Fundamentals of Computer Science - programming exam

Instructions

IMPORTANT - BEFORE YOU START:

1. Open and edit the `.py` file in Spyder to solve the exercises.
2. Upload the `.py` file to BBoard.

Additional notes

- The exam is **not open book**! No notes, smartphones, pen drives, ...
- **Clean your mess** after you finish! Comments and tests/asserts are welcome, but unnecessary code should be deleted.
- **Tests are provided** for each exercise, although they are initially commented out. They are very helpful to understand the intended behavior and to check the correctness of your implementation. Make sure to uncomment and run them as you go though the exercises..

Part 1 (30min)

Note: For the exercises in part 1, do not make use of libraries, python's built-in functions, sets, dictionaries, list's methods, etc.. unless otherwise stated. Just plain loops and if/else statements.

Ex. 1.1 - P-Norm

The standard Euclidean norm of a vector $\mathbf{x}$ (of length n) is defined as the square root of the sum of the squares of
the elements, $\|\mathbf{x}\| = \sqrt{\sum_{i=0}^{n-1} x_i^2}$.
To make the translation into Python easier, we have defined the elements as x_i with i ranging from 0 to $n-1$, included.

The generalization of this is the p-norm, in which one takes the p-th root of the sum of the p-th power of the absolute value
(Python function: `abs` from `math` library) of the elements:

$$\|\mathbf{x}\|_p = \left(\sum_{i=0}^{n-1} |x_i|^p \right)^{1/p} \tag{1}$$

Of course, the Euclidean norm is just the 2-norm with this terminology.

Write a function called `pnorm` that takes two arguments, a list of numbers and the parameter `p` , and returns the
p-norm of the elements of the list. The parameter p should be optional, and default to 2 if not given.

The function should check that the first argument is a list, and that the second argument is a strictly positive integer or float, and
give an error otherwise.

Ex. 1.2 - Run-length (RL) encoding

Run-length (RL) encoding is a transformation of a sequence (for us here, a string) into another sequence (for us here, a list). It works by taking the first sequence and counting the consecutive "runs" of each symbol. For example:

```
"AAABBCAA" ➡ [3, 'A', 2, 'B', 1, 'C', 2, 'A']
```

You can think of it as reading it out-loud: "three A's, two B's, one C, two A's". So you have an alternation of "run" lengths and "run" items. (For long "runs", this can save a lot of space!)

Your task is to perform the inverse transformation: write a function called `rl_decode` that takes a list `l` as an argument, representing a RL-encoded list (like the one on the right above), and returns the original string.

An empty list as input results in an empty string as output.

Tip: Build the output string incrementally, one character at a time. Don't worry about efficiency.

Example of the desired behavior:

```
In [1]: rl_decode([3, 'A', 1, 'B', 2, 'C', 1, 'D'])
Out[1]: 'AAABCCD'# 3 As, 1 B, 2 Cs, 1 D

In [2]: rl_decode([3, '1', 4, '5'])
Out[2]: '1115555'# 3 ones, 4 fives

In [3]: rl_decode([])
Out[3]: ''
```

Ex. 1.3 - Max Occurrencies

Write a function called `max_occurrences` that takes a list of integers as its only argument and returns the number and the value of the element that occurs the most in the list.

Examples of the desired behavior:

```
In [1]: l = [1, 2, 3, 2, 1, 2, 3, 2, 4]
In [2]: max_occurrences(l)
Out[2]: (4, 2)

In [3]: l = [1, 1, 2, 2, 2, 3, 3, 3, 3]
In [4]: max_occurrences(l)
Out[4]: (4, 3)
```

Note: If there are multiple elements with the same maximum occurrences, return any of them.

Part 2 (50 min)

Ex. 2.1 - Table

We implement a class named `Table` representing data arranged in a tabular structure, similarly to an Excel's table or to a pandas' DataFrame.

We denote with `t` an object of the `Table` class. A table object stores the data in a field named `data`, a dictionary whose keys
are the column's names and whose values are lists containing the columns' elements.

We will implement some of the typical operations performed on tables as methods of our `Table` class. Let's start!

1. Define the class `Table` and its constructor that takes an empty argument list. The constructor should initialize the following 3 internal fields:
   - `data`, initialized to an empty `dict`. It will contain the `column name` -> `column values` associations.
   - `nrows` initialized to 0. It represents the number of rows.
   - `ncols` initialized to 0. It represents the number of columns.

