21 Tic-Tac-Toe: Exploring state

One of my favorite movies is the 1983 release *War Games* starring Matthew Broderick, whose character, David, plays a young hacker who enjoys cracking into computer systems ranging from his school's grade book to a Pentagon server that has the potential to launch intercontinental ballistic missiles. Central to the plot is the game of Tic-Tac-Toe, a game so simple that it usually ends in a draw between the two players.

In the movie, David engages Joshua, an artificial intelligence (AI) agent, who is capable of playing lots of nice games like chess. David would rather play the game Global Thermonuclear War with Joshua. Eventually David realizes that Joshua is using the simulation of a war game to trick the US military into initiating a nuclear first strike against the Soviet Union. Understanding the mutually assured destruction (MAD) doctrine, David asks Joshua to play himself at Tic-Tac-Toe so that he can explore the futility of games that can never result in victory. After hundreds or thousands of rounds all ending in draws, Joshua concludes that "the only winning move is not to play," at which point Joshua stops trying to destroy the Earth and suggests instead that they could play "a nice game of chess."

I assume you already know the game of Tic-Tac-Toe, but we'll review briefly in case your childhood missed countless games of this with your friends. The game starts out with a 3-by-3 square grid. There are two players who take turns marking first X and then O in the cells. A player wins by placing their mark in any three squares in a straight line, horizontally, vertically, or diagonally. This is usually impossible, as each player will generally use their moves to block a potential win by their opponent.

We will spend the last two chapters writing Tic-Tac-Toe. We will explore ideas for representing and tracking program *state*, which is a way of thinking about how the pieces of a program change over time. For instance, we'll start off with a blank board, and the first player to go is X. Play alternates between the X and O, and after each round two cells on the board will have been taken by the two players. We'll need to keep track of these moves and more, so that, at any moment, we always know the state of the game.

If you recall, the hidden state of the `random` module proved to be a problem in chapter 20, where an early solution we explored produced inconsistent results depending on the order of the operations that used the module. In this exercise, we're going to think about ways to make the state of our game, and any changes to it, explicit.

In this chapter, we'll write a program that plays just one turn of the game; then in the next chapter we'll expand the program to handle a full game. This version of the program will be given a string that represents the state of the playing board at any time during a game. The default is the empty board at the beginning of the game, before either player has made a move. The program may also be given one move to add to that board. It will print a picture of the board and report if there is a winner after making the move.

For this program, we need to track at least two ideas in our state:

- The board, identifying which player has marked which squares of the grid
- The winner, if there is one

For the next version, we'll write an interactive version of the game where we will need to track and update several more items in the state through a complete game of Tic-Tac-Toe.

In this exercise, you will

- Consider how to use elements like strings and lists to represent aspects of a program's state
- Enforce the rules of a game in code, such as preventing a player from playing in a cell that has already been taken
- Use a regular expression to validate the initial board
- Use `and` and `or` to reduce combinations of Boolean values to a single value
- Use lists of lists to find a winning board
- Use the `enumerate()` function to iterate a `list` with the index and value

21.1 Writing tictactoe.py

You will create a program called tictactoe.py in the 21_tictactoe directory. As usual, I would recommend you start the program using new.py or template.py. Let's discuss the parameters for the program.

The initial state of the board will come from a `-b` or `--board` option that describes which cells are occupied by which players. Since there are nine cells, we'll use a string that is nine characters long, composed only of the characters `X` and `O`, or the period (`.`) to indicate that the cell is open. The default board will be a string of nine dots. When you display the board, you will either display the player's mark in a cell or the cell's number, from one to nine. In the next version of the game, this number will be used by the player to identify a cell for their move. As there is no winner for the default board, the program should print "No winner":

```
$ ./tictactoe.py
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
No winner.
```

The `--board` option will describe which cells should be marked for which player, where the positions in the string describe the different cells, ascending from 1 to 9. In the string `X.O..O..X`, the positions 1 and 9 are occupied by "X" and positions 3 and 6 by "O" (see figure 21.1).

