Classes, objects, inheritance

```
class Character: # Defines a class called 'Character'
  # Below we define the constructor
  def __init__(self, name, healthPts): # Always put 'self' as first argument of a method
    self.name = name
    self.hp = healthPts
    self.hpMax = healthPts
    self.alive = True


    def wounded(self,damagePts):
      self.hp = self.hp - damagePts
      if self.hp<0:
        self.hp = 0
        self.alive = False
        print(str(self)+" is dead")

  def healed(self,pts):
    if self.alive:
      self.hp = self.hp + pts
      if self.hp>self.hpMax:
        self.hp = self.hpMax
    else:
      print(str(self)
            +" cannot be healed since he is already dead")

  def __str__(self): # This method is called to obtain the result of the expression 'str(self)'
    return self.name+" HP:"+str(self.hp)+"/"+str(self.hpMax)
# End of class definition
mario = Character("Mario",30) # Creates a 'Character' object
#########################################################################################################
class Warrior(Character): # Defines a child class 'Warrior' that will inherit
                          # properties and methods from the 'Character' class
  def __init__(self,name, healthPts, attackPts):
    super().__init__(name, healthPts) # Calls the constructor of the parent class
    self.ap = attackPts

  def attack(self,otherCharacter):
    print(str(self)+" attacks "+str(otherCharacter))
    otherCharacter.wounded(self.ap)

  def __str__(self):
    return super().__str__()+" AP:"+str(self.ap)
# End of class definition
#########################################################################################################
class Priest(Character): # Defines another child class of 'Character'
  def __init__(self,name, healthPts, magicPts):
    super().__init__(name,healthPts)
    self.mp = magicPts

  def resurrect(self,otherCharacter):
    #...
#########################################################################################################
luigi = Warrior("Luigi",40,5) # Creates a 'Warrior' object
peach = Priest("Peach", 20,30) # Creates a 'Priest' object
#########################################################################################################
```

- In the following exercises, we will implement some two-player turn-based board games using Python classes and objects.
- You should finish at least exercises 1 to 4 to get a decent mark.
- Exercise 5 is an intermediate-level exercise where we implement the well-known Minmax algorithm which provides a general artificial intelligence for two-player deterministic board games.
- Exercise 6 is an extension the previous exercise where we leverage symmetries of a game in order to speed up computation in the Minmax algorithm.
- Your work should be handed in in pairs, as a single `.py` file. Questions requiring a written answer (in exercise 6) should be answered at the end of your code as a comment.

Exercise 1 (The board)

Define a class `Board` that represents a gameboard and implement the following methods:

1. A constructor `Board(size)` that initializes an empty board with `size` rows and columns.
2. A method `place(self,piece,square)` that places a piece at a given square of the board. Pieces will be represented by a string whose first character is either `w` (white) or `b` (black) to indicate the player they belong to (example: `wK` (knight) or `bQ` (queen) in a game of chess or simply `w` or `b` in a game of Tic-tac-toe or Checkers). Squares will be represented by a tuple of integers `(row,column)`. Rows and columns will be indexed starting at zero. If the given square is already occupied, this method returns `False`. Else it returns `True` if placement has succeeded.
3. A method `isEmpty(self,square)` returning `True` or `False` depending on whether the given square is empty or not.
4. A method `display(self)` that prints the board on the screen (in console mode).
5. A method `remove(self,square)` that removes the piece at the given square. Returns `True` or `False` depending on success.
6. A method `move(self,square1,square2)` that moves the piece at `square1` to `square2`. Will return `True` or `False` whether the movement has succeeded or not (ie: `square2` is already occupied, or there is no piece at `square1`).
7. A method `clone(self)` that returns a copy of the board.
8. A method `__eq__(self,other)` that tests where the current board is the same as `other`. Returns a boolean. With this method, two clones of the same board should be considered equal.

Exercise 2 (Game rules)

Consider the following class prototype:

```
class State:
  def __init__(self): # initializes the board (depends on the game)
    self.board = None # to be implemented...
    self.turn = "w" # indicates that it is the turn of the white player
    self.winner = None # Remains 'None' until the game has ended,
                       # in which case it is equal to 'w', 'b' or 'd' if
                       # there is a draw.

  def allowedMoves(self):
    raise NotImplementedError("This method is yet to be implemented") # It will return
```

```
                    # a set of moves allowed to the current player from this state
  def applyMove(self,move):
    raise NotImplementedError("This method is yet to be implemented") # It will compute
      # and return the state that is obtained by applying 'move' to the current state (or
      # None if failure)

  def display(self):
    raise NotImplementedError("This method is yet to be implemented") # It will display
      # the current state (for example: the board and whose turn it is)

  def clone(self):
    raise NotImplementedError("This method is yet to be implemented") # It will return
                                                                  # a copy of the state

  def __eq__(self,other):
    raise NotImplementedError("This method is yet to be implemented") # Tests whether two
      # states are the same. Returns a boolean. With this method, two clones of the same
      # state should be considered equal.
```

Moves can be represented by a tuple of the following form:

- either a tuple `("place",square, piece)`
- either a tuple `("move",square1,square2)`
- any other tuple `(action,...)` where `action` would be any action specific to the game. For example in Chess we would have an action `take` to take a piece of the opponent. In Checkers we would also have an action `take` but its effect would be different.

1. Implement `applyMove` for the class prototype `State`. `applyMove` should take care of the two first types of moves in the list above (`"place"` and `"move"`). It should also take care of changing turns.
2. Implement `display`, `clone` and `__eq__` for the class prototype `State`.
3. Write a child class `TicTacToeState` that implements the above abstract class for the game Tic-Tac-Toe. Use an attribute `self.SIZE` (by default: 3) to allow for different board sizes.

