CSci 1133
Lab Exercise 6

Multidimensional Lists

The majority of imperative languages provide syntactic support for linear structures with more than one dimension. These structures are useful for representing "tabular" form data such as spreadsheets, general matrices, *n*-dimensional discrete spaces and the like.

Python supports multidimensional structures through the use of *nested* lists. In other words, a two dimensional table or matrix is represented as a *list* of rows in which each individual row is another *list* of column values. This "list within a list" is an important concept that is employed in many computational situations.

Indexing the individual values in a multidimensional structure is straightforward: To retrieve the m^{th} row (a list!), you simply provide the row index:

```
somematrix[m]
```

To obtain the n^{th} column value in the m^{th} row, you would index the row list:

```
somematrix[m][n]
```

Warm-up

1). **Fun With Matrices**

Write a pure function that will construct and return a *square* matrix of order *n*:

```
def matrix(n,init)
```

Recall that a *square* matrix is one with an equal number of rows and columns, and the *order* of a square matrix is equal to the number of rows/columns. The function accepts two arguments: the *order* of the matrix and an initial value, and returns a two-dimensional list (rows of columns) with each element set to the initial value.

[Hint: Start with an empty list and use nested while (or for) loops to construct each row list and add it the list of rows.]

Thoroughly test your function to make sure it works for various orders and initial values.

2). **More Fun With Matrices**

Write a pure function that will accept any square matrix as a multidimensional list (rows of columns) argument and determine and return the *order* of the matrix:

```
def order(m):
```

Stretch

1). **Identity Matrix**

The *identity* matrix of order *n* is an *n* x *n* matrix in which all of the elements are 0 except for the major diagonal elements, which are all 1. The major diagonal elements are [0][0], [1][1], [2][2], ... , [*n*-1][*n*-1].

Write a pure function that will take a single value *n*, and return an identity matrix of order *n*:

```
def identity(n):
```

Be sure to test your function to make sure it works!

2). **Sparse Matrices**

A *sparse* matrix contains few non-zero values relative to the total size of the matrix. Write a pure function that will take a matrix (two-dimensional list) as an argument and populate it with randomly placed elements:

```
def populate(m,n,value):
```

Where `n` is an integer containing the number of elements to be generated, and `value` is some constant value that will be written to each randomly assigned matrix location. The values should be evenly distributed throughout the entire matrix, `m`.

Hint: use the matrix function you created in part 2.

Workout

1). **Grid World**

Turtle graphics is built using the Python *TkInter* graphics library, which is quite sophisticated and very fast. Turtle drawing is intentionally "slowed down" so that you can visualize it as it takes place. By employing a few "tricks", you can greatly speed things up and build graphical interfaces that are much more robust and functional. The first trick involves turning off the display refresh operation while you draw the image, and then "updating" the display all at once. Doing this will speed up complex image drawing by several orders of magnitude! You need to use two *screen* object methods:

```
screen.tracer(0)
screen.update()
```

In order to use them, you first obtain a "screen" object using the turtle method `getscreen()`:

```
screen = turtle.getscreen()
```

then turn the tracer off and do all of your drawing. When you're ready to update the display, use the `screen.update()` method to cause everything you've just drawn to be displayed.

Write a non-pure function (procedure) that will accept a sparse matrix as an argument and visualize it using Turtle graphics:

```
def showMatrix(turtle_object, m):
```

The `showMatrix` function will visualize the matrix contents using a grid of dots. Each grid location will correspond to a single matrix location (row, column). The presence of a "dot" indicates a non-zero entry.

First, you need to set the display coordinates to match the matrix extent using the `screen.setworldcoordinates()` method. In other words, the lower left corner of the display will become coordinate (0,0) and the upper right corner will be (rows-1, columns-1). Changing the screen coordinates in this way simplifies the mapping of matrix indices to screen coordinates by matching the "grid" and matrix coordinates.

Using the turtle `.goto` and `.dot` methods, plot a red dot for each matrix location that is nonzero. Be sure to use the `tracer(0)` and `update()` methods to speed up the process.

Test your procedure using an identity matrix of order 100.

Also test your procedure by creating a 50x50 square matrix and populating it with 500 random points (use the functions you created earlier).

Challenge

1). **Conway's Game of Life**

In the early 1970's, the British mathematician John Conway described an entertaining simulation game based on John VonNeumann's *cellular automata* concepts. You can find a complete description of the game and its simple rules here:

http://en.wikipedia.org/wiki/Conway's_Game_of_Life

A cellular automaton is a discrete system in which simulated "organisms" reside on a two-dimensional grid and self-replicate or perish according to a few simple rules. Each grid cell contains zero or one organisms, represented by the integer values 0 and 1 respectively. Starting with a specific (or random) population of organisms, the simulation proceeds in discrete time-steps. At each time step, the status of each cell in the grid is determined by its 8 immediate neighbors according to the following rules:

1. If the cell contains an organism and has fewer than 2 organisms as neighbors, it perishes from "loneliness"
2. If the cell contains an organism with 2 or 3 immediate neighbors, it remains alive
3. If the cell contains an organism and has more than 3 immediate neighbors, it perishes from overcrowding
4. If the cell is unoccupied and has *exactly* three live neighbors, a new organism is spawned in that cell

For this problem, you will construct a working simulation of Conway's Game of Life.

You've already created most of what you will need in the Warm-up and Stretch portions of the lab, but you will need a few more pieces:

Write a pure function named `neighbors` that will take three arguments: a two-dimensional grid (matrix) and the row and column values of a specific cell in the grid, and return the number of adjacent cells that contain live organisms. Note that each cell in the grid contains the value 1 or 0 indicating the presence/absence of a live organism. Do not count the cell that is being interrogated, just the 8 immediately adjacent cells:

`[r-1][c-1]`	`[r-1][c]`	`[r-1][c+1]`
`[r][c-1]`	`[r][c]`	`[r][c+1]`
`[r+1][c-1]`	`[r+1][c]`	`[r+1][c+1]`

Also, for boundary cells (i.e., row 0, col 0, etc.) treat the cells that are "off the grid" as containing zeros.

Now, write another pure function named `update` that will take a two-dimensional grid (matrix) with the current configuration of organisms and construct/return a *new* grid based on the 4 rules outlined above (do not modify the existing grid! can you explain why?). You will need to repeatedly call the `neighbors` function to determine the number of neighbors.

To complete the Game of Life, write a procedure named `life` that will do the following:

- Create a grid (matrix) with 100 rows and 100 columns containing all zeros
- Initialize the grid with a population of live organisms (it will be more effective if you "cluster" them near the middle)
- To run the game, repeat the following for 50 cycles (or use text input to "single step" the simulation):
    1. Clear the current display
    2. Display the current grid
    3. Update the grid

2). **More Life**

Do the following:

- Extend the `neighbors` function to "wrap" around the grid (i.e., the cells "above" row zero would be on the bottom of the grid, etc.)
- Create an initialization function that will create a random population in the middle of the grid with a specified radius from the middle cell.