Here is how that grid would be rendered by the program:

```
$ ./tictactoe.py -b X.O..O..X
-------------
| X | 2 | O |
-------------
| 4 | 5 | O |
-------------
| 7 | 8 | X |
-------------
No winner.
```

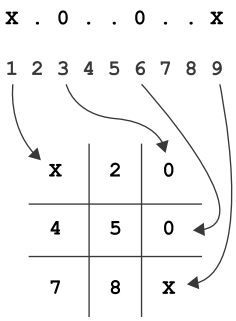

Figure 21.1 The board is nine characters describing the nine cells of the board.

We can additionally modify the given `--board` by passing a `-c` or `--cell` option of 1–9 *and* a `-p` or `--player` option of "X" or "O." For instance, we can mark the first cell as "X" like so:

```
$ ./tictactoe.py --cell 1 --player X
-------------
| X | 2 | 3 |
-------------
```

```
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
No winner.
```

The winner, if any, should be declared with gusto:

```
$ ./tictactoe.py -b XXXOO....
-------------
| X | X | X |
-------------
| O | O | 6 |
-------------
| 7 | 8 | 9 |
-------------
X has won!
```

As usual, we'll use a test suite to ensure that our program works properly. Figure 21.2 shows the string diagram.

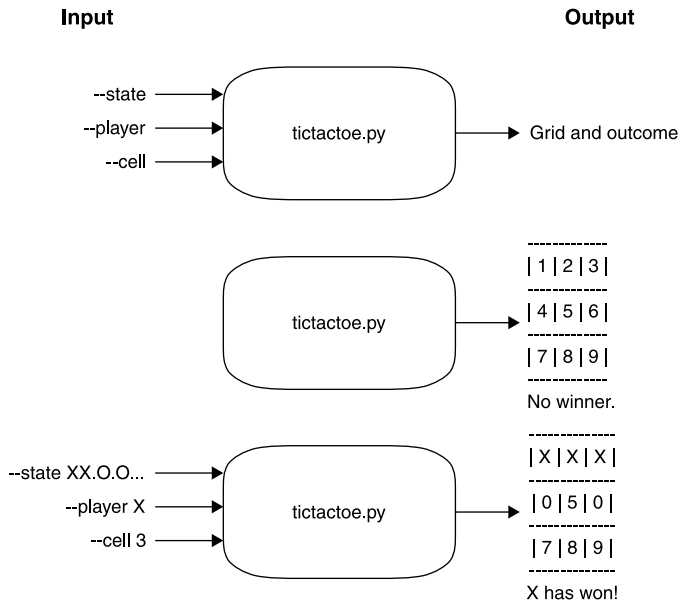

Figure 21.2 Our Tic-Tac-Toe program will play one turn of the game using a board, player, and cell. It should print the board and winner.

21.1.1 Validating user input

There's a fair bit of input validation that needs to happen. The `--board` needs to ensure that any argument is exactly 9 characters and is composed only of `X`, `O`, and `.`:

```
$ ./tictactoe.py --board XXXOOO..
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: --board "XXXOOO.." must be 9 characters of ., X, O
```

Likewise, the `--player` can only be `X` or `O`:

```
$ ./tictactoe.py --player A --cell 1
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: argument -p/--player: \
invalid choice: 'A' (choose from 'X', 'O')
```

And the `--cell` can only be an integer value from 1 to 9:

```
$ ./tictactoe.py --player X --cell 10
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: argument -c/--cell: \
invalid choice: 10 (choose from 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Both `--player` and `--cell` must be present together, or neither can be present:

```
$ ./tictactoe.py --player X
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: Must provide both --player and --cell
```

Lastly, if the `--cell` specified is already occupied by an `X` or an `O`, the program should error out:

