Tic-Tac-Toe Program

by Justin Johnsen

WHAT'S COVERED

In this lesson, you will learn to create a program that uses lists and applies earlier concepts. Specifically, this lesson covers:

1. Review of Data Collections

Let's quickly review the different data collection types that we looked at and consider when it would be optimal to use each option.

Lists

Lists are defined by square brackets.

```
myNums = [6, 4, 1, 25, 90]
myNames = ["Brighticorn", "Balloonicorn"]
```

When to use lists:

- When the data is changing over time. Since lists are mutable, we can change the elements at any time.
- When data is homogenous (groupings of similar elements) such as a group of names or a group of prices.
- If you'd like to repeat the same steps over a group of elements in a data collection type.
- When you're keeping track of one group such as the number of elements in a data collection type or the order of the elements in a data collection type.

Sets

Sets are defined by curly brackets.

```
myNums = {6, 4, 1, 25, 90}
myNames = {"Brighticorn", "Balloonicorn"}
```

When to use sets:

- When data is meant to be constant and not to change. Remember, sets are immutable. We can add or remove elements but cannot change or replace an element once the set is created.
- When data is meant to be unique. Sets do not allow duplicate values.
- If we need to find the elements that are in two sets or the complete list of elements in both sets without duplication.

Tuples

Tuples are defined by round brackets.

```
myEmployee = ("Brighticorn", "sparkles", 1)
days_of_week = ("Mon", "Tues", "Wed", "Thurs", "Fri", "Sat", "Sun")
```

When to use tuples:

- When data is NOT changing over time. Remember, tuples are completely unchangeable, meaning that we cannot change, add, or remove elements after the tuple has been created.
- When data is heterogeneous (diverse), meaning you may not have the same types of elements in the iterable object, such as having a name as one element and age as another element.
- When repeating the same code over each piece of data in a tuple.
- When we need order. If we recall, lists and tuples are ordered, so elements can be found by index positions.

Dictionaries

Dictionaries are defined by curly brackets, with each element using a key/pair value separated by a colon.

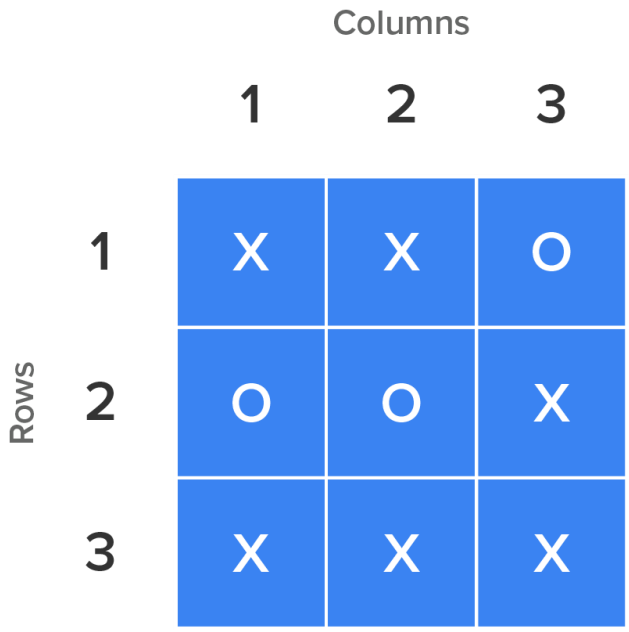
```
words_by_letter = {'a': ['apple', 'ant'], 'b': ['banana', 'bundt']}
```

When to use dictionaries:

- When you need to associate two or more pieces of data.
- When you're keeping count of multiple things like the number of words that are used in a paragraph or the number of times a number is entered.
- When you're keeping track of multiple things you've seen, organized by some category such as the number of blue items that appear or the number of appetizers on a menu.

2. The Start of Tic-Tac-Toe

Let's first start by thinking about a Tic-Tac-Toe board. If we look at it, it has 3 rows and each row has 3 squares (columns).



Each of the squares can be empty or contain an X or an O. The position of each square never changes, so the order does matter. The contents of each of the squares can change, so it is mutable. Given what we know about the different data collection types, we want to think about what is the best collection type to represent a row in a Tic-Tac-Toe board. Keep in mind that a row has 3 squares; each square contains information that can change; and the order matters. An element in a set and tuple cannot change once set up, so those are out. A dictionary takes a key:pair of values but we only need an X or O, so that's not going to work as we intended.

Given that criteria, a list would be the best choice.

We can model a row like this.

EXAMPLE

```
row0 = ["-", "-", "-"]
```

Given that a Tic-Tac-Toe board has 3 rows, we can now define the rows like this.

