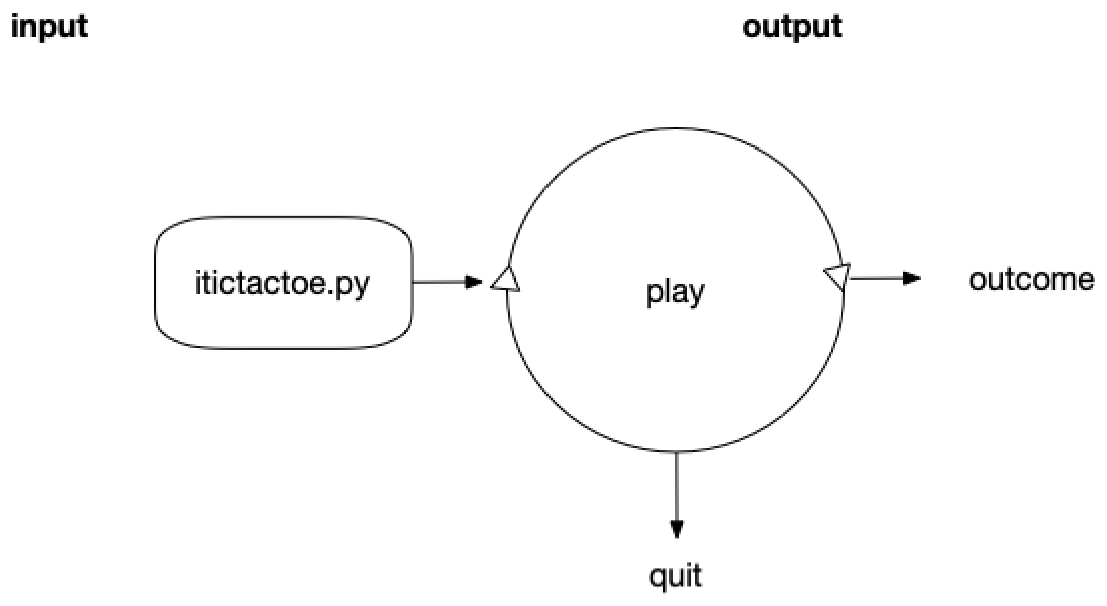
1. Tic-Tac-Toe Redux: An interactive version with type hints

In this last exercise, we're going to revisit the Tic-Tac-Toe game from the previous chapter. That version played one round of the game by accepting some 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 rounds as needed for the game to end with a winner or a draw. This program will be different from all the other programs in this book in that 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 iteractively 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 round, the game will decide to stop if it determines there is a win or a draw.

1.1. Writing `itictactoe.py`

This is the one program where I don't provide an integration test since the program doesn't take any arguments. 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, here is an approximation of how we could think of the program starting with no inputs and then looping until some outcome is determined or the player quits:



I would encourage you to start off by running the `itictactoe.py` program to play a few rounds of the game. The first thing you may notice is that it should clear the screen of any text and show you an empty board along with a prompt for the `X` player's move. I'll type `1<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 the 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. It's likewise 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 a valid 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 your previous version to validate the user input.

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

```
-------------
| X | O | 3 |
-------------
| 4 | X | 6 |
-------------
| 7 | O | 9 |
-------------
Player X, what is your move? [q to quit]: 9
X has won!
```

1.1.1. Tuple talk

In this version, we'll write a 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:

1. The cells of the **board**, like `..XO..X.O`

2. The current **player**, either `X` or `O`
3. Any **error** such as the player entering a cell that is occupied or does not exist or a value that cannot be converted to a number.
4. Whether to the user wishes to **quit** the game early.
5. Whether the game is a **draw**, which is when all the cells of the grid are occupied but there is no winner and so we must quit.
6. The **winner**, if any, so I know when to quit.

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 when they've been returned by something like `match.groups` when a regular expression contains capturing parentheses like in "Rhymer" or "Mad Libs" or using `zip` to combine two lists like in "WOD." 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.

1.1.2. 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 that represents the idea of a sequence of characters that can be contained in a variable that has some `len` (length), 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.

It's common practice to capitalize class names. 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:

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

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

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

Just as there is a `list` type that is also a function called `list()`, we can use the function `State` 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 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 such that their order no longer matters:

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

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

1.1.3. 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 (https://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.

The `typing` module defines a `NamedTuple` class that we can use as the base for a `class` of our own. We can create our own classes to represent ideas such as our game `State`. 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 an `str`:

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

And 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 the 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 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 can start the game off with the 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'
```

1.1.4. Type verification with `mypy`

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

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

The benefit to use 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      ①
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None


state = State(quit='False') ②

print(state)
```

① `quit` is defined as a `bool`, which means it should only allow values of `True` and `False`.

② 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:

```python
#!/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   ①
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None


state = State(quit=True) ②

print(state)
```

① Again, the `quit` is a `bool` value.