```
$ ./tictactoe.py --player X --cell 1 --board X..O.....
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: --cell "1" already taken
```

I would recommend you put all this error checking into `get_args()` so that you can use `parser.error()` to throw the errors and halt the program.

21.1.2 Altering the board

The initial board, once validated, describes which cells are occupied by which player. This board can be altered by adding the `--player` and `--cell` arguments. It may seem silly to not just pass in the already altered `--board`, but this is necessary practice for writing the interactive version.

If you represent `board` as a `str` value, like `'XX.O.O..X'`, and you need to change cell 3 to an `X`, for instance, how will you do that? For one thing, cell 3 is not found at *index* `3` in the given `board`—the index is *one less* than the cell number. The other issue is that a `str` is immutable. Just as in chapter 10's Telephone program, you'll need to figure out a way to modify one character in the board value.

21.1.3 Printing the board

Once you have a board, you'll need to format it with ASCII characters to create a grid. I recommend you make a function called `format_board()` that takes the `board` string as an argument and returns a `str` that uses dashes (`-`) and vertical pipes (`|`) to create a table. I have provided a unit.py file that contains the following test for the default, unoccupied grid:

```
def test_board_no_board():
    """makes default board"""

    board = """
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
""".strip()

    assert format_board('.' * 9) == board
```

Use triple quotes because the string has embedded newlines. The final str.strip() call will remove the trailing newline used to format the code.

If you multiply a string by an integer value, Python will repeat the given string that number of times. Here we create a string of nine dots as the input to format_board(). We expect the return should be an empty board as formatted here.

Now try formatting a board with some other combination. Here's another test I wrote that you might like to use, but feel free to write your own:

```
def test_board_with_board():
    """makes board"""

    board = """
-------------
| 1 | 2 | 3 |
-------------
| O | X | X |
-------------
| 7 | 8 | 9 |
-------------
""".strip()

    assert format_board('...OXX...') == board
```

The given board should have the first and third rows open and the second row with "OXX."

It would be impractical to test every possible combination for the board. When you're writing tests, you'll often have to rely on spot-checking your code. Here I am checking the empty board and a non-empty board. Presumably if the function can handle these two arguments, it can handle any others.

21.1.4 Determining a winner

Once you have validated the input and printed the board, your last task is to declare a winner if there is one. I chose to write a function called `find_winner()` that returns either `X` or `O` if one of those is the winner, or returns `None` if there is no winner. To test

this, I wrote out every possible winning board, to test my function with values for both players. You are welcome to use this test:

```
def test_winning():
    """test winning boards"""

    wins = [('PPP......'), ('...PPP...'), ('......PPP'), ('P..P..P..'),
            ('.P..P..P.'), ('..P..P..P'), ('P...P...P'), ('..P.P.P..')]

    for player in 'XO':
        other_player = 'O' if player == 'X' else 'X'

        for board in wins:
            board = board.replace('P', player)
            dots = [i for i in range(len(board)) if board[i] == '.']
            mut = random.sample(dots, k=2)
            test_board = ''.join([
                other_player if i in mut else board[i]
                for i in range(len(board))
            ])
            assert find_winner(test_board) == player
```

This is a list of the board indexes that, if occupied by the same player, would win.

Check for both players, X and O.

Determine which is the opposite player from X or O.

Iterate through each of the winning combinations.

Change all the P (for "player") values in the given board to the player that we're checking.

Find the indexes of the open cells (indicated by a dot).

Randomly sample two open cells. We will mutate these, so I call them mut.

Alter the board to change the two selected mut cells to other_player.

Assert that find_winner() will determine that this board wins for the given player.

I also wanted to be sure I would not falsely claim that a losing board is winning, so I also wrote the following test to ensure that `None` is returned when there is no winner:

```
def test_losing():
    """test losing boards"""

    losing_board = list('XXOO.....')

    for _ in range(10):
        random.shuffle(losing_board)
        assert find_winner(''.join(losing_board)) is None
```

No matter how this board is arranged, it cannot win, as there are only two marks for each player.

Run 10 tests.

Shuffle the losing board into another configuration.

Assert that, no matter how the board is arranged, we will still find no winner.

If you choose the same function names as I did, you can run `pytest -xv unit.py` to run the unit tests I wrote. If you wish to write different functions, you can create your own unit tests either inside your tictactoe.py file or in another unit file.

After printing the board, be sure to print "{Winner} has won!" or "No winner" depending on the outcome. All righty, you have your orders, so get marching!

21.2 Solution

We're taking baby steps towards the full, interactive game in the next chapter. Right now we need to cement some basics on how just one turn will be played. It's good to make iterations of difficult programs, where you start as simply as possible and slowly add features to build a more complex idea.