EXAMPLE

```
row0 = ["-", "-", "-"]
row1 = ["-", "-", "-"]
row2 = ["-", "-", "-"]
```

Ok, so using a list makes sense for each row. Next, let's think about the Tic-Tac-Toe board. Let's consider what data collection type would be best to represent the board data. A board has 3 rows and the position of each row cannot change, as the order does matter, so a set is out. With a tuple, we cannot change the data elements once they are set. Since we want the players to make changes, this would not work. Dictionaries could work but would add further complexity when we can use a list for the same purpose. So in this case too, the use of a list is the best choice.

We could model our board like this:

EXAMPLE

```
board = [row0, row1, row2]
```

Step 1: Board Creation

Instead of this structure, we can also use nested lists to define it, which would have the same result. It could be declared like the following:

TRY IT

Directions: Enter the following code into the IDE to start the build process for the Tic-Tac-Toe board.

Note: Being that the nested lists are within an overall list, the indentation doesn't matter as much. But ideally, you want to have it visually aligned so you as the developer can see what it should look like. All we did below is select the Enter key after each comma and then hit the Tab key until the inner lists were vertically aligned on the IDE code editor.

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

We use this nested list as it makes the data easier to access and helps reflect the real-world behavior of the Tic-Tac-Toe board. We have three columns with three elements in each row, which resembles the board.

For now, we'll make this game a bit easier by allowing players to alternate and choose their position. Once they have selected the position, they will be able to place the X or O in that position as long as the position still has a "-" (hyphen). We will use the hyphen as an empty position.

At this point, we will not introduce a checking function to determine if anyone has won. For now, this board is just meant to be a replacement for drawing out the board on paper. As we work through this, try to think about what ways we can improve this process going forward.

First, we'll start by setting up the board and printing it out.

TRY IT

Directions: Add the `print()` function to see how this board looks on output.

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

print(board)
```

The output shows the start of the board.

```
[['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]
```

Does that look like a Tic-Tac-Toe board? Not a traditional board, right? Even though we indented the list in the IDE's editor window, that does not affect the output.

Although this is our board, it would be a bit difficult to tell who won visually. To help with that, we want to output each row separately.

TRY IT

Directions: Now try the code below with each row being separately outputted to the screen.

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

print(board[0])
print(board[1])
print(board[2])
```

The output shows the initial Tic-Tac-Toe board.

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
```

That's a bit better, but continue to think about ways that this could be improved on as we move into the next step.

Step 2: User Input

Next, we can prompt the user to enter in a value between 1-3 for the column and then again (1-3) for the row.

Note: Although the list starts with an indexing of 0 and goes to 2 (3 elements), that would be confusing to the end-user. That's why we will let them select between 1 and 3.

TRY IT

Directions: Enter the code to output directions to the X user for their selection.

```
col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
```

Would we need to reduce the value that was entered by 1? Correct, we need to subtract 1 from the input.

TRY IT

Directions: Enter the code below to subtract 1 from the user input selection.

```
col -= 1
row -= 1
```

We can now place an X in that position and output the board again.

TRY IT

Directions: Enter the following code to place the “X” in the row and column that has been identified.

```
board[row][col] = "X"
print(board[0])
print(board[1])
print(board[2])
```

Let’s test what we have so far.

TRY IT

Directions: Make sure your code looks like the code snippet below, then run the program so far.

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

print(board[0])
print(board[1])
print(board[2])

col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
col -= 1
row -= 1

board[row][col] = "X"
```

```
print(board[0])
print(board[1])
print(board[2])
```

The output shows the first move.

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
X player, select a column 1-3: 1
X player, select a row 1-3: 2
['-', '-', '-']
['X', '-', '-']
['-', '-', '-']
```

All looks good so far. Did you (as player X) get your X in the column/row you selected?

Next, we have to prompt the player playing O to go next. We can use the same approach.

TRY IT

Directions: Try adding the code for the second player to enter their position.

```
col = int(input("O player, select a column 1-3: "))
row = int(input("O player, select a row 1-3: "))
col -= 1
row -= 1

board[row][col] = "O"
print(board[0])
print(board[1])
print(board[2])
```

While entering this code, you may have noticed or questioned a possible scenario. What if player O enters a position that player X already chose? We don't want to overwrite that position. Rather, we'll just say that the user ends up forfeiting their turn, and the game moves on to the next player. This is certainly not ideal but it is important to think about as we move on, since ideally, we would want the user to keep selecting a spot until they choose one that is available. Look for that type of correction in future lessons. For now, we'll just check if the spot is empty by looking for the "-" (hyphen) character. If the selection is the "-" character, we'll replace it with an O and if it wasn't, we'll let the user know that spot was already taken and the other player selects.

TRY IT

Directions: Enter the code to check if the space has been filled or not.

