Lab-8

Object Oriented Programming

In this paradigm, entities are represented as real world data. For instance, we want to represent a dog. In OO paradigm, we simply create a class named “dog”, and give it attributes (colour, age, sex, etc) and behaviour (bark, run, eat, etc). Behaviour is changed through “methods” (functions in simple words) that make changes to attributes.

What makes OO programming powerful is its ability to do the following:

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Here’s the documentation: https://docs.python.org/3/tutorial/classes.html

About classes: https://python-textbok.readthedocs.io/en/1.0/Classes.html

OOP Principles:
https://python-textbok.readthedocs.io/en/1.0/Object_Oriented_Programming.html

Sample code on inheritance
https://gitlab.com/mugsiiit/iss-lab-code/tree/master/Lab-8

Example / Practice Exercise: (not to be submitted)

1. Create a Bank Account class that has the following behaviour: (fill out the functions and add data). Make it fully functional.

```
class BankAccount:
    """Bank Account protected by a pin number."""
    def __init__(self, pin):
        """Initial account balance is 0 and pin is 'pin'."""
    def deposit(self, pin, amount):
        """Increment account balance by amount and return new
balance."""
    def withdraw(self, pin, amount):
        """Decrement account balance by amount and return amount
withdrawn."""
    def get_balance(self, pin):
        """Return account balance."""
    def change_pin(self, oldpin, newpin):
        """Change pin from oldpin to newpin."""
```

(Hint: Read about encapsulation for the pin
https://docs.python.org/3/tutorial/classes.html#private-variables)

2. Create a SavingsAccount class that behaves just like a BankAccount, but also has an interest rate and a method that increases the balance by the appropriate amount of interest. 5% interest if the balance is <= $200, 10% if balance exceeds $500. Make sure the value is added to the balance. Create appropriate methods, as required.
   (Hint: Child class)
3. You can create a subclass of a class that is a subclass of another class and so on. Create a FeeSavingsAccount class that behaves just like a SavingsAccount, but also charges a fee of $1 every time you withdraw money. The fee should be deducted before each withdrawal. (Hint: function overriding)

The directory named “banking” contains solutions for the 3 qns.
https://gitlab.com/mugsiiit/iss-lab-code/tree/master/Lab-8

Exercise: (to be submitted)

To understand OOPs principles, today you’ll be making a terminal-based chess emulator! Yay?! (Don’t worry it’s easier than it sounds).

Aim: Create a program using the various aspects of object oriented programming, which can allow two players to play chess on your terminal.

Functional Requirements:
Arrangement of Pieces:

Normal Rules of chess:

Format: Piece(num_points x num_occurences = total)

- Pawns(1x8 = 8):
  - On it’s own first move, can move two steps or one step forward.
  - Can only move in the forward direction, if the square in front of it is empty.
  - Captures: By moving diagonal

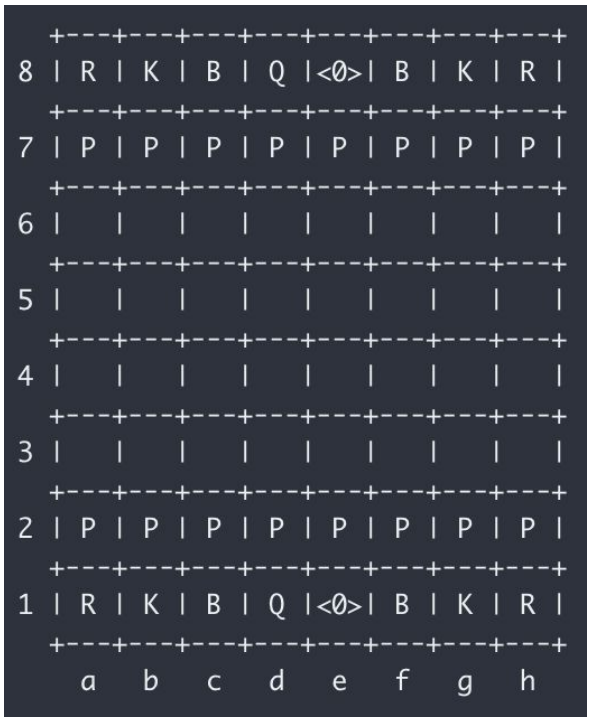
- Rooks(5x2 = 10):
    - Can move unlimited steps through empty squares horizontally or vertically
    - Captures: By moving horizontally or vertically
- Bishops(3x2 = 6):
    - Can move only diagonally(any number of squares) on the same color square that they start out on.
    - Captures: through diagonal movement
- Knights(3x2 = 6):
    - Can move in an L shape(2 steps vertical/horizontal followed by 1 step horizontal/vertical (**note the ordering**)
    - Captures: through the L movement
- Queens(9x1 = 9):
    - Can move unlimited(through empty squares) squares in vertical, horizontal and diagonally.
    - Captures: same as movement
- Kings(No points as capture => game over):
    - Move 1 step in any direction
    - Captures: 1 step in any direction

Events to be modelled:

- Castle (Both King-Side and Queen-side)
- Empassant(Check online for this rule)
- Check (king in attack)
- Checkmate - Game over

I/O Specs:

Board:



Print the board every time a move is made. Board should look something like the above image. Feel free to use a different structure/combination of '-', '|', ..etc to suit your default terminal fonts. Do however follow the rest of the conventions: Naming of pieces, and labelling of the sides.

Implementation Hints(Suggestions which do not need to be followed strictly):

- Make a class Board.
    - This class, when instantiated should initialize a grid(2d matrix) of size 8x8.
    - Make a method called to_print which will print out the current state of the board.
    - Populate the board, on initialization, with all the pieces(objects to be created, instructions below)
    - Create an update_board method which can update the state of the board(move things around). The update_board method could be what is called whenever a new move is made.

Input:

- Input will be of the form [a-h][1-8] [a-h][1-8] (ie. Original Location, New Location)
    - King Side Castle will be denoted by 0-0
    - Queen Side Castle will be denoted by 0-0-0
    - Example inputs:
        - e2 e4
        - h7 h6
        - 0-0-0
- Input is blocking, which means that the program waits(state of the board doesn't change), until an input is given.

Pieces:

- Create a large Piece class from which the rest of the Pieces can inherit.
    - Things that the Piece class should have:
        - Methods:
            - Move
            - Check_valid_move
        - Attributes:
            - Current Position