Lab 1: Introduction to Python

Learning objectives:

- Practice writing a simple application in Python
- Gain experience reading a language specification

This lab provides a start on learning Python, which is necessary for projects 1, 4, and 5. You will also read sections of the Python reference, which will give you practice in reading and understanding language specifications, something you will be doing throughout the rest of the semester.

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab01/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Python practice.* Implement a simulation of Conway's Game of Life in Python3. While the original simulation is on an infinite grid, your implementation should be on a finite grid. Edge cells have fewer neighbors but should otherwise follow the same rules as any other cell.

   Starter code is in `life.py`, and a test of a full simulation is in `life_test.py`.

   a) Implement the `make_grid()` function, which constructs a grid with the given number of rows and columns and the given live cells. Use the NumPy library to represent the grid. (The quickstart is a good place to start learning about the library.)

   The data are stored in a $(rows+2) \times (cols+2)$ array so that we don't need special cases for the borders when simulating a timestep. (The additional cells are called *ghost cells*, and in our implementation, they always contain zeros.)

   Run the provided doctests via the following at the command line:

   ```
   $ python3 -m doctest life.py
   ```

   Tests for other functions will fail, since you haven't implemented them yet, but the tests for `make_grid()` should pass.

   b) Complete the `timestep()` function, which performs a single timestep of Conway's Game of Life. The first argument is the input grid, which should not be modified, and the second argument is the output grid.

   Hint: A lookup table can be helpful here. For each interior (i.e. non-ghost) cell, count the number of live neighbors it has, then look up the new value of the cell in a table that maps each possible count of neighbors to the correct new value.

   c) Implement the `simulate()` function. It should start by constructing two grids. Then for each timestep, it should:

   - Call `timestep()` to compute the new grid values.
   - Swap the two grids.
   - Use `print_grid()` to print out the values in the new grid.

   Keep in mind that Python has reference semantics, so that an assignment `a = b` does **not** make a copy of an object -- rather it modifies `a` so that it points to the same object as `b`.

   You can swap two variables `a` and `b` as follows:

   ```
   a, b = b, a
   ```

   This modifies `a` to refer to what `b` previously referred to, and `b` to point to the object that was previously referenced by `a`.

   We have provided a full test case in `life_test.py` and the expected output in `life_test.correct`:

   ```
   $ python3 life_test.py | diff - life_test.correct
   ```

2. *Python special methods and iterators.* Read through the section in the Python reference on special methods, particularly the subsections on basic customization and emulating container types. Also read the section in the reference on iterators. Then complete the definition of the `Range` class, which is a simplified version of Python's built-in `range` type, in `range.py`.

   a) The `Range` class is a container representing a fixed sequence of integers. It has the following methods:

      - `__init__()`: The constructor, which takes in the start, stop, and optional step. If the step is not given, it defaults to 1. We will only handle the case where the step is a positive integer.
      - `__iter__()`: Returns an iterator over the range, represented as a `RangeIter` object.
      - `__len__()`: Returns the number of integers in the range.
      - `__contains__()`: Returns whether or not the given integer is a member of the range.

      Implement the `__len__()` and `__contains__()` methods.

   b) The `RangeIter` class implements the iterator interface and is used for iterating over a `Range`. Implement the `__next__()` method.

   Once you have completed both classes, you can run the tests in `range_test.py` from the command line:

   ```
   $ python3 range_test.py | diff - range_test.correct
   ```