SUPINFO Academic Dept.

SUPINFO International University

INSTITUTE OF INFORMATION TECHNOLOGY

Algorithms in Python

First Mini - Project

Two little strategy games

Version 1.0
Last update: 31/10/2017
Use: Students/Staff
Author: Laurent GODEFROY

SUMMARY

1 PREAMBULE

This exam is **individual**.

Any form of plagiarism or using codes available online or any other type of support, even partial, is prohibited and will cause a 0, a cheater mention, and even a disciplinary board.

You must upload an archive with the format « .zip » containing the source code of your project.

This mini-project won't be defended by a viva.

A grading **scale** is given at the end of this topic.

The goal of this mini-project is to program two little combinatory and abstract strategy games in Python.

2 ALAK

2.1 THE RULES OF THIS GAME

Important note: no codes are requested in this sub part. We will only explain the rules.

This version of the game Alak has been created in 2001 by Alan Baljeu[1]. Two players fight each other in a one-dimensional board of **n** squares. The first player has white pawns and the second one has black pawns. At the beginning of the game, all the squares are empty.

Whites begin, and then the two players will put alternately one of their pawns while respecting the following constraints :

1. A player can put a pawn on a square only if the square is empty and has not been occupied by one of his pawn during the previous turn (see the following rules of capture). If a player can not put a pawn anymore, the game is over.

[1] https://ca.linkedin.com/in/alan-baljeu-64788b8

2. If a group of adjacent pawns that belong to the adversary does not have adjacent squares after that a player has played, the group of pawns is captured and removed from the board. According to the first rule, the adversary will not be able to put a pawn on one of these squares during the next turn. The “group of pawns” must contain at least one pawn.
3. On the other hand, a player can put a pawn and create a group of adjacent pawns without empty adjacent squares and without that this group of pawns being captured.

When the game is over, the winner is the player which has the most of pawns. A draw match is possible.

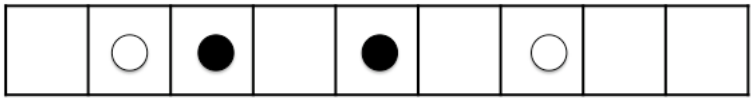
Here is an example of the beginning of a game. The game board has 9 squares :

The player 1 put a pawn on the square number 2 :

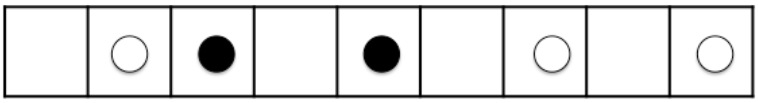
The player 2 put a pawn on the square number 3 :

The player 1 put a pawn on the square number 7 :

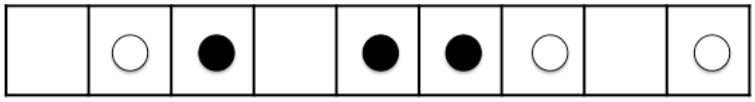
The player 2 put a pawn on the square number 5 :



The player 1 put a pawn on the square number 9 :



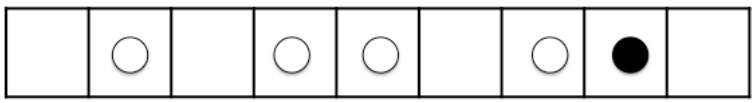
The player 2 put a pawn on the square number 6 :



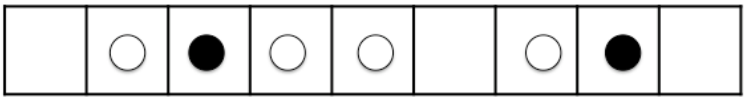
The player 1 put a pawn on the square number 4. Two groups of adjacent white pawns have not empty adjacent squares, so they are captured :

The player 2 is not able to put pawns on the squares number 3,5 and 6 (because they were occupied by his pawns during the previous turn). He chooses to put a pawn on the square number 8. Then, he captures the pawn of the player 1 that was in square number 9 because this pawn does not have empty adjacent squares :

The player 1 is not able to put a pawn on the square number n°9. He chooses to put a pawn on the square number 5 :



The player 2 put a pawn on the square number 3 (according to the third rules, he is able to do it):



And so on.

2.2 PROGRAMMING THIS GAME IN PYTHON

It is highly recommended to read this part before starting to code.