```
#!/usr/bin/env python3
"""Tic-Tac-Toe"""

import argparse
import re


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Tic-Tac-Toe',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-b',
                        '--board',
                        help='The state of the board',
                        metavar='board',
                        type=str,
                        default='.' * 9)

    parser.add_argument('-p',
                        '--player',
                        help='Player',
                        choices='XO',
                        metavar='player',
                        type=str,
                        default=None)

    parser.add_argument('-c',
                        '--cell',
                        help='Cell 1-9',
                        metavar='cell',
                        type=int,
                        choices=range(1, 10),
                        default=None)

    args = parser.parse_args()

    if any([args.player, args.cell]) and not all([args.player, args.cell]):
        parser.error('Must provide both --player and --cell')

    if not re.search('^[.XO]{9}$', args.board):
        parser.error (f'--board "{args.board}" must be 9 characters of ., X, O')

    if args.player and args.cell and args.board[args.cell - 1] in 'XO':
        parser.error(f'--cell "{args.cell}" already taken')

    return args
```

The --board will default to nine dots. If you use the multiplication operator (*) with a string value and an integer (in any order), the result is the string value repeated that many times. So "'.' * 9" will produce '.........'.

The --player must be either X or O, which can be validated using choices.

The --cell must be an integer from 1 to 9, which can be validated with type=int and choices=range(1, 10), remembering that the upper bound (10) is not included.

The combination of any() and all() is a way to test that both arguments are present or neither is.

Use a regular expression to check that --board is comprised of exactly nine valid characters.

If both --player and --cell are present and valid, verify that the cell in the board is not currently occupied.

Modify the board if both cell and player are "truthy." Since the arguments are validated in get_args(), it's safe to use them here. That is, I won't accidentally assign an index value that is out of range because I have taken the time to check that the cell value is acceptable.

Since the cells start numbering at 1, subtract 1 from the cell to change the correct index in board.

Since we may need to alter the board, it's easiest to convert it to a list.

Look for a winner in the board.

Print the board.

Print the outcome of the game. The find_winner() function returns either X or O if one of the players has won, or None to no indicate no winner.

Define a function to format the board. The function does not print() the board because that would make it hard to test. The function returns a new string value that can be printed or tested.

Iterate through the cells in the board and decide whether to print the player, if the cell is occupied, or the cell number, if it is not.

The return from the function is a new string created by joining all the lines of the grid on newlines.

Define a function that returns a winner or the value None if there is no winner. Again, the function does not print() the winner but only returns an answer that can be printed or tested.

Iterate through both players, X and O.

There are eight winning boards, which are defined as eight lists of the cells that need to be occupied by the same player. Note that I chose here to represent the actual zero-offset index values and not the 1-based values I expect from the user.

Iterate through each winning combination of cells, unpacking them into the variables i, j, and k.

```
# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    board = list(args.board)

    if args.player and args.cell:
        board[args.cell - 1] = args.player

    print(format_board(board))
    winner = find_winner(board)
    print(f'{winner} has won!' if winner else 'No winner.')