② 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
```

1.1.5. Updating immutable structures

If one of the advantages to using `NamedTuples` is their *immutability*, then 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 the `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
```

I could use the `State` class to overwrite the existing `state`. That is, since I can’t change anything *inside* the `state` variable, I can instead point the `state` to an entirely new value. We did this in exercises [1] where we needed to change a `str` value because they are also *immutable* in Python.

To do this, I can copy all the current values that haven’t changed for `quit` and such:

```
>>> 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 provided are changed, and the result is a new `State`:

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

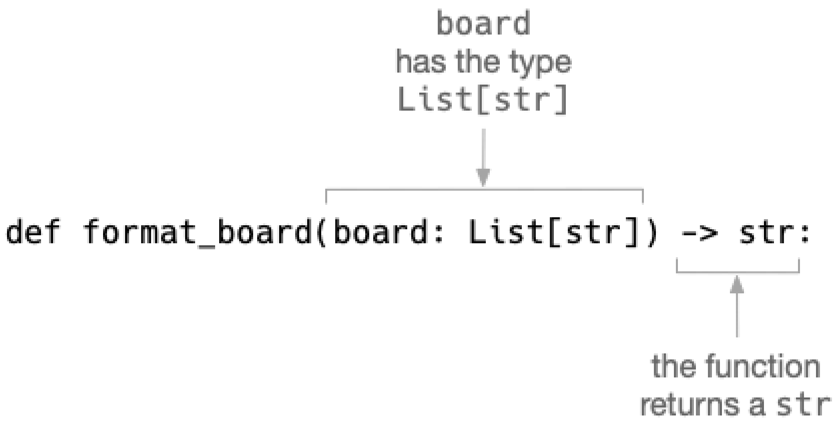
We overwrite our existing `state` with the return from `state._replace` because the original `state` is not changed.

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

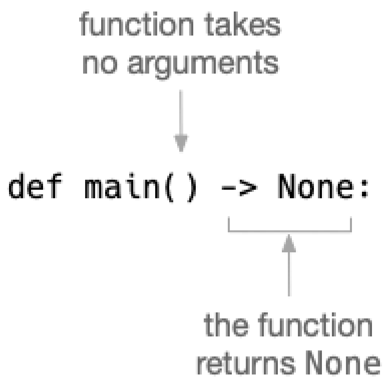
This is much more convenient than having to list all the fields — we only need to specify the fields that changed. We are also prevented from accidentally modifying any of the other fields, and we are likewise prevented from forgetting any fields or setting them to the wrong types!

1.1.6. Adding type hints to function definitions

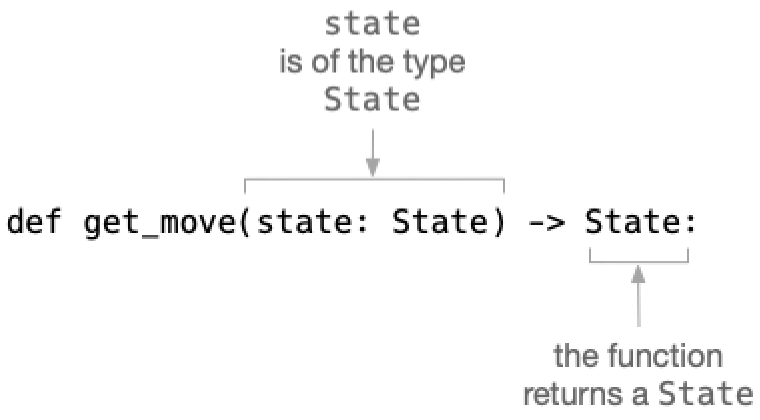
Now let’s look at how we can add type hints to our function definitions! For an example, we can modify our `format_board` function to indicate that it takes the parameter called `board` which is a list of string values by adding `board: List[str]`. Additionally, the function returns a `str` value, so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



The annotation for `main` indicates that the `None` value is returned:



What's really terrific is that we can define a function that takes a value of the type `State`, and `mypy` would check that this kind of value is actually being passed!



Let's look at an interactive solution that incorporates all of these ideas to create a program that has the additional benefits of data immutability and type safety!

1.2. Interactive solution

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

from typing import List, NamedTuple, Optional                ①


class State(NamedTuple):                                     ②
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None


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

    state = State()                                          ③

    while True:                                              ④
        print("\033[H\033[J")                                ⑤
        print(format_board(state.board))                     ⑥

        if state.error:                                      ⑦
            print(state.error)

        state = get_move(state)                              ⑧

        if state.quit:                                       ⑨
            print('You lose, loser!')
            break
        elif state.winner:                                   ⑩
            print(f'{state.winner} has won!')
            break
        elif state.draw:                                     ⑪
            print("All right, we'll call it a draw.")
            break


# --------------------------------------------------
def get_move(state: State) -> State:                         ⑫
    """Get the player's move"""

    player = state.player                                    ⑬
    cell = input(f'Player {player}, what is your move? [q to quit]: ') ⑭

    if cell == 'q':                                        ⑮
        return state._replace(quit=True)                   ⑯

    if not (cell.isdigit() and int(cell) in range(1, 10)): ⑰
        return state._replace(error=f'Invalid cell "{cell}", please use 1-9') ⑱

    cell_num = int(cell)                                   ⑲
    if state.board[cell_num - 1] in 'XO':                  ⑳
        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)


# --------------------------------------------------
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
    ])


# --------------------------------------------------
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


# --------------------------------------------------
```

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

① Import the classes we’ll need from the `typing` module.