Important notes :

- We can possibly implement some subroutines in addition to those asked.
- The board will be a one-dimensional list of integers equals to 0, 1 or 2. An empty square will be represented by a 0, a pawn of the first player will be represented by a 1 and a pawn of the second player by a 2.
- Instead of re-calculating again and again the dimension of this list, we will passed in paramater the size of the list to our subroutines.
- To memorize the number of the squares where pawns have been captured in the previous turn, we will use a list that contains two list : the first list concerns the first player and the second list concerns the second player.
- The example of visual rendering is just indicative. You can improve it.

Notations of the subroutines' parameters that we will implement :

- « board » : a one-dimensional list of integers equals to 0, 1 or 2 that represents the game board.
- « n » : a strictly positive integer equal to the number of elements that contains the « board ».
- « player » : integer representing the player which it's the turn (example : 1 if it's the turn of the player 1 and 2 if it's the turn of the second player).
- « removed » : a list that memorizes the number of the squares where pawns have been captured during the previous turn.
- « i » : any integer.

Implement the following subroutines in a file called "alak.py" :

- A function « newBoard(n) » that returns a one-dimensional list that represents the initial state of the game board with **n** squares.
- A procedure « display(board,n) » that print the board on the python console. We will represent an empty square by a '.', a white pawn by a 'x' and a black pawn by a 'o'. After few turns, we should have a display like this one :

```
. x o . o o x . x
1 2 3 4 5 6 7 8 9
```

- A function « possible(board,n,player,removed,i) » that returns **True** if **i** is the index of a square where the player **player** is able to put a pawn and else it returns **False**.
- A function « select(board,n,player,removed) » that asks the player **player** to input a number of square where he is able to put a pawn. We will suppose that this square exists, so we won't test it here. As long as this number is not valid because of the rules, we will ask to input another number. Finally, the function will return this number.
- A procedure « put(board,n,player,removed,i) » where we suppose that **i** is the number of a square where the player **player** can put a pawn. This procedure realizes this drop and what it implies (capture, update of **removed**, etc.).
- A function « again(board,n,player,removed) » that returns **True** if the player **player** can put a pawn on the board and else it returns **False**.

- A function « win(board,n) » that returns a string that indicates the outcome of the game.
- A procedure « alak(n) » that will use the previous subroutines (and others if it's needed) to permits to two players to play a complete game on a board of **n** squares.

Here is the example of the subpart 2.1 again, but this time with our program and with a game over:

```
. . . . . . . . .
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 2

. x . . . . . . .
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 3

. x o . . . . . .
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 2
Choose a licit number of square : 10
Choose a licit number of square : 7

. x o . . . x . .
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 5

. x o . o . x . .
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 9

. x o . o . x . x
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 6
```

```
. x o . o o x . x
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 4

. x . x . . x . x
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 3
Choose a licit number of square : 5
Choose a licit number of square : 6
Choose a licit number of square : 8

. x . x . . x o .
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 9
Choose a licit number of square : 5

. x . x x . x o .
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 2
Choose a licit number of square : 3

. x o x x . x o .
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 1

x x o x x . x o .
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 6
```

```
x x o . . o . o .
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 5
Choose a licit number of square : 9

x x o . . o . o x
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 5

x x o . o o . o x
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 4

x x . x o o . o x
1 2 3 4 5 6 7 8 9

Player  2
Choose a licit number of square : 7

x x . x o o o o x
1 2 3 4 5 6 7 8 9

Player  1
Choose a licit number of square : 3

x x x x o o o o x
1 2 3 4 5 6 7 8 9

Winner : 1
```

2.3 A CIRCULAR BOARD

Implement another version of this game (we will keep the previous functional parts), where the first square of the board is adjacent with the last square of the board.

2.4 BONUS

Implement another version of this game (we will keep the previous functional parts), where we delete the third rules. It's mean that if a player puts a pawn and creates a group of adjacent pawns without empty adjacent squares, then the group is captured.

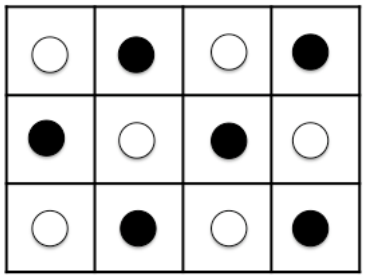
3 CLOBBER

3.1 THE RULES OF THIS GAME

Importantes notes : no codes are requested in this sub part, where we will explain the rules.