# --------------------------------------------------
def format_board(board):
    """Format the board"""

    cells = [str(i) if c == '.' else c for i, c in enumerate(board, 1)]
    bar = '-------------'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
    ])


# --------------------------------------------------
def find_winner(board):
    """Return the winner"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7],
               [2, 5, 8], [0, 4, 8], [2, 4, 6]]

    for player in ['X', 'O']:
        for i, j, k in winning:
```

```
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
                return player


# ------------------------------------------------
if __name__ == '__main__':
    main()
```

Create a combo that is the value of the board for each of i, j, and k.

Check if the combo is the same player in every position.

If that is True, return the player. If this is never True for any of the combinations, we exit the function without returning a value, and so None is returned by default.

21.2.1 Validating the arguments and mutating the board

Most of the validation can be handled by using `argparse` effectively. Both the `--player` and `--cell` options can be handled by the `choices` option. It's worth taking time to appreciate the use of `any()` and `all()` in this code:

```
if any([args.player, args.cell]) and not all([args.player, args.cell]):
    parser.error('Must provide both --player and --cell')
```

We can play with these functions in the REPL. The `any()` function is the same as using `or` in between Boolean values:

```
>>> True or False or True
True
```

If *any* of the items in a given `list` is "truthy," the whole expression will evaluate to `True`:

```
>>> any([True, False, True])
True
```

If `cell` is a non-zero value, and `player` is not the empty string, they are both "truthy":

```
>>> cell = 1
>>> player = 'X'
>>> any([cell, player])
True
```

The `all()` function is the same as using `and` in between all the elements in a `list`, so *all* of the elements need to be "truthy" in order for the whole expression to be `True`:

```
>>> cell and player
'X'
```

Why does that return `X`? It returns the last "truthy" value, which is the `player` value, so if we reverse the arguments, we'll get the `cell` value:

```
>>> player and cell
1
```

If we use `all()`, it evaluates the truthiness of anding the values, which will be `True`:

```
>>> all([cell, player])
True
```

We are trying to figure out if the user has provided only *one* of the arguments for `--player` and `--cell`, because we need both or we want neither. So we pretend `cell` is `None` (the default) but `player` is `X`. It's true that `any()` of those values is "truthy":

```
>>> cell = None
>>> player = 'X'
>>> any([cell, player])
True
```

But it's not true that they *both* are:

```
>>> all([cell, player])
False
```

So when we `and` those two expressions, they return `False`,

```
>>> any([cell, player]) and all([cell, player])
False
```

because that is the same as saying this:

```
>>> True and False
False
```

The default for `--board` is provided as nine dots, and we can use a regular expression to verify that it's correct:

```
>>> board = '.' * 9
>>> import re
>>> re.search('^[.XO]{9}$', board)
<re.Match object; span=(0, 9), match='.........'>
```

Our regular expression creates a character class composed of the dot (`.`), "X," and "O" by using `[.XO]`. The `{9}` indicates that there must be exactly 9 characters, and the `^` and `$` characters anchor the expression to the beginning and end of the string, respectively (see figure 21.3).

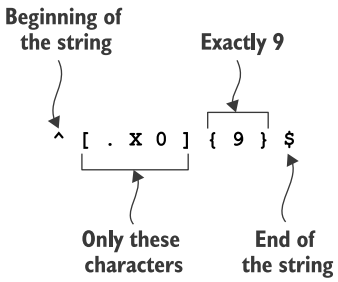

Figure 21.3 We can use a regular expression to exactly describe a valid `--board`.

You could manually validate this using the magic of `all()` again:

- Is the length of `board` exactly 9 characters?
- Is it true that each of the characters is one of those allowed?

Here is one way to write it:

```
>>> board = '...XXXOOO'
>>> len(board) == 9 and all([c in '.XO' for c in board])
True
```

The `all()` part is checking this:

```
>>> [c in '.XO' for c in board]
[True, True, True, True, True, True, True, True, True]
```

Since each character `c` ("cell") in `board` is in the allowed set of characters, all the comparisons are `True`. If we change one of the characters, a `False` will show up:

```
>>> board = '...XXXOOA'
>>> [c in '.XO' for c in board]
[True, True, True, True, True, True, True, True, False]
```

Any `False` value in an `all()` expression will return `False`:

