Representing Grids

Mathematically, a *grid* is a partition of a 2D region into a disjoint collection of *cells*. Typically, these cells are all a single simple shape such as a square, triangle or hexagon. Several mini-projects, including 2048, Zombie Apocalypse, and the Fifteen puzzle, involve rectangular grids of squares. Grids are useful in many computational applications because they provide a convenient way to partition a geometric region in a way that can be easily modeled as a 2D data structure.

Indexing rectangular grids

For now, we focus on rectangular grids that are composed entirely of squares. Mathematically, such a grid has a height **`grid_height`** and a width **`grid_width`**, measured in terms of individual squares. The standard terminology when referring to the size of such a grid is to specify height first, followed by width. For example, a three-by-four grid is three squares high and four squares wide.

When working with such grids, we will index individual squares in the grid in the same manner that entries in a matrix are indexed, top to bottom then left to right. In particular, a square is indexed by its row i and its column j in the grid where the row index i lies in the range $0, 1, ..., height - 1$ and the column index j lies in the range $0, 1, ..., width - 1$. This program produces a diagram in CodeSkulptor that illustrates this indexing scheme.

When compared to canvas coordinates, this matrix-based indexing scheme transposes the horizontal and vertical dimensions. For example, note that the coordinates given to **`draw_polygon`** in the diagram-plotting program linked above order the column index first and the row index second when generating vertices of each square drawn in the grid. For now, you don't need to worry about this issue since the GUIs that we provide for each mini-project handle this transposition without any effort on your part.

Modeling rectangular grids in Python

Given a square, we can store the index associated with a square as the tuple (row, col) in Python. Then, we can represent some relevant property of that square as the entry **`cells[row][col]`** in the 2D list **`cells`**. Note the expression **`cells[row]`** returns a 1D list corresponding to a row of the grid. We can initialize this 2D list via the code fragment:

```
cells = [ [... for col in range(grid_width)] for row in range(grid_height)]

```

Note that if **`row`** or **`col`** are not used in the expression **`...`**, Pylint will warn that these variables are unused. To avoid this warning, you can rename these variables to **`dummy_row`** and **`dummy_col`** to alert Pylint that these variables are intentionally unused. This renaming will suppress the warning.

Mark as completed