This game has been created by Michael Albert, J.P. Grossman and Richard J. Nowakowski in 2001. Two players fight each other on a rectangular board of **n** rows and **p** columns. The first player has white pawns and the second one has black pawns.

At the beginning of the game, the pawns are positioned like this (example with a board of 3 rows and 4 columns) :



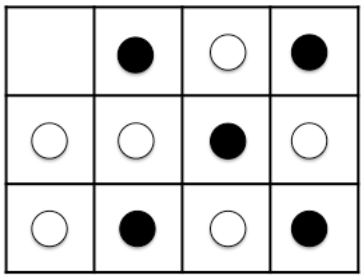
So, there is a white pawn in the square on the first row and first column, and then an alternation of white pawns and black pawns from square to square.

Alternately, players move one of their pawns to an adjacent square orthogonally that contains a pawn of the adversary and capture it. A player is not able to move his pawns diagonally, or move to an empty square, or to a square that already contains one of his pawns.

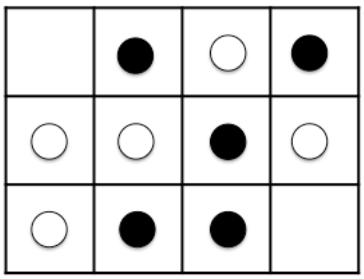
The winner is the last player able to move a pawn. In other words, as soon as a player is not able to move a pawn anymore, he loses.

We will give an example of game on a board of 3 rows and 4 columns.

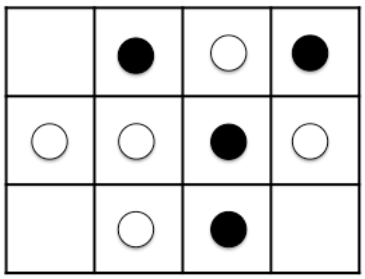
First player's turn :



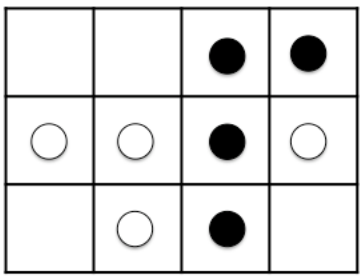
Second player's turn :



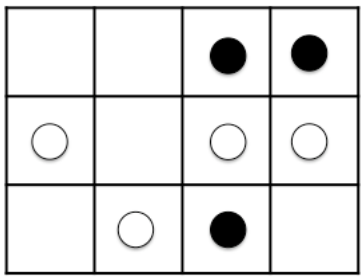
First player's turn :



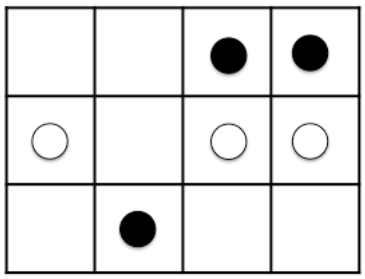
Second player's turn :



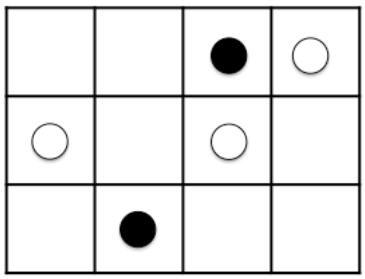
First player's turn :



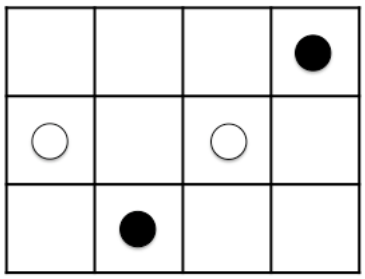
Second player's turn :



First player's turn :



Second player's turn :



The first player can not move any of his pawns. He loses.

3.2 PROGRAMMING THIS GAME IN PYTHON

It is highly recommended to read this part before starting to code.

Important notes :

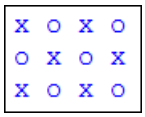
- We can possibly implement some subroutines in addition to those asked.
- The board will be a two-dimensional list of integers equal to 0, 1 or 2. An empty square will be represented by a 0, a pawn of the first player will be represented by a 1 and a pawn of the second player by a 2.
- Instead of re-calculating again and again the dimension of this list, we will passed in paramater the size of the list to our subroutines.
- The example of visual rendering is just indicative. You can improve it.

