22 Tic-Tac-Toe redux: An interactive version with type hints

In this last exercise, we're going to revisit the Tic-Tac-Toe game from the previous chapter. That version played one turn of the game by accepting an initial `--board` and then modifying it if there were also valid options for `--player` and `--cell`. It printed the one board and the winner, if any. We're going to extend those ideas into a version that will always start from an empty board and will play as many turns as needed to complete a game, ending with a winner or a draw.

This program will be different from all the other programs in this book because it will accept no command-line arguments. The game will always start with a blank "board" and with the X player going first. It will use the `input()` function to interactively ask each player, X and then O, for a move. Any invalid move, such as choosing an occupied or non-existing cell, will be rejected. At the end of each turn, the game will decide to stop if it determines there is a win or a draw.

In this chapter you will

- Use and break out of an infinite loop
- Add type hints to your code
- Explore tuples, named tuples, and typed dictionaries
- Use `mypy` to analyze code for errors, especially misuse of types

22.1 Writing itictactoe.py

This is the one program where I won't provide an integration test. The program doesn't take any arguments, and I can't easily write tests that will interact dynamically with the program. This also makes it difficult to show a string diagram, because the output of the program will be different depending on the moves you make. Still, figure 22.1 is an approximation of how you could think of the program starting with no inputs and then looping until some outcome is determined, or the player quits.

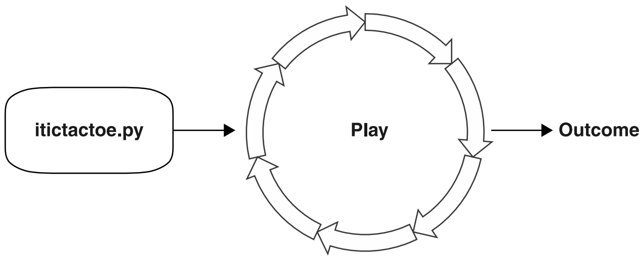


Figure 22.1 This version of Tic-Tac-Toe accepts no arguments and will play in an infinite loop until some conclusion like a win, draw, or forfeit.

I encourage you to start off by running the solution1.py program to play a few rounds of the game. The first thing you may notice is that the program clears the screen of any text and shows you an empty board, along with a prompt for the X player's move. I'll type `1` and press Enter:

```
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Player X, what is your move? [q to quit]: 1
```

Then you will see that cell `1` is now occupied by X, and the player has switched to O:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Player O, what is your move? [q to quit]:
```

If I choose `1` again, I am told that cell is already taken:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
```

```
-------------
| 7 | 8 | 9 |
-------------
Cell "1" already taken
Player O, what is your move? [q to quit]:
```

Note that the player is still O because the previous move was invalid. The same happens if I put in some value that cannot be converted to an integer:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Invalid cell "biscuit", please use 1-9
Player O, what is your move? [q to quit]:
```

Or if I enter an integer that is out of range:

```
-------------
| X | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Invalid cell "10", please use 1-9
Player O, what is your move? [q to quit]:
```

You should be able to reuse many of the ideas from chapter 21's version of the game to validate the user input.

If I play the game to a conclusion where one player gets three in a row, it prints the winning board and proclaims the victor:

```
-------------
| X | O | 3 |
-------------
| 4 | X | 6 |
-------------
| 7 | O | X |
-------------
X has won!
```

22.1.1 Tuple talk

In this version, we'll write an interactive game that always starts with an empty grid and plays as many rounds as necessary to reach a conclusion with a win or a draw. The idea of "state" in the last game was limited to the board—which players were in which cells. This version requires us to track quite a few more variables in our game state:

- The cells of the board, like `..XO..X.O`
- The current player, either `X` or `O`
- Any error, such as the player entering a cell that is occupied or that does not exist or a value that cannot be converted to a number
- Whether the user wishes to quit the game early
- Whether the game is a draw, which happens when all the cells of the grid are occupied but there is no winner
- The winner, if any, so we know when the game is over

You don't need to write your program exactly the way I wrote mine, but you still may find yourself needing to keep track of many items. A `dict` is a natural data structure for that, but I'd like to introduce a new data structure called a "named tuple," as it plays nicely with Python's type hints, which will figure prominently in my solution.

We've encountered tuples throughout the exercises. They've been returned by something like `match.groups()` when a regular expression contains capturing parentheses, like in chapters 14 and 17; when using `zip` to combine two lists, like in chapter 19; or when using `enumerate()` to get a list of index values and elements from a `list`. A `tuple` is an immutable `list`, and we'll explore how that immutability can prevent us from introducing subtle bugs into our programs.

You create a `tuple` whenever you put commas between values:

