IMMUTABLE PROGRAMMING

WRITING FUNCTIONAL PYTHON

Cale Pennington

@vengefulpickle

github.com/cpennington

COMPARE

Python

Mutable by default (mostly)

Haskell

Immutable by default

IMMUTABILITY ALLOWS LOCAL THINKING

IMMUTABILITY IN PYTHON

```python
# Attribute assignment
x.foo = 1

# Item assignment
x["foo"] = 3

# Methods that modify state
x.add("foo")

# Modifying an objects own attributes
self.foo = 4
```

IMMUTABILITY IN PYTHON

```
# Name assignment
x = 1

# Reading attributes
x = self.foo

# Read-only methods
x = y.items()
```

IMMUTABILITY IN PYTHON

```python
object()

{"foo": 1}

{"foo"}

["foo", "bar"]

(i for i in range(3))

(object(), object())
```

IMMUTABILITY IN PYTHON

```
1

"bar"

(1, 2, 3)

((1, 2), (2, 3))

frozenset(1, 2, 3)
```

TOOLS FOR LOCAL THINKING

- @property
- tuple (and namedtuple)
- Commands

THE SETUP

GAME LOOP

```
def main():
    board = Board()
    while not board.is_finished:
        print(board)

        move = input(f"Player {board.player.value} move (x y)? ")
        x, y = move.split()

        board.do_move(int(x), int(y))

class Board():
    def __init__(self):
        self.board = [[Player.NA]*3]*3]

    def do_move(self, x, y):
        if self.board[x][y] == Player.NA:
            self.board[x][y] = self.player
```

PROPERTY

```
@property
def player(self):
    plays = Counter(sum(self.board, []))
    if plays[Player.O] < plays[Player.X]:
        return Player.O
    else:
        return Player.X
```

TESTS

```
def test_game_end(self):
    self.assertFalse(self.board.is_finished)
    self.board.do_move(0, 0)
    self.assertFalse(self.board.is_finished)
```

TESTS

```
def test_game_end(self):
    self.assertFalse(self.board.is_finished)
    self.board.do_move(0, 0)
    self.assertFalse(self.board.is_finished)

======================================================================
FAIL: test_game_end (tictactoe_v4_properties.TestTicTacToe)
----------------------------------------------------------------------
Traceback (most recent call last):
File "../tictactoe_v4_properties.py", line 93, in test_game_end
    self.assertFalse(self.game.is_finished)
AssertionError: True is not false
```

TESTS

```python
def test_moves_made(self):
    # Store the state of the board before a move
    before = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Make a single move
    self.board.do_move(0, 0)

    # Store the state of the board after the move
    after = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Compare the state before and after
    self.assertEqual(after - before, {(0, 0, Player.X)})
    self.assertEqual(before - after, {(0, 0, Player.NA)})
```

TESTS

```
def test_moves_made(self):
    # Store the state of the board before a move
    before = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Make a single move
    self.board.do_move(0, 0)

    # Store the state of the board after the move
    after = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Compare the state before and after
    self.assertEqual(after - before, {(0, 0, Player.X)})
    self.assertEqual(before - after, {(0, 0, Player.NA)})
```

TESTS

```python
def test_moves_made(self):
    # Store the state of the board before a move
    before = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Make a single move
    self.board.do_move(0, 0)

    # Store the state of the board after the move
    after = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Compare the state before and after
    self.assertEqual(after - before, {(0, 0, Player.X)})
    self.assertEqual(before - after, {(0, 0, Player.NA)})
```

TESTS

```
======================================================================
FAIL: test_moves_made (tictactoe_v4_properties.TestTicTacToe)
----------------------------------------------------------------------
Traceback (most recent call last):
File "../tictactoe_v4_properties.py", line 116,
in test_moves_made
    self.assertEqual(after - before, {(0, 0, Player.X)})
AssertionError: Items in the first set but not the second:
(1, 0, <Player.X: 'X'>)
(2, 0, <Player.X: 'X'>)
```

TESTS

```
class Board():
    def __init__(self):
        self.board = [[Player.NA]*3]*3]

    def __init__(self):
        self.board = [[Player.NA]*3 for _ in range(3)]
```