   After this, you should be able to create an empty table as follows:

   ```
   t = Table()
   ```

2. Define the method `t.add_column(name, values)` for adding a column to the table. `name` is assumed to be a string representing the column name, `values` a list containing columns values. After the call, `t.data` will contain the new key `name` with corresponding value `values`. Moreover, the field `ncols` is incremented by 1 in the operation, and the field `nrows` is set to the length of `values`.
3. Extend the method `t.add_column(name, values)` adding a few consistency checks. The method will raise an exception if any of the following conditions applies:
   - There is already a column with name `name`.
   - `values` is not a list.
   - The length of `values` is different from `nrows`, unless both `nrows` and `ncols` are 0 (i.e. this is the first column being added).
4. Define the method `t.columns()` returning a list of column names.
5. Define the method `t.remove_column(name)` removing the column `name` from `data` and updating the `ncols` fields. If `name` is not an existing column, raise an exception.
6. Define the special method `__getitem__` to return the column values if `t[name]` is executed.
7. Define the special method `__setitem__` to add a new column when `t[name] = values` executed, or to replace an existing one if `name` is already a column (here, as in any of the exercises, you can exploit previously defined methods).

Ex. 2.2 - CheckerBoard

We implement a class named `CheckersBoard` representing the board in a checkers game.

The board has n^2 squares arranged in n rows and n columns, with $n = 8$ in most game's variants.

We will denote with $i = 0, 1, \ldots, n-1$ the row index and with $j = 0, 1, \ldots, n-1$ the column index.

During a game, a square can be either empty (represented by the integer 0), or occupied by a Player1 piece (represented by the integer 1) or by a Player2 piece (represented by the integer 2).

In order to track the state of the game, our board will contain a field `state`, a list of lists such that `state[i][j]` corresponds to the state (either 0, 1, or 2) of the square in row `i` and column `j`.

Let's start!

1. Define the special method `__init__`. The construct should take an argument `n` with default value `8`. It will initialize a field `state` as a list of length `n` whose elements are lists of length `n` representing the rows. The inner lists in `state` should be initialized to all zeros (empty board). Store also `n` as an internal field.

   ```
   board = CheckersBoard(8)
   board = CheckersBoard() # equivalent to above
   ```

2. Define the method `board.add_piece(i, j, player)`. The method assumes `player` to be equal to 1 and 2, it will modify the `state` of the board and will return a boolean value denoting whether the operation was successful or not:
   - If the board's `state` in position `(i, j)` is not empty, return `False` without changing the `state`.
   - If the position `(i, j)` is empty, assign that square to `player` changing `state` accordingly and return `True`.

   ```
   board = CheckersBoard()
   ok = board.add_piece(1,1, player=1)
   assert ok
   assert board.state[1][1] == 1
   ```

3. Define the method `board.remove_piece(i, j)`.
   - If the square `(i, j)` is already empty, just return `False`
   - If the square `(i, j)` is not empty, set it to empty (i.e. `0`) and return `True`.
4. Define the special method `__repr__`. After defining the method, the code

   ```
   board = CheckersBoard()
   board.add_piece(1, 1, player=1)
   board.add_piece(6, 2, player=2)
   print(board)
   ```

   will produce the output

```
CheckersBoard:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 2, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
```

Hint: you can use `str(row)` to convert each row in `state` to a string.

5. Define the method `board.move_piece(i, j, inew, jnew)`. The method will try to move a piece from position `(i, j)` to position `(inew, jnew)`.
   - If the position `(i, j)` is empty or if position `(inew, jnew)` is occupied just return `False` without changing the `state`.
   - Otherwise, change the `state` moving the piece to the new position and return `True`.

```
board = CheckersBoard()
board.add_piece(1,1, player=1)
ok = board.move_piece(1, 1, 2, 2)
assert ok
assert board.state[1][1] == 0  # old position is now empty
assert board.state[2][2] == 1  # new position is occupied by the player
```

6. Define the method `board.start_game()`. Set the `state` of the board to all zeros, except for
   - The first 3 rows, which will contain Player1's pieces at alternating positions.
   - The last 3 rows, which will contain Player2's pieces at alternating positions.

   The exact `state` obtained should be the one observed in the output below:

```
board = CheckersBoard()
board.start_game()
print(board)
```

```
CheckersBoard:
[0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 2, 0, 2, 0, 2, 0]
[0, 2, 0, 2, 0, 2, 0, 2]
[2, 0, 2, 0, 2, 0, 2, 0]
```