```
>>> all([c in '.XO' for c in board])
False
```

The last piece of validation checks if the `--cell` being set to `--player` is already occupied:

```
if args.player and args.cell and args.board[args.cell - 1] in 'XO':
    parser.error(f'--cell "{args.cell}" already taken')
```

Because `--cell` starts counting from 1 instead of 0, we must subtract 1 when we use it as an index into the `--board` argument. Given the following inputs, the first cell has been set to `X`, and now `O` wants the same cell:

```
>>> board = 'X........'
>>> cell = 1
>>> player = 'O'
```

We can ask if the value in `board` at `cell - 1` has already been set:

```
>>> board[cell - 1] in 'XO'
True
```

Or you could instead check if that position is *not* a dot:

```
>>> boards[cell - 1] != '.'
True
```

It's rather exhausting to validate all the inputs, but this is the only way to ensure that the game is played properly.

In the `main()` function, we might need to alter the `board` of the game if there are arguments for both cell and player. I decided to make `board` into a `list` precisely because I might need to alter it in this way:

```
if player and cell:
    board[cell - 1] = player
```

21.2.2 Formatting the board

Now it's time to create the grid. I chose to create a function that returns a string value that I could test rather than directly printing the grid. Here is my version:

```
def format_board(board):
    """Format the board"""

    cells = [str(i) if c == '.' else c for i, c in enumerate(board, start=1)]
    bar = '-------------'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        bar,
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
    ])
```

I used a list comprehension to iterate through each position and character of board using the enumerate() function. Because I would rather start counting from index position 1 than 0, I used the start=1 option. If the character is a dot, I want to print the position as the cell number; otherwise, I print the character, which will be X or O.

The asterisk, or "splat" (*), is shorthand to expand the list returned by the list slice operation into values that the str.format() function can use.

The "splat" syntax of `*cell[:3]` is a shorter way of writing the code, like so:

```
return '\n'.join([
    bar,
    cells_tmpl.format(cells[0], cells[1], cells[2]), bar,
    cells_tmpl.format(cells[3], cells[4], cells[5]), bar,
    cells_tmpl.format(cells[6], cells[7], cells[8]), bar
])
```

The `enumerate()` function returns a `list` of tuples that include the index and value of each element in a `list` (see figure 21.4). Since it's a lazy function, I must use the `list()` function in the REPL to view the values:

```
>>> board = 'XX.O.O...'
>>> list(enumerate(board))
[(0, 'X'), (1, 'X'), (2, '.'), (3, 'O'), (4, '.'), (5, 'O'), (6, '.'), (7, '.'),
      (8, '.')]
```

In this instance, I would rather start counting at 1, so I can use the `start=1` option:

```
>>> list(enumerate(board, start=1))
[(1, 'X'), (2, 'X'), (3, '.'), (4, 'O'), (5, '.'), (6, 'O'), (7, '.'), (8, '.'),
      (9, '.')]
```

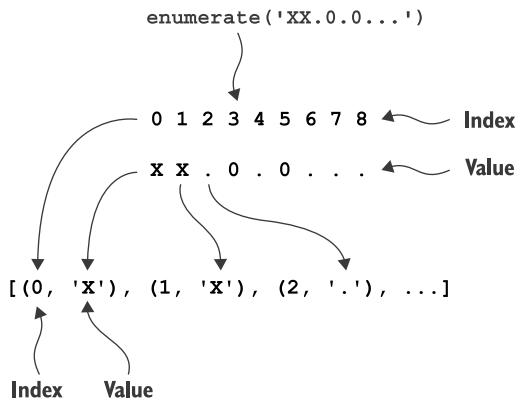

Figure 21.4 The `enumerate()` function will return the index and value of items in a series. By default, the initial index is 0.

This list comprehension could alternatively be written as a `for` loop:

