Introduction to Programming Homework 10

Due Friday Dec 2

You will turn in your homework **via GitHub**! Please use this link to start your repository :

https://classroom.github.com/assignment-invitations/f4ecae0b54f26dfec384a2306f8793f8

Exercise 1 (Arrays)

Write a module called `array_fun.py`

- **a.** Read the end of Lecture 10 starting with the slicing and views section.

For the rest of this exercise, don't use loops!!!

- **b.** Learn about the `array.sum()` instance method. Given a square 2-dimensional `numpy` array `M`, we say that `M` is a **stochastic (transition) matrix** if the sum of each **column** of `M` is equal to `1.0`.
  - write a function called `is_STM(A)` that checks if a `numpy` array `A` is a 2-dimensional square stochastic transition matrix.
  - write a function called `make_STM(A)` which takes a `numpy` array `A` and turns it into a stochastic transition matrix if possible by making each column sum to `1.0`. If `A` is not a square 2-dimensional `numpy` array, raise an error.

- **c.** Learn about the function `numpy.unique()`. Write a function called `most_frequent(A)` which takes an array `A` and returns (any) most frequent entry.

- **d.** Learn about the function `numpy.bincount()`. Write a **one line** function `array_with_bincount(C)` which takes a 1-dimensional array `C` of integers and returns an array `A` such that `numpy.bincount(A)` is equal to `C`.

- **e.** Write a **one line** function `distance_matrix(points)` which
  - takes an array `points` of shape `(k,d)` representing `k` points in $\mathbb{R}^d$
  - returns a matrix `D` such that `D[i,j]` is the Euclidean distance between `points[i]` and `points[j]`.

- **f.** Write a function called `planar_to_polar(points)` which
  - takes an array `points` of shape `(k,2)` representing `k` points in $\mathbb{R}^2$ in **Cartesian coordinates**.
  - returns an array of shape `(k,2)` representing `k` points in $\mathbb{R}^2$ in **polar coordinates** where the first column is the radius.

- **g.** Write a function called `polar_to_planar(points)` which
  - takes an array `points` of shape `(k,2)` representing `k` points in $\mathbb{R}^2$ in **polar coordinates** where the first column is the radius.
  - returns an array of shape `(k,2)` representing `k` points in $\mathbb{R}^2$ in **Cartesian**

coordinates.

- **h.** Use fancy indexing to write a function `swap_cols(A,i,j)` which will take a **2-dimensional array** `A` and swap column `j` with column `i`. You don't need to return anything as you should **modify** the original array `A`.

- **i.** Define a (nested) structured data type `color_point` where one has a `'position'` given as `('x','y')` and a `'color'` given as `('r','g','b')`. Write a function `random_color_points(num_points)` which generates a random 1-dimensional array of length `num_points` of `color_points`. For example, you should be able to run

```
A = random_color_points(10)
print(A.shape == (10,))
print(A['position'])
print(A['position']['x'])
print(A['color']['r'])
```

Exercise 2 (Game of Life)

For this exercise, you will implement John Conway's Game of Life. See Wikipedia for animations and examples.

Imagine a grid or board of "cells" where black cells are "alive" and white cells are empty/dead.

The game runs in discrete time. At every step, this set of rules is used to produce the next version of the board :

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by over-population.
- Any empty/dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Note that each cell has at most `8` neighbors.

We will represent a game grid by a 2-dimensional (integer) array `B` of `1`'s and `0`s with shape `(m, n)`. To make it easier to implement the game, we will assume out board `B` has **walls** of `0`'s around the perimeter. This means that at all stages of the game the top, bottom, right and left edges of `B` are always `0`'s.

(Note : you can think of this as just playing the game on the middle grid `B[1:-1, 1:-1]`.)

- **a.** Write a function called `neighbor_count(B)` which returns a 2-dimensional array `N` of shape `(m-2, n-2)` such that `N[i,j]` is the number of neighbors of `B[i+1, j+1]`. **Do not use loops !**

- **b.** Learn about how binary operators `|` and `&` work for boolean arrays. Use these to write functions :
  - `births(N,B)` - returns a boolean array showing births
  - `survive(N,B)` - returns a boolean array showing survivors
  - `deaths(N,B)` - returns a boolean array showing deaths

- **c.** Write a function called `game_of_life(B, num_steps)` which will return a `3`-dimensional array `G` of shape `(num_steps+1, m, n)` such that `G[0,:,:]` is `B` and `G[i+1,:,:]` is the next step in the game after `G[i,:,:]`.

Remark : you can check your game just by inspecting `print(G[i,:,:])` for `i = 0, ..., num_steps + 1` to see if everything is working.