```
>>> cell, player
(1, 'X')
```

It's most common to put parentheses around them to make it more explicit:

```
>>> (cell, player)
(1, 'X')
```

We could assign this to a variable called `state`:

```
>>> state = (cell, player)
>>> type(state)
<class 'tuple'>
```

We index into a `tuple` using `list` index values:

```
>>> state[0]
1
>>> state[1]
'X'
```

Unlike with a `list`, we cannot change any of the values inside the `tuple`:

```
>>> state[1] = 'O'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

It's going to be inconvenient remembering that the first position is the `cell` and the second position is the `player`, and it will get much worse when we add all the other fields. We could switch to using a `dict` so that we can use strings to access the values of `state`, but dictionaries are mutable, and it's also easy to misspell a key name.

22.1.2 Named tuples

It would be nice to combine the safety of an immutable `tuple` with named fields, which is exactly what we get with the `namedtuple()` function. First, you must import it from the `collections` module:

```
>>> from collections import namedtuple
```

The `namedtuple()` function allows us to describe a new `class` for values. Let's say we want to create a class that describes the idea of `State`. A class is a group of variables, data, and functions that together can be used to represent some idea. The Python language itself, for example, has the `str` class, which represents the idea of a sequence of characters that can be contained in a variable that has some `len` (length), and which can be converted to uppercase with `str.upper()`, can be iterated with a `for` loop, and so forth. All these ideas are grouped into the `str` class, and we've used `help(str)` to read the documentation for that class inside the REPL.

The class name is the first argument we pass to `namedtuple()`, and the second argument is a `list` of the field names in the class. It's common practice to capitalize class names:

```
>>> State = namedtuple('State', ['cell', 'player'])
```

We've just created a new `type` called `State`!

```
>>> type(State)
<class 'type'>
```

Just as there is a function called `list()` to create a `list` type, we can now use the `State()` function to create a named tuple of the type `State` that has two named fields, `cell` and `player`:

```
>>> state = State(1, 'X')
>>> type(state)
<class '__main__.State'>
```

We can still access the fields with index values, like any `list` or `tuple`:

```
>>> state[0]
1
>>> state[1]
'X'
```

But we can also use their names, which is much nicer. Notice that there are no parentheses at the end, as we are accessing a field, not calling a method:

```
>>> state.cell
1
>>> state.player
'X'
```

Because `state` is a `tuple`, we cannot mutate the value once it has been created:

```
>>> state.cell = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

This is actually *good* in many instances. It's often quite dangerous to change your data values once your program has started. You should use tuples or named tuples whenever you want a list- or dictionary-like structure that cannot be accidentally modified.

There is a problem, however, in that there's nothing to prevent us from instantiating a `state` with the fields out of order *and of the wrong types*—`cell` should be an `int`, and `player` should be a `str`!

```
>>> state2 = State('O', 2)
>>> state2
State(cell='O', player=2)
```

In order to avoid that, you can use the field names, so that their order no longer matters:

```
>>> state2 = State(player='O', cell=2)
>>> state2
State(cell=2, player='O')
```

Now you have a data structure that looks like a `dict` but has the immutability of a `tuple`!

22.1.3 Adding type hints

We still have a big problem in that there's nothing preventing us from assigning a `str` to the `cell`, which ought to be an `int`, and vice versa for `int` and `player`:

```
>>> state3 = State(player=3, cell='X')
>>> state3
State(cell='X', player=3)
```

Starting in Python 3.6, the `typing` module allows you to add *type hints* to describe the data types for variables. You should read PEP 484 (www.python.org/dev/peps/pep-0484/) for more information, but the basic idea is that we can use this module to describe the appropriate types for variables and type signatures for functions.

I'm going to improve our `State` class by using the `NamedTuple` class from the `typing` module as the base class. First we need to import from the `typing` module the classes we'll need, such as `NamedTuple`, `List`, and `Optional`, the last of which describes a type that could be `None` or some other class like a `str`:

```
from typing import List, NamedTuple, Optional
```

Now we can specify a `State` class with named fields, types, and even default values to represent the initial state of the game where the board is empty (all dots) and player X goes first. Note that I decided to store the `board` as a `list` of characters rather than a `str`:

```
class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None
```

We can use the `State()` function to create a new value that's set to the initial state:

