Chapter 37
TicTacToe Game

37.1 Introduction

In this chapter we will explore the creation of a simple TicTacToe (or Noughts and Crosses) game using an Object Oriented approach. This example utilises:

- Python classes, methods and instance variables/attributes.
- Abstract Base Classes and an abstract method.
- Python Properties.
- Python lists.
- A simple piece of game playing logic.
- While loops, for loops and if statements for flow of control behaviour.

The aim of the game is to make a line of 3 counters (either X or O) across a 3 by 3 grid. Each player takes a turn to place a counter. The first player to achieve a line of three (horizontal, vertically or diagonally) wins.

37.2 Classes in the Game

We will begin by identifying the key classes in the game. Note that there is not necessarily a right or wrong answer here; although one set of classes may be more obvious or easier to understand than another.

In our case we will start with what data we will need to represent for our TicTacToe game as recommended back in the 'Introduction to Object Orientation' chapter.

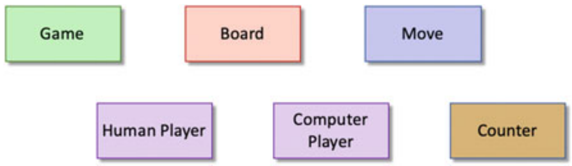
Our key data elements include:

- the tic-tac-toe board itself,
- the players involved in the game (both computer and human),
- the state of the game, i.e. whose go it is and whether someone has won,

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_37

- the moves being made by the players etc.
- the counters used which are traditionally O and X (hence the alternative name 'Noughts and Crosses').

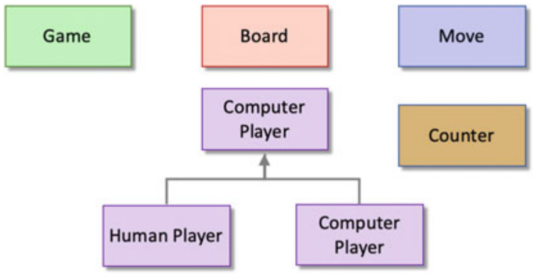
Based on an analysis of the data one possible set of classes is shown below:



In this diagram we have

- `Game` the class that will hold the board, players and the core game playing logic,
- `Board` this is a class that represents the current state of the TicTacToe board or grid within the game,
- `Human Player` this class represents the human player involved in the game,
- `Computer Player` this class represents the computer playing the game,
- `Move` this class represents a particular move made by a player,
- `Counter` which can be used to represent the counters to play with; this will be either X or Y.

We can refine this a little further. For example, much of what constituents a player will be common for both the human and the computer player. We can therefore introduce a new class `Player`, with both `Computer Player` and `Human Player` inheriting from this class, for example:

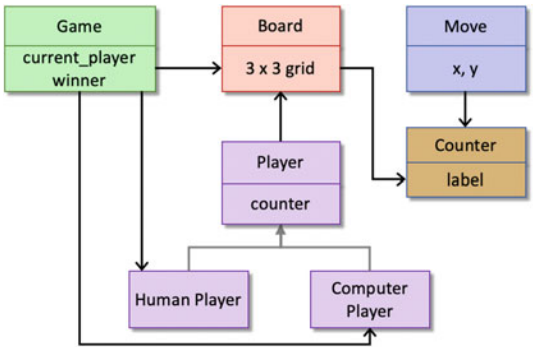


In terms of the data held by the classes we can say:

- `Game` has a board, a human and a computer player. It also has links to the current player and an attribute indicating whether a player has won.
- `Board` holds a 3 by 3 grid of cells. Each cell can be empty or contains a counter.
- `Player` Each player has a current counter and can see the board.

- Move represent a players selected move; it therefore holds the counter being played and the location to put the counter in.
- Counter holds a label indicating either X or O.