```
cells = []
for i, char in enumerate(board, start=1):
    cells.append(str(i) if char == '.' else char)
```

Initialize an empty list to hold the cells.

Unpack each tuple of the index (starting at 1) and value of each character in board into the variables i (for "integer") and char.

If the char is a dot, use the string version of the i value; otherwise, use the char value.

Figure 21.5 illustrates how `enumerate()` is unpacked into `i` and `char`.

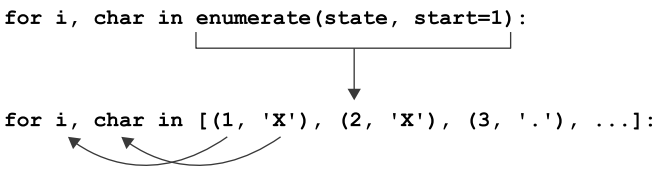

Figure 21.5 The tuples containing the indexes and values returned by `enumerate()` can be assigned to two variables in the `for` loop.

This version of `format_board()` passes all the tests found in unit.py.

21.2.3 Finding the winner

The last major piece to this program is determining if either player has won by placing three of their marks in a row horizontally, vertically, or diagonally.

```
def find_winner(board):
    """Return the winner"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7],
               [2, 5, 8], [0, 4, 8], [2, 4, 6]]
```

There are eight winning positions—the three horizontal rows, the three vertical columns, and the two diagonals—so I decided to create a list where each element is also a list that contains the three cells in a winning configuration.

```
    for player in ['X', 'O']:
        for i, j, k in winning:
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
                return player
```

It's typical to use i as a variable name for "integer" values, especially when their life is rather brief, as here. When more similar names are needed in the same scope, it's also common to use j, k, l, etc. You may prefer to use names like cell1, cell2, and cell3, which are more descriptive but also longer to type. The unpacking of the cell values is exactly the same as the unpacking of the tuples in the previous enumerate() code (see figure 21.6).

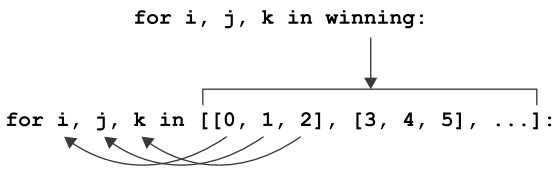

Figure 21.6 As with the unpacking of the `enumerate()` tuples, each list of three elements can be unpacked into three variables in the `for` loop.

The rest of the code checks if either `X` or `O` is the only character at each of the three positions. I worked out half a dozen ways to write this, but I'll just share this one alternate version that uses two of my favorite functions, `all()` and `map()`:

```
for combo in winning:
    group = list(map(lambda i: board[i], combo))
    for player in ['X', 'O']:
        if all(x == player for x in group):
            return player
```

Iterate through each combination of cells in winning.

Use map() to get the value of board at each position in the combination.

Check for each player, X and O.

See if all the values in the group are equal to the given player.

If so, return that player.

If a function has no explicit `return` or never executes a `return`, as would be the case here when there is no winner, Python will use the `None` value as the default `return`. We'll interpret `None` to mean there is no winner when we print the outcome of the game:

```
winner = find_winner(board)
print(f'{winner} has won!' if winner else 'No winner.')
```

That covers this version of the game that plays just one turn of Tic-Tac-Toe. In the next chapter, we'll expand these ideas into an interactive version that starts with a blank board and dynamically requests user input to play the game.

21.3 Going further

- Write a game that will play one hand of a card game like Blackjack (Twenty-one) or War.

Summary

- This program uses a `str` value to represent the Tic-Tac-Toe board with nine characters representing `X`, `O`, or `.` to indicate a taken or empty cell. We sometimes convert that to a `list` to make it easier to modify.
- A regular expression is a handy way to validate the initial board. We can declaratively describe that it should be a string exactly nine characters long composed only of the characters `.`, `X`, and `O`.
- The `any()` function is like chaining `or` between multiple Boolean values. It will return `True` if *any* of the values is "truthy."
- The `all()` function is like using `and` between multiple Boolean values. It will return `True` only if every one of the values is "truthy."
- The `enumerate()` function will return the list index and value for each element in an iterable like a `list`.