```
>>> state = State()
>>> state.board
['.', '.', '.', '.', '.', '.', '.', '.', '.']
>>> state.player
'X'
```

You can override any default value by providing the field name and a value. For instance, we could start the game off with player `O` by specifying `player='O'`. Any field we don't specify will use the default:

```
>>> state = State(player='O')
>>> state.board
['.', '.', '.', '.', '.', '.', '.', '.', '.']
>>> state.player
'O'
```

We get an exception if we misspell a field name, like `playre` instead of `player`:

```
>>> state = State(playre='O')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() got an unexpected keyword argument 'playre'
```

22.1.4 Type verification with Mypy

As nice as all the above is, *Python will not generate a runtime error if we assign an incorrect type*. For instance, I can assign `quit` a `str` value of `'True'` instead of the `bool` value `True`, and nothing at all happens:

```
>>> state = State(quit='True')
>>> state.quit
'True'
```

The benefit of type hints comes from using a program like Mypy to check our code. Let's place all this code into a small program called typehints.py in the repo:

```
#!/usr/bin/env python3
""" Demonstrating type hints """

from typing import List, NamedTuple, Optional


class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None


state = State(quit='False')

print(state)
```

quit is defined as a bool, which means it should only allow values of True and False.

We are assigning the str value 'True' instead of the bool value True, which might be an easy mistake to make, especially in a very large program. We'd like to know this type of error will be caught!

The program will execute *with no errors*:

```
$ ./typehints.py
State(board=['.', '.', '.', '.', '.', '.', '.', '.', '.'], player='X', \
quit='False', draw=False, error=None, winner=None)
```

But the Mypy program will report the error of our ways:

```
$ mypy typehints.py
typehints.py:16: error: Argument "quit" to "State" has incompatible type
     "str"; expected "bool"
Found 1 error in 1 file (checked 1 source file)
```

If I correct the program like so,

```
#!/usr/bin/env python3
""" Demonstrating type hints """

from typing import List, NamedTuple, Optional


class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None
```

Again, quit is a bool value.

```
state = State(quit=True)

print(state)
```

We have to assign an actual bool value in order to pass muster with Mypy.

now Mypy will be satisfied:

```
$ mypy typehints2.py
Success: no issues found in 1 source file
```

22.1.5 Updating immutable structures

If one of the advantages of using `NamedTuples` is their *immutability*, how will we keep track of changes to our program? Consider our initial state of an empty grid with the player X going first:

```
>>> state = State()
```

Imagine X takes cell 1, so we need to change `board` to `X........` and the `player` to `O`. We can't directly modify `state`:

```
>>> state.board=list('X........')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

We could use the `State()` function to create a new value to overwrite the existing `state`. That is, since we can't change anything *inside* the `state` variable, we could instead point `state` to an entirely new value. We did this in the second solution in chapter 8, where we needed to change a `str` value, because they are also immutable in Python.

To do this, we can copy all the current values that haven't changed and combine them with the changed values:

```
>>> state = State(board=list('X........'), player='O', quit=state.quit, \
    draw=state.draw, error=state.error, winner=state.winner)
```

The `namedtuple._replace()` method, however, provides a much simpler way to do this. Only the values we provide are changed, and the result is a new `State`:

```
>>> state = state._replace(board=list('X........'), player='O')
```

We overwrite our `state` variable with the return from `state._replace()`, just as we have repeatedly overwritten string variables with new values:

```
>>> state
State(board=['X', '.', '.', '.', '.', '.', '.', '.', '.', '.'], player='O', \
      quit=False, draw=False, error=None, winner=None)
```

This is much more convenient than having to list all the fields—we only need to specify the fields that have changed. We are also prevented from accidentally modifying any of the other fields, and we are likewise prevented from forgetting or misspelling any fields or setting them to the wrong types.

22.1.6 Adding type hints to function definitions

Now let's look at how we can add type hints to our function definitions. For an example, we can modify our `format_board()` function to indicate that it takes a parameter called `board`, which is a list of string values, by adding `board: List[str]`. Additionally, the function returns a `str` value, so we can add `-> str` after the colon on the `def` to indicate this, as in figure 22.2.

Board has the type `List[str]`.

```
def format_board(board: List[str]) -> str:
```

The function returns a `str`.

Figure 22.2 Adding type hints to describe the type of the parameter and the return value

The annotation for `main()` indicates that the `None` value is returned, as shown in figure 22.3.

Function takes no arguments.

```
def main() -> None:
```

The function returns None.