② Declare a `class` that is based on the `NamedTuple` class. Define field names, types, and defaults for the values this class can hold.

③ Instantiate our initial `state` as an empty grid and the first player as `X`.

④ Start an infinite loop using `while True`. When we have a reason to stop, we can `break` out of the loop.

⑤ Print a special sequence that most terminals will interpret as a command to clear the screen.

⑥ Print the current state of the board.

⑦ If there is an error (such as the user didn’t choose a valid cell), print that.

⑧ Get the next move from the player. The `get_move` accepts a `State` type and returns one, too. We overwrite our existing `state` variable each time through the loop.

⑨ If the user has decided to withdraw from the game prematurely, insult them and `break` from the loop.

⑩ If there is a winner, proclaim the victor and `break` from the loop.

⑪ If we have reached a stalemate where all cells are occupied but there is no winner, declare a draw and `break` from the loop.

⑫ Define a `get_move` function that takes and returns a `State` type.

⑬ Copy the `player` from the `state` since I’ll refer to it several times in the function body.

⑭ 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` (control key plus "c") to interrupt the program.

⑮ First check if they want to quit.

⑯ If so, replace the `quit` value of our `state` with `True` and return with the new state. Note that no other values in the `state` are modified.

⑰ Check if the user gave us a value that can be converted to a digit using `str.isdigit` and if the `int` version of the value is in the valid range.

⑱ 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.

⑲ After we have verified that `cell` is a valid integer value, convert it to an `int`.

⑳ See if the `board` is open at the indicated `cell`.

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`.

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

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

Return a new `state` value where we update the `board`, switch the `player` to the other, and check if there is a winner or a draw.

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.

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 `str` value which means it could also return `None`.

1.3. A version using `TypedDict`

New to Python 3.8 is the `TypedDict` class that looks very similar to a `NamedTuple`. One crucial difference is that you cannot (yet) set default values for the fields:

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

from typing import List, Optional, TypedDict ①


class State(TypedDict):                     ②
    board: str
    player: str
    quit: bool
    draw: bool
    error: Optional[str]
    winner: Optional[str]
```

① Import `TypedDict` instead of `NamedTuple`.

② Base our `State` on a `TypedDict`.

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

```python
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']`:

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

    if state['error']:
        print(state['error'])

    state = get_move(state)

    if state['quit']:
        print('You lose, loser!')
        break
    elif state['winner']:
        print(f"{state['winner']} has won!")
        break
    elif state['draw']:
        print('No winner.')
        break
```

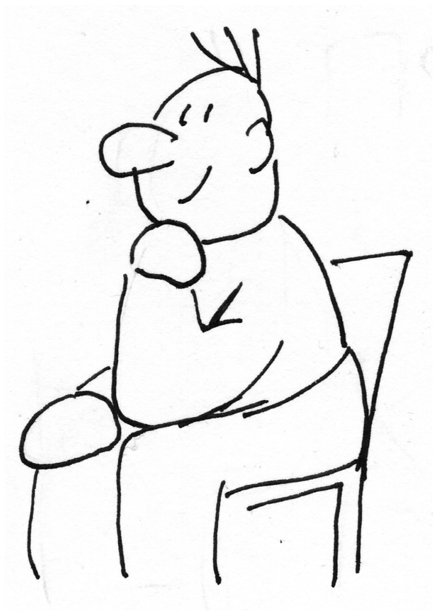
Beyond the convenience of accessing the fields, I prefer the read-only nature of the `NamedTuple` to the mutable `TypedDict`. Note how in the `get_move` function, I can change the `state`:

```python
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':
        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 might choose, the greater lesson I hope to impart is that we try to be explicit about the "state" of the program and when and how it changes.

1.3.1. Thinking about state

The idea of program state describes how a program can remember changes to variables over time. Our first version accepted a given `--state` and possibly values for `--cell` and `--player` that might alter the state. Then the board prints a representation of the state. In the interactive version, the state always begins as an empty grid, and the state changes with each round 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 a 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 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!

1.4. Review

- The first program used a `str` value to represent the "state" of the Tic-Tac-Toe board with nine characters representing `X`, `O`, or `.` to indicate a taken or empty cell, respectively.
- The second program used a `NamedTuple` to create a data structure for a much expanded idea of "state" that included many variables such as the current board, the current player, any errors, etc. The `NamedTuple` behaves a bit like a `dict`, a bit like an object, but retains the immutability of tuples.
- Type hints allow you to annotate variables and function parameters and return values.
- Python itself will ignore type hints at run-time, but `mypy` can use type hints to find logical errors in your code before you ever run it.
- 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.

1.5. Going further

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

[1] See "Apples and Bananas" solution 2.