Exercise 3 (Players)

Consider the following class prototype:

```
class Player:
  def __init__(self,color):
    self.color = color # 'w' or 'b'

  def play(self,state): # Returns the move played in state 'state'
    raise NotImplementedError("This method is yet to be implemented")
```

1. Implement the method `play` in the class `Player` so as to choose a random move among all allowed moves. The class `Player` should be compatible with all games.
2. Write a child class `HumanPlayer` that inherits from `Player`. Method `play` will ask the user in which square he will place his piece. This class should be compatible with Tic-Tac-Toe.
3. Write a function `playGame(initialState,wPlayer,bPlayer)` where *initialState* is a `State` object, and `wPlayer` and `bPlayer` are two `Player` objects. This function will perform the main loop, ie: print the current state of the game, ask the current player for his move, apply the current player's move, and so on... until the game has ended.

Exercise 4 (Other games)

1. Consider the following "Stones" game:

   At the beginning, we have some random number of stones (say between 9 and 25). Each player takes in turn at least one, and at most three stones. The player who manages to obtain the last stone wins.

   Implement the class `StonesState` and `HumanStonesPlayer` for the "Stones" game. For this game, *moves* will be simply represented by an integer between 1 and 3.

2. Write `ConnectFourState` and `HumanConnectFourPlayer` for the Connect Four game. For this game, *moves* will be represented by an integer between 0 and $n_{\text{columns}} - 1$, this integer being the index the column where the next token will be dropped.

Exercise 5 (Minmax AI)

In this exercise we implement a class `MinmaxAIPlayer` that inherits from `Player` and uses the minmax algorithm to decide which move to play. This class should be compatible with all games.

We say a state s' is a *child* of state s if there is some allowed move that allows to go from state s to state s'. The *minmax* score of a state s is defined as follows:

- if s is a terminal state (ie: the game has ended), then $score(s) = 1$ if I have won, $score(s) = -1$ if my opponent has won, and $score(s) = 0$ if there is a draw.
- if s is a state where it is my turn to play, then $score(s) = \max\{score(s') \text{ where } s' \in \{\text{children of } s\}\}$.
- if s is a state where it is my opponent's turn to play, then $score(s) = \min\{score(s') \text{ where } s' \in \{\text{children of } s\}\}$.

1. Implement the class `MinmaxAIPlayer`. This class should possess a method `computeMinmax(self,state)` which computes the minmax score of `state`. You should use the *memoizing* technique: ie, whenever the score of some state is computed, it should be stored in a dictionary[1] for further reference, to avoid having to re-compute several times the score of a same state.

2. Test your `MinmaxAIPlayer` against other types of players, for the 3×3 Tic-Tac-Toe game, the 4×4 Tic-Tac-Toe game and the "Stones" game.

3. Modify your `MinmaxAIPlayer` to allow for a maximum computation time. Whenever the computation time exceeds some bound, the AI will simply choose a random move. [2]

Exercise 6 (Using symmetries to speed up computation time)

In this exercise, we implement the following idea to speed up computation times of the Minmax AI. Whenever some Tic-Tac-Toe state s' is related to another state s by a transformation γ (ie: $s' = \gamma(s)$) then they have the same minmax score, or opposite minmax scores. Here γ can be any of the following transformations:

- Symmetries with respect to a vertical, horizontal or diagonal axis.
- Rotations of 90, 180 or 270 degrees.
- Exchanging the two players.
- Any composition of the above.

[1] The keys of such a dictionary would be `State` objects. Because of the internal mechanics of Python, an object can be a key in a dictionary only if there is some hash code associated to it. Therefore you might need to write a method `__hash__(self)` for `State` objects that returns a hash code which you can determine however you wish. The only constraint is that two clones of a same state should have the same hash code.

[2] Write `import time` at the beginning of your code and use `time.time()`

We denote by Γ the set of such transformations. The *orbit of s under Γ*, denoted as $\Gamma(s)$, is defined as the set of all states we obtain by repeatedly applying transformations of Γ to the state s. In other words, elements of $\Gamma(s)$ are states that can be written as $\gamma_1 \circ \gamma_2 \circ \ldots \circ \gamma_n(s)$ where the γ_i are elements of Γ.

The idea is that once we know the score of s, we can easily deduce the score of all states that are in the orbit of s. This avoids unnecessary computations.

Now consider the following class prototype:

```
class AbstractTransformation:
  def __init__(self):
    pass

  def applyTo(self,x): # returns a new state, the result of
                       # applying the transformation to the state x
    return x.clone()
```

1. Implement classes `DiagonalSymmetry` and `Rotation90` which inherit from `AbstractTransformation` and which represent (respectively) the symmetry with respect to the diagonal of the board, and the rotation of the board of 90 degrees.

2. Implement a class `ExchangePlayers` which inherits from `AbstractTransformation` and which represents the transformation which consists of exchanging colors and turns.

3. Do we need any other transformations to obtain the whole orbit of a state ? Justify. And if it is the case, implement them.

4. Write a function `computeOrbit(state,transformations)` that computes the orbit of `state` under some set of transformations `transformations`. The return value should be a set of `State` objects.

5. Modify the class `MinmaxAIPlayer` so as to leverage symmetries of the game.
   - The constructor should take a list of transformations as an additional argument, under whose orbit the absolute value of the minmax score is left unchanged.
   - Whenever the minmax score of some state s is computed, the minmax score of all all states in the orbit of s should be simultaneously memoized to optimize computation times.

6. Compare computation times on 3×3 Tic-Tac-Toe, with and without using symmetries.

7. What transformation group Γ would we use for the "Stones" game ? and for Connect Four ?