We can now update the class diagram with data and links between the classes:



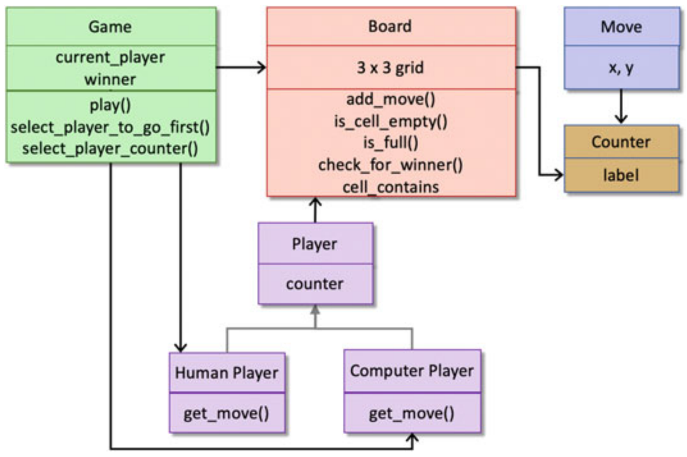
At this point it looks as though the HumanPlayer and ComputerPlayer classes are unnecessary as they do not hold any data of there own. However, in this case the behaviour for the HumanPlayer and the ComputerPlayer are quite different.

The HumanPlayer class will prompt the human user to select the next move. In contrast the ComputerPlayer class must implement an algorithm which will allow the next move to be generated within the program.

Other behavioural aspects of the classes are:

- Game this must hold the overall logic of the game. It must also be able to select which player will go first. We will also allow the human player to select which counter they will play with (X or O). This logic will also be placed within the Game class.
- Board The Board class must allow a move to be made but it must also be able to verify that a move is legal (as cell is empty) and whether a game has been won or whether there is a draw. This latter logic could be located within the game instead; however the Board holds the data necessary to determine a win or a draw and thus we are locating the logic with the data.

We can now add the behavioural aspects of the classes to the diagram. Note we have followed the convention here for separating the data and behaviour into different areas within a class box:



We are now ready to look at the Python implementation of our class design.

37.3 Counter Class

The `Counter` class is given below; it is a data oriented class, sometimes referred to as a *value* type. This is because it holds *values* but does not include any behaviour.

```
class Counter:
    """ Represents a Counter used on the board """


    def __init__(self, string):
        self.label = string

    def __str__(self):
        return self.label

# Set up Counter Globals
X = Counter('X')
O = Counter('O')
```

We have also defined two constants X and Y to represent the X and O counters used in the game.

37.4 Move Class

The Move class is given below; it is another a data oriented class or *value* type.

```
class Move:
    """ Represents a move made by a player """

    def __init__(self, counter, x, y):
        self.x = x
        self.y = y
        self.counter = counter
```

37.5 The Player Class

The root of the Player class hierarchy is presented below. This class is an *abstract* class in which the get_move() method is marked as being *abstract*. The class maintains a reference to the board and to a counter.

```
class Player(metaclass=ABCMeta):
    """ Abstract class representing a Player
        and their counter """

    def __init__(self, board):
        self.board = board
        self._counter = None

    @property
    def counter(self):
        """ Represents Players Counter - may be X or Y"""
        return self._counter

    @counter.setter
    def counter(self, value):
        self._counter = value

    @abstractmethod
    def get_move(self): pass

    def __str__(self):
        return self.__class__.__name__ + '[' +
str(self.counter) + ']'
```

Note that counter is defined as a Python property.

The class Player is extended by the classes HumanPlayer and ComputerPlayer.

37.6 The HumanPlayer Class

This class extends the abstract `Player` class and defines the `get_move()` method. This method returns a `Move` object representing the counter to be placed and the location in the 3 × 3 grid in which to place the counter. Note that the `get_move()` method relies on a reference being maintained by the player to the board so that it can check that the selected location is empty.