```
if board[row][col] == '-':
  board[row][col] = "O"
else:
  print("Oops, that spot was already taken. ")
```

Step 3: Wash, Rinse, and Repeat

Finally, we want to repeat this structure 7 more times, as we have 2 entries already and have 9 total boxes to fill. Since this can get a bit busy with the number of times we repeat, we'll add in some comments to indicate each move.

TRY IT

Directions: Enter the next 7 attempts to complete the Tic-Tac-Toe game. See the comments to separate each attempt.

```
#the following code creates the board
board = [ ["-", "-", "-"],
          ["-", "-", "-"],
          ["-", "-", "-"] ]

print(board[0])
print(board[1])
print(board[2])

#this is player X's first move
col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
col -= 1
row -= 1

board[row][col] = "X"
print(board[0])
print(board[1])
print(board[2])

#this is player O's first move
col = int(input("O player, select a column 1-3: "))
row = int(input("O player, select a row 1-3: "))
col -= 1
row -= 1
```

```
if board[row][col] == '-':
  board[row][col] = "O"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player X’s second move
col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
col -= 1
row -= 1

if board[row][col] == '-':
  board[row][col] = "X"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player O’s second move
col = int(input("O player, select a column 1-3: "))
row = int(input("O player, select a row 1-3: "))
col -= 1
row -= 1

if board[row][col] == '-':
  board[row][col] = "O"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player X’s third move
col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
col -= 1
row -= 1
```

```
if board[row][col] == '-':
  board[row][col] = "X"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player O’s third move
col = int(input("O player, select a column 1-3: "))
row = int(input("O player, select a row 1-3: "))
col -= 1
row -= 1

if board[row][col] == '-':
  board[row][col] = "O"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player X’s fourth move
col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
col -= 1
row -= 1

if board[row][col] == '-':
  board[row][col] = "X"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player O’s fourth move
col = int(input("O player, select a column 1-3: "))
row = int(input("O player, select a row 1-3: "))
col -= 1
row -= 1
```

```
if board[row][col] == '-':
  board[row][col] = "O"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])

#this is player X's fifth move
col = int(input("X player, select a column 1-3: "))
row = int(input("X player, select a row 1-3: "))
col -= 1
row -= 1

if board[row][col] == '-':
  board[row][col] = "X"
else:
  print("Oops, that spot was already taken. ")
print(board[0])
print(board[1])
print(board[2])
```

We're now ready to test it to see how we did!

TRY IT

Directions: Run the Tic-Tac-Toe Program and enter choices for players X and O.

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
X player, select a column 1-3: 1
X player, select a row 1-3: 1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
O player, select a column 1-3: 1
O player, select a row 1-3: 2
['X', '-', '-']
['O', '-', '-']
['-', '-', '-']
X player, select a column 1-3: 1
X player, select a row 1-3: 3
```

```
['X', '-', '-']
['O', '-', '-']
['X', '-', '-']
O player, select a column 1-3: 2
O player, select a row 1-3: 1
['X', 'O', '-']
['O', '-', '-']
['X', '-', '-']
X player, select a column 1-3: 2
X player, select a row 1-3: 2
['X', 'O', '-']
['O', 'X', '-']
['X', '-', '-']
O player, select a column 1-3: 2
O player, select a row 1-3: 3
['X', 'O', '-']
['O', 'X', '-']
['X', 'O', '-']
X player, select a column 1-3: 3
X player, select a row 1-3: 2
['X', 'O', '-']
['O', 'X', 'X']
['X', 'O', '-']
O player, select a column 1-3: 3
O player, select a row 1-3: 1
['X', 'O', 'O']
['O', 'X', 'X']
['X', 'O', '-']
X player, select a column 1-3: 3
X player, select a row 1-3: 3
['X', 'O', 'O']
['O', 'X', 'X']
['X', 'O', 'X']
```

We've basically replaced the paper version with a computerized version. However, you may notice that there are a lot of repeating elements. We're missing other features like checking if a player has already won, as well as checking if the player is entering a value that is within the correct range. During future lessons on loops, we'll see how much we can trim this program down and optimize the code even further.

To see the final version of this program visit **Sophia's Python code page**

SUMMARY

We started this lesson with a **review of data collection types**. Based on the properties of the data collection types, we determined that the list type was the best match for our **Tic-Tac-Toe program** since we know that the elements will need to change and the order of the elements is important. We designed and implemented the board of the program using nested lists and structured it to resemble a real-world Tic-Tac-Toe board. We programmed both players' input and sent the results to the screen. This program does not handle incorrect player input like choosing an element that is already taken, or col/row selections outside the size of the board. We will come back to this program in the upcoming lessons to optimize the code.

Best of luck in your learning!