` when a win happens.

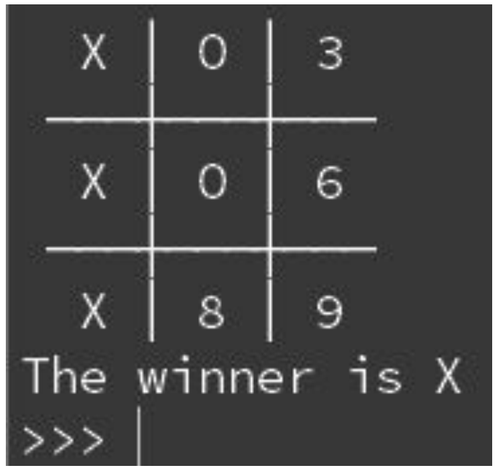
Step 2

When a player has won, create an output that states which player has won the game.

Step 3

Test the game to make sure that the correct winner is announced at the end of the game.

An example output can be seen below:



Task: Add validation checks

Step 1

Create a new function that will make sure that a string number between 1 and 9 has been entered by the player in the move function. The function should have the following interface:

Identifier: `checkpositions`

Parameters: `player`

Return values: `position`

Task: Add validation checks

Step 2

The `checkpositions` function should:

- Ask which location the piece should be placed
- Check that the location is a string number between 1 and 9
- If the value is entered incorrectly then it should continue to ask for the correct input
- Return the valid value back to the move function to the position variable

Tip: you need to make sure that the user is entering a string character and not an integer. You can use a list to hold the valid values and check the input against the list.

Task: Add validation checks

Step 3

Test that your new function works correctly by entering correct and incorrect data.

An example output is shown - >

Step 4

Ensure that appropriate validation checks have been used at any other input in the game.

```
x's turn. Where would you like to place your piece?
f
You must enter a number from 1 to 9
3
  1  |  2  |  x
 ____|_____|____
  4  |  5  |  6
 ____|_____|____
  7  |  8  |  9
o's turn. Where would you like to place your piece?
h
You must enter a number from 1 to 9
2
  1  |  o  |  x
 ____|_____|____
  4  |  5  |  6
 ____|_____|____
  7  |  8  |  9
x's turn. Where would you like to place your piece?
```

Test your program

Introduction

Copy and complete the testing table on the next slide to ensure that your program works correctly.

Remember to include tests that use **erroneous, boundary** and **normal** data.

Testing table

Test number	Test description	Input (if required)	Expected output	Actual output	If the test was unsuccessful. How was it fixed?