To support the `get_move()` method a `_get_user_input()` method has been defined. This method could have been defined as a stand alone function as it is really independent of the `HumanPlayer`; however it has been defined within this class to keep the related behaviour together. It also follows the Python convention by starting the method name with an underbar (_) which indicates that the method is private and should not be accessed from outside of the class.

```
class HumanPlayer(Player):
    """ Represents a Human Player and their behaviour """

    def __init__(self, board):
        super().__init__(board)

    def _get_user_input(self, prompt):
        invalid_input = True
        while invalid_input:
            print(prompt)
            user_input = input()
            if not user_input.isdigit():
                print('Input must be a number')
            else:
                user_input_int = int(user_input)
                if user_input_int < 1 or user_input_int > 3:
                    print('input must be a number in the range
1 to 3')
                else:
                    invalid_input = False
        return user_input_int - 1

    def get_move(self):
        """ Allow the human player to enter their move """
        while True:
            row = self._get_user_input('Please input the row:
')
            column = self._get_user_input('Please input the
column: ')

            if self.board.is_empty_cell(row, column):
                return Move(self.counter, row, column)
            else:
                print('That position is not free')
                print('Please try again')
```

37.7 The ComputerPlayer Class

This class provides an algorithmic implementation of the `get_move()` method. This algorithm tries to find the best empty grid location in which to place the counter. If it cannot find one of these locations free then it randomly finds an empty cell to fill. The `get_move()` method could be replaced with whatever game playing logic you want.

```
class ComputerPlayer(Player):
    """ Implements algorithms for playing game """

    def __init__(self, board):
        super().__init__(board)

    def randomly_select_cell(self):
        """ Use a simplistic random selection approach
        to find a cell to fill. """
        while True:
            # Randomly select the cell
            row = random.randint(0, 2)
            column = random.randint(0, 2)
            # Check to see if the cell is empty
            if self.board.is_empty_cell(row, column):
                return Move(self.counter, row, column)

    def get_move(self):
        """ Provide a very simple algorithm for selecting a
move"""
        if self.board.is_empty_cell(1, 1):
            # Choose the center
            return Move(self.counter, 1, 1)
        elif self.board.is_empty_cell(0, 0):
            # Choose the top left
            return Move(self.counter, 0, 0)
        elif self.board.is_empty_cell(2, 2):
            # Choose the bottom right
            return Move(self.counter, 2, 2)
        elif self.board.is_empty_cell(0, 2):
            # Choose the top right
            return Move(self.counter, 0, 2)
        elif self.board.is_empty_cell(0, 2):
            # Choose the top right
            return Move(self.counter, 2, 0)
        else:
            return self.randomly_select_cell()
```

37.8 The Board Class

The `Board` class holds a 3 by 3 grid of cells in the form of a list of lists. It also defines the methods used to verify or make a move on the board. The `check_for_winner()` method determines if there is a winner given the current board positions.

```
class Board:
    """ The ticTacToe board"""

    def __init__(self):
        # Set up the 3 by 3 grid of cells
        self.cells = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', '
', ' ']]
        self.separator = '\n' + ('-' * 11) + '\n'

    def __str__(self):
        row1 = ' ' + str(self.cells[0][0]) + ' | ' +
str(self.cells[0][1]) + ' | ' + str(self.cells[0][2])
        row2 = ' ' + str(self.cells[1][0]) + ' | ' +
str(self.cells[1][1]) + ' | ' + str(self.cells[1][2])
        row3 = ' ' + str(self.cells[2][0]) + ' | ' +
str(self.cells[2][1]) + ' | ' + str(self.cells[2][2])
        return row1 + self.separator + row2 + self.separator +
row3

    def add_move(self, move):
        """ A a move to the board """
        row = self.cells[move.x]
        row[move.y] = move.counter

    def is_empty_cell(self, row, column):
        """ Check to see if a cell is empty or not"""
        return self.cells[row][column] == ' '

    def cell_contains(self, counter, row, column):
        """ Check to see if a cell contains the provided
counter """
        return self.cells[row][column] == counter

    def is_full(self):
        """ Check to see if the board is full or not """
        for row in range(0, 3):
            for column in range(0, 3):
                if self.is_empty_cell(row, column):
                    return False
        return True

    def check_for_winner(self, player):
        """ Check to see if a player has won or not """
        c = player.counter
        return (# across the top
                (self.cell_contains(c, 0, 0) and
self.cell_contains(c, 0, 1) and self.cell_contains(c, 0, 2)) or
                # across the middle
                (self.cell_contains(c, 1, 0) and
self.cell_contains(c, 1, 1) and self.cell_contains(c, 1, 2)) or
                # across the bottom
                (self.cell_contains(c, 2, 0) and
self.cell_contains(c, 2, 1) and self.cell_contains(c, 2, 2)) or
                # down the left side
                (self.cell_contains(c, 0, 0) and
self.cell_contains(c, 1, 0) and self.cell_contains(c, 2, 0)) or
                # down the middle
                (self.cell_contains(c, 0, 1) and
self.cell_contains(c, 1, 1) and self.cell_contains(c, 2, 1)) or
                # down the right side
                (self.cell_contains(c, 0, 2) and
self.cell_contains(c, 1, 2) and self.cell_contains(c, 2, 2)) or
                # diagonal
                (self.cell_contains(c, 0, 0) and
self.cell_contains(c, 1, 1) and self.cell_contains(c, 2, 2)) or
                # other diagonal
                (self.cell_contains(c, 0, 2) and
self.cell_contains(c, 1, 1) and self.cell_contains(c, 2, 0)))
```