Spooky action at a distance

TESTS

```python
class Board():
    def __init__(self):
        self.board = [[Player.NA]*3]*3]

    def __init__(self):
        self.board = [[Player.NA]*3 for _ in range(3)]
```

Saaad... ☹

IMMUTABLE

STORAGE

```
class Board():
    def __init__(self):
        self.board = ((Player.NA, )*3, )*3
```

TESTS

```python
def test_moves_made(self):
    # Store the state of the board before a move
    before = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Make a single move
    self.board.do_move(0, 0)

    # Store the state of the board after the move
    after = {(x, y, self.board.board[x][y]) for (x, y) in ALL_MOVES}

    # Compare the state before and after
    self.assertEqual(after - before, {(0, 0, Player.X)})
    self.assertEqual(before - after, {(0, 0, Player.NA)})
```

TESTS

```
def test_moves_made(self):
    # Store the state of the board before a move
    before = BoardState()




    # Store the state of the board after the move
    after = before.do_move(0, 0)

    # Compare the state before and after
    self.assertEqual(after - before, {(0, 0, Player.X)})
    self.assertEqual(before - after, {(0, 0, Player.NA)})
```

STORAGE

```
class Board():
    def __init__(self):
        self.board = ((Player.NA, )*3, )*3
```

STORAGE

```
class BoardState(namedtuple('_BoardState', ['board'])):
    ...

BoardState.__new__.__defaults__ = (((Player.NA, )*3, )*3, )
```

NAMEDTUPLE

```python
from collections import namedtuple

Widgit = namedtuple('Widgit', ['height', 'weight'])
x = Widgit(10, 20)
x.height  # 10
x.weight  # 20
list(x)   # [10, 20]
```

STORAGE

```
class BoardState(namedtuple('_BoardState', ['board'])):
    ...

BoardState.__new__.__defaults__ = (((Player.NA, )*3, )*3, )
```

ACTION

```
def do_move(self, x, y):
    if self.board[x][y] == Player.NA:
        self.board[x][y] = self.player
```

ACTION

```
def do_move(self, x, y):
    if self.board[x][y] == Player.NA:
        new_row = replace(self.board[x], y, self.player)
        new_board = replace(self.board, x, new_row)
        return BoardState(new_board)
    else:
        return self

def replace(tpl, idx, value):
    return tpl[:idx] + (value, ) + tpl[idx+1:]
```

COMMANDS

PLAYER

```
def main():
    states = [BoardState()]
    while not states[-1].is_finished:
        print(states[-1])
        move = input(f"Player {states[-1].player.value}: "
                     "x y to move, u to undo, "
                     "gN to revert to move N)? ")
        if move == 'u':
            states.pop()
        elif move.startswith('g'):
            states = states[:int(move.replace('g','')) + 1]
        else:
            try:
                x, y = move.split()
                states.append(states[-1].do_move(int(x), int(y)))
            except:
                print("Invalid move")
```

PLAYER

```
def move_human(state):

    while True:
        print(state)
        move = input(f"Player {state.player.value}: "
                     "x y to move, u to undo, "
                     "gN to revert to move N)? ")
        if move.startswith('u'):
            return Undo(1)
        elif move.startswith('g'):
            return RevertTo(int(move.replace('g', '')) + 1)
        else:
            try:
                x, y = move.split()
                return Move(int(x), int(y))
            except:
                print("Invalid move")
```

```
class Undo(namedtuple('_Undo', ['count'])):
    def apply(self, board_states):
        return board_states[:-self.count]

class Move(namedtuple('_Move', ['x', 'y'])):
    def apply(self, board_states):
        if board_states[-1].board[self.x][self.y] == Player.NA:
            return board_states + (
                board_states[-1].do_move(self.x, self.y),
            )
        else:
            return board_states

class RevertTo(namedtuple('_RevertTo', ['idx'])):
    def apply(self, board_states):
        return board_states[:self.idx]
```

```
class Undo(namedtuple('_Undo', ['count'])):
    def apply(self, board_states):
        return board_states[:-self.count]

class Move(namedtuple('_Move', ['x', 'y'])):
    def apply(self, board_states):
        if board_states[-1].board[self.x][self.y] == Player.NA:
            return board_states + (
                board_states[-1].do_move(self.x, self.y),
            )
        else:
            return board_states

class RevertTo(namedtuple('_RevertTo', ['idx'])):
    def apply(self, board_states):
        return board_states[:self.idx]
```

```
class Undo(namedtuple('_Undo', ['count'])):
    def apply(self, board_states):
        return board_states[:-self.count]

class Move(namedtuple('_Move', ['x', 'y'])):
    def apply(self, board_states):
        if board_states[-1].board[self.x][self.y] == Player.NA:
            return board_states + (
                board_states[-1].do_move(self.x, self.y),
            )
        else:
            return board_states

class RevertTo(namedtuple('_RevertTo', ['idx'])):
    def apply(self, board_states):
        return board_states[:self.idx]
```

