General information

For each question, paste your solution in the text field under the description.

- All answers should be provided as working Python code
- You should perform your own tests to verify that the code is correct. However test cases need not be provided with your answer. If you do provide test cases, you should identify those clearly (for example with comments).
- For maximum credits, the provided answer should be in the best possible complexity class. Less efficient answers will be given partial credits.
- It is not necessary to comment on your code, it is more important that the code is clean and easy to read. Comments are good to document assumptions that you make, if any.

For a 2-question exam.

The maximum possible total number of points is 35. 14 points correspond to grade 3, 21 points correspond to grade 4, and 28 points correspond to grade 5.

For a 3-question exam.

The maximum possible total number of points is 60. 24 points correspond to grade 3, 36 points correspond to grade 4, and 48 points correspond to grade 5.

Allowed aids:

- You can refer to your 1-page of handwritten notes.
- You can use the Self-Practice website in sandbox mode to test your code. From the Safe Exam Browser, you should click the "Python Sandbox" button on the bottom left of your screen to open it. This will open the self-practice website that we used in the course in another window. Note that your code won't be saved if you switch exercise in the self-practice website, or if you close or reload its window. Therefore it is advised to have one Self-Practice window open for each question and switch between windows using Alt+Tab or Option+Tab. Every time you click on "Python Sandbox", a new window will appear.
- You can refer to the "python reference" link at the top of the Self-Practice. Again, it is advised to open it in a separate window.

- Questions are provided in Swedish for convenience, but in case of discrepancy the English version applies. You can use the (so-called "hamburger") menu to choose the language of questions.

Disallowed aids: anything else. In particular, it is not allowed to use the following electronic tools.

- AI tools (Copilot, ChatGPT, Llama, ...)
- Communication tools (phone, chat, mail, social media, ...)
- Electronic documentation (Google, stack overflow, python guides or manuals, ...)
- Books
- Notes in printed or electronic format

Q1 Escape! (15 points)

Consider a spherical planetoid of mass M and radius r. Consider further an inertial object at the surface of the planetoid, moving with a velocity V directly away from the planetoid. If the object moves slowly, it will fall back onto the planetoid. Otherwise, it will be able to escape its gravitational field. If so it will continue to slow down, approaching but never quite reaching its hyperbolic excess speed e.

The minimum velocity to escape is called `v`, and is given by the following equation:

$$v^2 r = 2GM$$

The hyperbolic excess speed `e` is given by the equation:

$$e^2 = V^2 - v^2$$

Write a function `escape`, taking as arguments `M`, `r` and `V`, which tests if the object will escape the gravitational field. If it escapes, `escape` should print the hyperbolic excess speed. Otherwise it should print "the object won't escape".

Assume $G = 6.674 \times 10^{-11}$

Example run:

```
>>> escape(7.3e22, 1.7e6, 5000) # moon example
Excess speed: 2394.1153345848056

>>> escape(6e24, 5.6e6, 5000) # earth example
The object won't escape
```

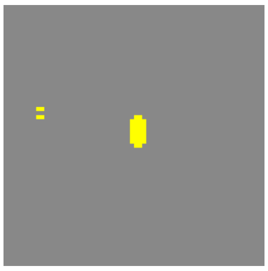
Answer

```python
import math
G = 6.674e-11
def escape(M,r,V):
   v2 = 2*G*M / r
   e2 = V**2 - v2
   if e2 > 0:
       print("Excess speed:", math.sqrt(e2))
   else:
       print("The object won't escape")

# moon example
escape(7.3e22, 1.7e6, 5000)
# earth example
escape(6e24, 5.6e6, 5000)
```

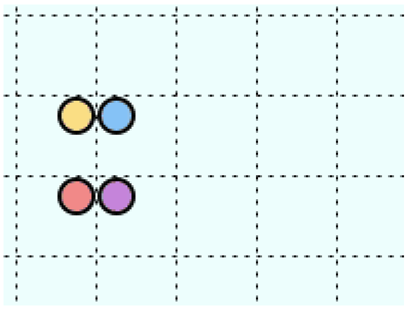
Q2 Single-Rotation (20 points)

You will implement the "single rotation" cellular automation. A cellular automaton is a function that changes the state of a matrix (or grid) of cells. A cell can either be occupied or empty. By iterating this function, one can observe interesting behavior. The following picture shows the evolution of a example matrix of cells, under the single-rotation automaton.



Click the picture to see a moving GIF

We will represent the matrix of cells as a list of list. An empty cell will contain `None`. Other cells may contain any other value. Our automaton works by conceptually dividing the matrix in 2 by 2 groups, thus forming groups of 4 cells. The only effect of the automation is to rotate a group of four cells **if and only if** the group has a single occupied cell. Other groups are left alone. For even iterations, the groups are divided across grid lines of even index. For odd iterations, the groups are divided across grid lines of odd index. Here is an example of several iterations:



Click the picture to see a moving GIF

Attention:

- Assume that a matrix is represented by a list of rows, themselves represented as lists. All rows have equal length. The size of the matrix is arbitrary, but has at least one element. There is no sharing of rows in a matrix.
- An empty cell is represented by `None`. Any other value represents an occupied cell.
- The division in groups might leave some groups incomplete near the edge of the grid. For this exercise, incomplete groups will never be rotated. You will have to detect this situation specifically.
- In the above picture rotation happens always clockwise. You can rotate in the other direction, but all rotations must be in the same direction.