Figure 22.3 The `main()` function accepts no parameters and returns `None`.

What's really terrific is that we can define a function that takes a value of the type `State`, and Mypy will check that this kind of value is actually being passed (see figure 22.4).

Try playing my version of the game and then writing your own that behaves similarly. Then take a look at how I wrote an interactive solution that incorporates these ideas of data immutability and type safety.

State is of the type `State`**.**

```
def get_move(state: State) -> State:
```

The function returns a `State`**.**

Figure 22.4 We can use custom types in type hints. This function takes and returns a value of the type `State`**.**

22.2 Solution

This is the last program! I hope that writing the simpler version in the previous chapter gave you ideas for making this work. Did the type hints and unit tests also help?

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

from typing import List, NamedTuple, Optional
```

Import the classes we'll need from the typing module.

```
class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None
```

Declare a class that is based on the NamedTuple class. Define field names, types, and defaults for the values this class can hold.

Start an infinite loop. When we have a reason to stop, we can break out of the loop.

Print a special sequence that most terminals will interpret as a command to clear the screen.

```
# --------------------------------------------------
def main() -> None:
    """Make a jazz noise here"""

    state = State()

    while True:
        print("\033[H\033[J")
        print(format_board(state.board))

        if state.error:
            print(state.error)
        elif state.winner:
            print(f'{state.winner} has won!')
            break

        state = get_move(state)
```

Instantiate the initial state as an empty grid and the first player as X.

Print the current state of the board.

Print any errors, such as the user not choosing a valid cell.

If there is a winner, proclaim the victor and break out of the loop.

Get the next move from the player. The get_move() function accepts a State type and returns one too. We overwrite the existing state variable each time through the loop.

```
        if state.quit:
            print('You lose, loser!')
            break
        elif state.draw:
            print("All right, we'll call it a draw.")
            break
```

If the game has reached a stalemate where all cells are occupied but there is no winner, declare a draw and break from the loop.

If the user has decided to withdraw from the game prematurely, insult them, and break from the loop.

Check if the user entered a value that can be converted to a digit using str.isdigit() and if the integer version of the value is in the valid range.

If so, replace the quit value of the state with True and return with the new state. Note that no other values in the state are modified.

Copy the player from the state, since we'll refer to it several times in the function body.

Define a get_move() function that takes and returns a State type.

Use the input() function to ask the player for their next move. Tell them how to quit the game early so they don't have to use Ctrl-C to interrupt the program.

```
# --------------------------------------------------
def get_move(state: State) -> State:
    """Get the player's move"""

    player = state.player
    cell = input(f'Player {player}, what is your move? [q to quit]: ')

    if cell == 'q':
        return state._replace(quit=True)

    if not (cell.isdigit() and int(cell) in range(1, 10)):
        return state._replace(error=f'Invalid cell "{cell}", please use 1-9')

    cell_num = int(cell)
    if state.board[cell_num - 1] in 'XO':
        return state._replace(error=f'Cell "{cell}" already taken')

    board = state.board
    board[cell_num - 1] = player
    return state._replace(board=board,
                          player='O' if player == 'X' else 'X',
                          winner=find_winner(board),
                          draw='.' not in board,
                          error=None)
```

First check if the user wants to quit.

See if the board is open at the indicated cell.

Return a new state value with the new board value, the current player switched to the other player, and if there is a winner or a draw.

After we have verified that cell is a valid integer value, convert it to an integer.

Use the cell value to update the board with the current player.

Copy the current board because we need to modify it and state.board is immutable.

If not, return an updated state that has an error. Note that the current state and player remain unchanged so that the same player has a retry with the same board until they provide valid input.

If not, return an updated state with an error. Again, nothing else about the state is changed, so we retry the round with the same player and state.

```
# --------------------------------------------------
def format_board(board: List[str]) -> str:
    """Format the board"""

    cells = [str(i) if c == '.' else c for i, c in enumerate(board, 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
    ])
```

The only change from the previous version of this function is the addition of type hints. The function accepts a list of string values (the current board) and returns a formatted grid of the board state.

```
# --------------------------------------------------
def find_winner(board: List[str]) -> Optional[str]:
    """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

    return None
```

This is also the same function as before, but with type hints. The function accepts the board as a list of strings and returns an optional string value, which means it could also return None.

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

22.2.1 A version using TypedDict

New to Python 3.8 is the `TypedDict` class, which looks very similar to a `NamedTuple`. Let's look at how using this as the base class changes parts of our program. One crucial difference is that you cannot (yet) set default values for the fields:

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

from typing import List, Optional, TypedDict
```

Import TypedDict instead of NamedTuple.

```
class State(TypedDict):
    board: str
    player: str
    quit: bool
    draw: bool
    error: Optional[str]
    winner: Optional[str]
```

Base State on a TypedDict.