37.9 The Game Class

The Game class implements the main game playing loop. The play() method will loop until a winner is found. Each time round the loop one of the players takes a turn and makes a move. A check is then made to see if the game has been won.

```
class Game:
    """ Contains the Game Playing Logic """

    def __init__(self):
        self.board = Board()
        self.human = HumanPlayer(self.board)
        self.computer = ComputerPlayer(self.board)
        self.next_player = None
        self.winner = None

    def select_player_counter(self):
        """ Let the player select their counter """
        counter = ''
        while not (counter == 'X' or counter == 'O'):
            print('Do you want to be X or O?')
            counter = input().upper()
            if counter != 'X' and counter != 'O':
                print('Input must be X or O')
        if counter == 'X':
            self.human.counter = X
            self.computer.counter = O
        else:
            self.human.counter = O
            self.computer.counter = X

    def select_player_to_go_first(self):
        """ Randomly selects who will play first -
        the human or the computer."""
        if random.randint(0, 1) == 0:
            self.next_player = self.human
        else:
            self.next_player = self.computer

    def play(self):
        """ Main game playing loop """
        print('Welcome to TicTacToe')
        self.select_player_counter()
        self.select_player_to_go_first()
        print(self.next_player, 'will play first first')
        while self.winner is None:
            # Human players move
            if self.next_player == self.human:
                print(self.board)
                print('Your move')
                move = self.human.get_move()
                self.board.add_move(move)
                if self.board.check_for_winner(self.human):
                    self.winner = self.human
                else:
                    self.next_player = self.computer
            # Computers move
            else:
                print('Computers move')
                move = self.computer.get_move()
                self.board.add_move(move)
                if self.board.check_for_winner(self.computer):
                    self.winner = self.computer
                else:
                    self.next_player = self.human
            # Check for a winner or a draw
            if self.winner is not None:
                print('The Winner is the ' + str(self.winner))
            elif self.board.is_full():
                print('Game is a Tie')
                break
        print(self.board)
```

37.10 Running the Game

To run the game we need to instantiate the `Game` class and then call the `play()` method on the object obtained. For example:

```
def main():
    game = Game()
    game.play()

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

A sample output from running the game is given below in which the human users goes first.

```
Welcome to TicTacToe
Do you want to be X or O? X
ComputerPlayer[Y] will play first first
Computers move
   |   |
-----------
   | Y |
-----------
   |   |
Your move
Please input the row: 1
Please input the column: 1
Computers move
 X |   |
-----------
   | Y |
-----------
   |   | Y
Your move
Please input the row: 2
Please input the column: 1
Computers move
 X |   | Y
-----------
 X | Y |
-----------
   |   | Y
Your move
Please input the row: 3
Please input the column: 1
The Winner is the HumanPlayer[X]
 X |   | Y
-----------
 X | Y |
-----------
 X |   | Y
```