Notations of the subroutines' parameters that we will implement :

- « board » : a two-dimensional list of integers equals to 0, 1 or 2 that represents the game board.
- « n » : strictly positive integer equal to the number of rows of the « board ».
- « p » : strictly positive integer equal to the number of columns of the « board ».
- « player » : integer that represents the player where it's the turn (for example, 1 for the first player and 2 for the second).
- « i » : any integer.
- « j » : any integer.

Implement the following subroutines in a file called "clobber.py" :

- A function « newBoard(n,p) » that returns a two-dimensional list representing the initial state of a game board with **n** rows and **p** columns.
- A procedure « display(board,n,p) » that realises the display of the board on the python console. We will represent an empty square by a '.', a white pawn by a 'x' and a black pawn by a 'o'. initially, a board of 3 rows and 4 columns will be print like this :



- A function « possiblePawn(board,n,p,player,i,j) » that returns **True** if **i** and **j** are the coordinates of the a pawn that the player **player** can move. Else it returns **False**.
- A function « selectPawn(board,n,p,player) » that aks the player **player** to input the coordinates of a movable pawn. We will suppose that this square exists, so we won't test it

here. As long as the coordinates are not valid because of the rules, we will ask to input another two numbers. Finally, the function will return the coordinates.

- A function « possibleDestination(board,n,p,player,i,j,k,l) » where we suppose that **i** and **j** are the coordinates of a pawn that the player **player** can move. This function returns **True** if **k** and **l** are the coordinates of an adversary's pawn that the player **player** is able to capture with his pawn of coordinates **i** and **j**, else it return **False**.
- A function « selectDestination(board,n,p,player,i,j) » where we suppose that **i** and **j** are the coordinates of pawn that the player **player** can move. This function aks the player **player** to input the coordinates of an adversary's pawn that he is able to capture with the pawn of coordinates **i** and **j**. We will suppose that this pawn exists, so we won't test it here. As long as these coordinates won't be valid because of the rules, we will ask to the player to input the coordinates. Finally, the function will return these coordinates.
- A function « again(board,n,p,player) » that returns **True** if the player **player** is able to move one of his pawns. Else, it returns **False.**
- A procedure « clobber(n,p) » will use the previous subroutines (and others if it's needed) to permits to 2 players to play a complete game on a board of **n** rows and **p** columns.

Here is the example of the subpart 3.1, but this time with our :

```
x o x o
o x o x
x o x o

Player 1 :
Select a pawn, row : 1
Select a pawn, column : 2
Select a pawn, row : 1
Select a pawn, column : 1
Select a destination, row : 2
Select a destination, column : 1

. o x o
x x o x
x o x o

Player 2 :
Select a pawn, row : 3
Select a pawn, column : 4
Select a destination, row : 3
Select a destination, column : 3

. o x o
x x o x
x o o .

Player 1 :
Select a pawn, row : 1
Select a pawn, column : 1
Select a pawn, row : 1
Select a pawn, column : 5
Select a pawn, row : 3
Select a pawn, column : 1
Select a destination, row : 3
Select a destination, column : 2
```

```
. o x o
x x o x
. x o .

Player 2 :
Select a pawn, row : 1
Select a pawn, column : 2
Select a destination, row : 1
Select a destination, column : 3

. . o o
x x o x
. x o .

Player 1 :
Select a pawn, row : 2
Select a pawn, column : 2
Select a destination, row : 2
Select a destination, column : 3

. . o o
x . x x
. x o .

Player 2 :
Select a pawn, row : 3
Select a pawn, column : 3
Select a destination, row : 3
Select a destination, column : 2

. . o o
x . x x
. o . .

Player 1 :
Select a pawn, row : 2
Select a pawn, column : 4
Select a destination, row : 1
Select a destination, column : 4
```

```
. . o x
x . x .
. o . .

Player 2 :
Select a pawn, row : 1
Select a pawn, column : 3
Select a destination, row : 1
Select a destination, column : 4

. . . o
x . x .
. o . .

Winner :  2
```

3.3 A CANNIBAL VERSION

Implement another version of this game (we will keep the previous functional parts), where the players can capture one of their own pawn in addition of those of the adversary.

4 GRADING SCALE

This grading scale is only **indicative** because it can change.

- Part 2 : 20 points
- Part 3 : 20 points

It makes a total of 40 points. The total grade will then be brought proportionally to a grade based on 20 points.