TESTS

```
def test_revert(self):
    self.assertEqual(
        RevertTo(2).apply(self.states),
        self.states[:2]
    )

def test_inverse(self):
    start = (BoardState(), )
    for x in range(3):
        for y in range(3):
            self.assertEqual(
                Undo(1).apply(Move(x, y).apply(start)),
                start
            )
```

LOOP

```
player_types = [move_human, move_random]
players = {
    Player.X: player_types[x_choice],
    Player.O: player_types[y_choice],
}

states = (BoardState(), )
while not states[-1].is_finished:
    move = players[states[-1].player](states[-1])
    states = move.apply(states)
```

RANDOM

```
def move_random(state):
    x = randrange(3)
    y = randrange(3)

    return Move(x, y)
```

ITERATION

SEARCH

```
def depth_first(board=None):
    if board is None:
        board = Board()

    yield board

    for x, y in ALL_MOVES:
        if board.board[x][y] != Player.NA:
            board.do_move(x, y)
            try:
                yield from depth_first(board)
            finally:
                board.board[x][y] = Player.NA
```

Saaad... ☹

SEARCH

```
def depth_first(board=None):
    if board is None:
        board = Board()

    yield board

    for x, y in ALL_MOVES:
        if board.board[x][y] != Player.NA:
            old_board = [list(board.board[x]) for x in range(3)]
            board.do_move(x, y)
            try:
                yield from depth_first(board)
            finally:
                board.board = old_board
```

SEARCH

```
def depth_first(board=None):
    if board is None:
        board = Board()

    yield board

    for x, y in ALL_MOVES:
        if board.board[x][y] != Player.NA:
            old_board = [list(board.board[x]) for x in range(3)]
            board.do_move(x, y)
            try:
                yield from depth_first(board)
            finally:
                board.board = old_board
```

Saaad... ☹

SEARCH

```
def depth_first(state=None):
    if state is None:
        state = BoardState()

    yield state

    for x, y in ALL_MOVES:
        next_state = state.do_move(x, y)
        if state != next_state:
            yield from depth_first(next_state)
```

FILTER

```
def depth_first_filter(filter_fn, state=None):
    if state is None:
        state = BoardState()

    yield state

    next_states = (state.do_move(x, y) for x, y in ALL_MOVES)

    next_states = (next_state for next_state in next_states
                   if state != next_state)

    next_states = filter_fn(state, next_states)

    for next_state in next_states:
        yield from depth_first_filter(filter_fn, next_state)
```

FILTER FUNCTION

```
def filter_finished(state, next_states):
    if state.winner is not None:
        return
    else:
        yield from next_states
```

MAIN

```
winning_states = {
    Player.X: [],
    Player.O: [],
    Player.NA: [],
}
for state in depth_first_filter(filter_finished):
    if state.winner is not None:
        winning_states[state.winner].append(state)

print("O wins", len(winning_states[Player.O]))
print("X wins", len(winning_states[Player.X]))
print("Tie", len(winning_states[Player.NA]))
```

RESULTS

```
> python tictactoe_v8_all_games.py
O wins 77904
X wins 131184
Tie 46080
```

IMMUTABILITY ALLOWS LOCAL THINKING

TOOLS FOR LOCAL THINKING

- @property
- tuple (and namedtuple)
- Commands

QUESTIONS?

REFERENCES

Talk: bit.ly/immutable-python-pres

Source Code: bit.ly/immutable-python-src