Tasks:

- Define the function `is_group_complete(m,i,j)`, which determines if the cell group located in matrix `m` starting at position `i`, `j` is complete. To clarify, a complete group is one that has its four cells within the boundaries of the whole grid. Return the result as a boolean. (5 points)

- Define the function `is_single_group(m,i,j)`, which tests if the cell group located in matrix `m` starting at position `i`, `j` has exactly one occupied cell and returns the result as a boolean. This function assumes that the group is complete. (5 points)
- Define the function `rotate_group(m,i,j)`. This function must **modify** the matrix `m` by rotating a group located in matrix `m` starting at position `i`, `j`. (5 points)
- Define the function `iteration(n,m)` which **modifies** the matrix by rotating all the groups in the matrix `m` according to the single-rotation rule defined above, for a single iteration. The number `n` is the iteration we're at. (5 points)

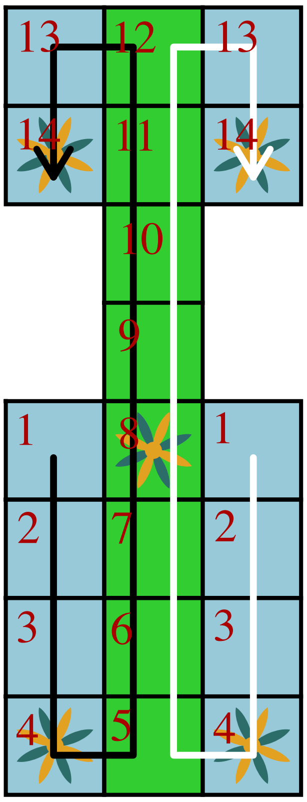
Answer

q2.py

Q3 The Royal Game of Ur (25 points)

The Royal Game of Ur is an ancient 2-player board game. You will model the following simplified rule set for the game (adapted from Wikipedia):

- The game is played using two sets of seven identical game pieces, similar to those used in draughts or checkers. One set of pieces is white and the other set is black. The gameboard is composed of two rectangular sets of boxes, one containing three rows of four boxes each and the other containing three rows of two boxes each, joined by a "narrow bridge" of two boxes.



- The gameplay involves elements of both luck and strategy. Movements are determined by throwing four coins, each yielding either zero or one point. Summing the points yields a result between 0 and 4 included. Such a throwing of coins is called hereafter a "roll". For instance if a player gets the results (0,0,1,1) for each of coins, we say that they "rolled a 2".
- The objective of the game is for a player to move all seven of their pieces first onto the board, then along the course and finally off the board before their opponent does so. When a piece is on one of the player's own squares, it is safe from capture. When it is on one of the eight squares in the middle of the board, the opponent's pieces may capture it by landing on the same space, sending the piece off the board. (Such a piece then must restart the course from the beginning.) This means there are six "safe" squares and eight "combat" squares. There can never be more than one piece on a single square at any given time.
- When a player rolls the number *n*, they may choose to move any of their pieces on the board forward by *n* squares, or place a new piece to the board at location *n* if they still have pieces that have not entered the game. A player may also pass. At most one piece can be moved in a single turn. (When /n/=0 and no piece has entered the board, then passing is the only option.)
- In order to remove a piece from the board, a player must roll exactly the number of spaces remaining until the end of the course

plus one. If the player rolls a number any higher or lower than this number, they may not remove the piece from the board. Once a player removes all their pieces off the board **in this manner**, that player wins the game.

You will implement a class `Ur` which manages the state of the game as it is being played.

Your class should have the following attributes:

- `cur_player` , an int in the [0,1] interval, representing the player currently playing
- `rolled` , an int representing the result of the previous roll, or None if no roll occured yet.
- `board` , a list of lists, such that `board[p][l]` contains the string "x" if there is a piece of player `p` at location `l` , or `None` otherwise.
- `not_started` , a list of ints, such that `not_started[p]` is the number of pieces still in the hand of player `p` (pieces which are not on the board yet, or have been pushed off in "combat").
- `arrived` , a list of ints, such that `arrived[p]` is the number of pieces of player `p` which managed to get off the board.

The class should have the following methods. Each method correspond an action of a player.

- `__init__(self)` : initialize the attributes for the initial game state. (6 points)
- `roll(self)` : simulate the coins flip for the current player and print the outcome. You can assume that this function will be called exactly once at the beginning of each player turn. Hint: use the function `randint(low,high)` from the `random` module. (3 points)
- `pass_action(self)` : don't make a move, and yield to the other player. (And wait for their roll) (1 point)
- `move(self,loc)` : move the piece of the current player located at the location `loc` , by the amount rolled previously. (15 points) The location is either a number between 1 and 14, as indicated in the above figure, or zero. If it is zero, then the player move is to bring a new piece onto the board. The target square depends on the selected source location and the previous roll. Structure the function as follows:

    - compute the target location for the moved piece (1 point)
    - verify if the target location is valid (2 points)
    - verify if the target square isn't occupied by a piece of the current player (2 points)
    - remove the piece from the specified source location (2 points)
    - handle "combat" if applicable (4 points)
    - place the piece in the target location (3 points)
    - print a message if the player has won (1 points)
    - or yield to the other player (0 point)

At any point, if an invalid move is attempted, do not make any change to the game state. Instead, print a suitable message and wait for another action from the same player.

Answer

q3.py