CSCI 235, Programming Languages, Python 2

Deadline: Sunday 20.10.2019, 21.00

Goal of this exercise is to get some experience with the basics of Python.

- **Read the complete task before starting!**
- **Submit the final answer into Moodle in a single file nim.py. Additional information must be put in Python comments**
- **Code that shows evidence of lazyness, not having read the complete task, incomplete testing, breaking good coding practice, are likely to result in loss of credit!**
- **Submitted code must be written by yourself. We may test for plagiarism**
- **You will be asked to demonstrate and explain your program during the lab. Unability to run or explain submitted code will raise suspicion.**
- **Avoid** `break` **and** `continue`. In **structured programming** the control flow of the program must be immediately visible from the lay out. Usage of `break` and `continue` causes control flow to be different from lay out. Returning from the middle of a function is OK.

Download the file **nim.py** as a starting point.

1. Add a method `__repr__( )` to `class Nim` that returns the state as a string that can be shown. If you type `st = nim.Nim([1,2,3])`, the output must look like

   ```
   1   : 1
   2   : 1 1
   3   : 1 1 1
   ```

2. Add a method `sum( self )` to `class Nim` that returns the complete number of sticks in the current game state.

3. Add a method `nimber( self )` to `class Nim` that returns the current nimber. The nimber is the xor of the numbers of sticks in the piles. The xor-operator is `^`

4. Add a method `randommove( self )` to `class Nim` that makes a random move. Use `random.randrange( )` to get a random number. The must must always make a correct move when there are sticks When more than one move is possible, it must choose a move randomly. Repeated calls must result in different moves.

   When it cannot make a move, because there are no sticks left, it must raise `CantMove( "no sticks left" )` .

5. Add a method `removelastmorethantwo( self )` to `class Nim`, which makes a move if there is exactly one row containing more than one stick. In that case, it should reduce this row in such a way that an odd number of rows containing exactly one stick remain.

   If this method cannot make a move, it should raise
   `CantMove( "more than one row has more than one stick" )`

6. Create a method `usermove( self )` that asks the user to make a move. It asks the user to select a row, and how many sticks must remain in this row.

   (a) If the user types something that is not a number, the system should tell this.

   (b) If the user types a row that does not exist, or is empty, the system should tell this.

   (c) If the user tries to leave a negative number of sticks, tries to add sticks, or remove no sticks at all, the system should tell this.

   The method must keep on asking until the user manages to make a correct move. It must never crash on incorrect input.

7. Create a method `makenimberzero( self )` that tries to make a move that makes the nimber zero. This is possible iff the nimber is currently non-zero. In case more than one such move is possible, it must randomly select one. (We check this.) Randomly select a row `r`, and check if `( state[r] ^ nmb ) < state[r]`. If yes, the move can be made. If not, select another row.

8. Create a method `optimalmove( self )` that makes an optimal move. It calls `removelastmovethantwo`, `makenimberzero`, and `makerandommove` until one of them does not raise an exception.

9. Add the function

```
def play( ) :
   st = Nim( [ 1, 2, 3, 4, 5, 6 ] )

   turn = 'user'
   while st. sum( ) > 1 :
```

```
   if turn == 'user' :
      print( "\n" )
      print( st )
      print( "hello, user, please make a move" )
      st. usermove( )
      turn = 'computer'
   else :
      print( "\n" )
      print( st )
      print( "now i will make a move\n" )
      print( "thinking" )
      for r in range( 15 ) :
         print( ".", end = "", flush = True )
         time. sleep( 0.1 )
      print( "\n" )

      st. optimalmove( )
      turn = 'user'

print( "\n" )

if turn == 'user' :
   print( "you lost\n" )
else :
   print( "you won\n" )
```

The initial state is winning for the user. As soon we make a mistake, your implementation must win.