We have to set our initial values when we instantiate a new `state`:

```
def main() -> None:
    """Make a jazz noise here"""

    state = State(board='.' * 9,
                  player='X',
                  quit=False,
                  draw=False,
                  error=None,
                  winner=None)
```

Syntactically, I prefer using `state.board` with the named tuple rather than the dictionary access of `state['board']`:

```
while True:
    print("\033[H\033[J")
    print(format_board(state['board']))

    if state['error']:
        print(state['error'])
    elif state['winner']:
        print(f"{state['winner']} has won!")
        break

    state = get_move(state)

    if state['quit']:
        print('You lose, loser!')
        break
    elif state['draw']:
        print('No winner.')
        break
```

Beyond the convenience of accessing the fields, I prefer the read-only nature of the `NamedTuple` to the mutable `TypedDict`. Note how in the `get_move()` function, we can change the `state`:

```
def get_move(state: State) -> State:
    """Get the player's move"""

    player = state['player']
    cell = input(f'Player {player}, what is your move? [q to quit]: ')

    if cell == 'q':
        state['quit'] = True
        return state

    if not (cell.isdigit() and int(cell) in range(1, 10)):
        state['error'] = f'Invalid cell "{cell}", please use 1-9'
        return state

    cell_num = int(cell)
    if state['board'][cell_num - 1] in 'XO':
```

Here we are directly modifying the TypedDict, whereas the NamedTuple version used state._replace() to return an entirely new state value.

Another place where the state is directly modifiable. You may prefer this approach.

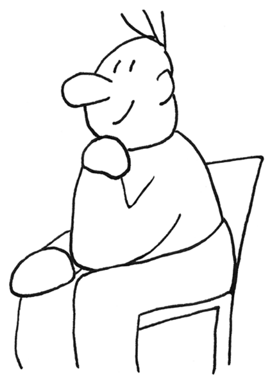
```
        state['error'] = f'Cell "{cell}" already taken'
        return state

    board = list(state['board'])
    board[cell_num - 1] = player

    return State(
        board=''.join(board),
        player='O' if player == 'X' else 'X',
        winner=find_winner(board),
        draw='.' not in board,
        error=None,
        quit=False,
    )
```

In my opinion, a `NamedTuple` has nicer syntax, default values, and immutability over the `TypedDict` version, so I prefer it. Regardless of which you choose, the greater lesson I hope to impart is that we should try to be explicit about the "state" of the program and when and how it changes.

22.2.2 Thinking about state



The idea of program state is that a program can remember changes to variables over time. In the previous chapter, our program accepted a given `--board` and possible values for `--cell` and `--player` that might alter the board. Then the game printed a representation of the board. In this chapter's interactive version, the board always begins as an empty grid and changes with each turn, which we modeled as an infinite loop.

It is common in programs like this to see programmers use *global variables* that are declared at the top of the program outside of any function definitions so that they are *globally* visible throughout the program. While common, it's not considered a best practice, and I would discourage you from ever using globals unless you can see no other way. I would suggest, instead, that you stick to using small functions that accept all the values required and return a single type of value. I would also suggest you use data structures like typed, named tuples to represent program state, and that you guard the changes to state very carefully.

22.3 Going further

- Incorporate spicier insults. Maybe bring in the Shakespearean generator?
- Write a version that allows the user to start a new game without quitting and restarting the program.
- Write other games like Hangman.

Summary

- Type hints allow you to annotate variables as well as function parameters and return values with the types of the values.
- Python itself will ignore type hints at runtime, but Mypy can use type hints to find errors in your code before you ever run it.
- A `NamedTuple` behaves a bit like a dictionary and a bit like an object but retains the immutability of tuples.
- Both `NamedTuple` and `TypedDict` allow you to create a novel type with defined fields and types that you can use as type hints to your own functions.
- Our program used a `NamedTuple` to create a complex data structure to represent the state of our program. The state included many variables, such as the current board, the current player, any errors, the winner, and so on, each of which was described using type hints.
- While it is difficult to write integration tests for an interactive program, we can still break a program into small functions (such as `format_board()` or `get_winner()`) for which we write and run unit tests.