Assignment 8 Python - Social Graphs

Due Apr 12, 2016 by 12pm **Points** 95 **Submitting** an external tool
Available until Apr 16, 2016 at 12pm

This assignment was locked Apr 16, 2016 at 12pm.

- **http://rice.codeskulptor.org/#comp130_assignment_social_graphs.py (http://rice.codeskulptor.org/#comp130_assignment_social_graphs.py)**
- **Test (http://codeskulptor.appspot.com/owltest/?urlTests=comp130.assignment_social_graphs_tests.py&urlPylintConfig=comp130.pylint_config.py)**

(5 points) Code style beyond what is automatically checked. Variable names make sense. Your code isn't overly complicated. Use list comprehensions where applicable instead of loops.

Graphs Classes

1. In class, you defined the `Graph` class using adjacency lists (or, more accurately, adjacency sets). Include that definition in the provided file, except rename the class as `Graph1`.
2. **(50 points total — 12 points for `add_node`, 12 points for `add_edge`, 2 points for `get_nodes`, 12 points for `get_node_neighbors`, 12 points for `get_edges`)**

   Complete the `Graph2` class definition. This class represents graphs as adjacency matrices.

   Hint: If the graph currently has *n* nodes, then `self.names` has length *n*, and `self.adjmatrix` has *n* rows and *n* columns. So, when adding a node, `self.names` should get one larger, and `self.adjmatrix` should get one larger in both dimensions. However, adding an edge doesn't change the size of the representation, except that the `add_edge(name_from, name_to)` method is defined as adding a node to the graph if it doesn't already exist, and the process of adding a new node will increase the size of matrix.

A Bit of Machinery

We have two graph classes with the same class interface. Any graph-using code we now write should work with either. To facilitate that, add the following definition:

```
def make_empty_graph():
    """
    This 'factory' function is a graph-implementation-independent way of instantiating a graph object.
    This function always returns an empty graph object.
    """
    return Graph1()
```

Your later code should call `make_empty_graph()`, rather than directly calling either graph class constructor. This has a couple advantages

- When testing the later code, you can easily change `Graph1` to `Graph2` in this function, to switch between graph definitions.
- It more clearly decouples your later code from a particular graph class. In effect, your code will say "I want to create an empty graph.", rather than saying "I want to create an empty `Graph1` graph."

There are more elegant ways of doing this bit of machinery. While we made sure that `Graph1` and `Graph2` have the same interface, from Python's perspective, that's merely an implicit happy coincidence. To explicitly declare that `Graph1` and `Graph2` share the same class interface would require introducing a little bit more of Python — class inheritance, which we won't be doing in this course.

Social Graphs

Think of a social graph for a stereotypical high school. Nerds have friends who are other nerds, and jocks have friends who are other jocks. There is no friendship that crosses this social divide. This is an example of a partition. We can separate the nodes/people of the high school social graph into two smaller sets of nodes/people such that there are no edges/friends between the two sets.

3. **(20 points total)**

   Write a function `is_partition(graph, nodes1, nodes2)` that takes a graph object and two sets of nodes. The graph object can be an instance of `Graph1` or `Graph2` (or, any other class with the same methods). It returns whether the given node sets partition the graph. I.e., the two node sets are disjoint, non-empty, but include all the graph nodes, and no graph edge crosses between the two sets.

4. **(5 points)**

   Write a function `connect_all(graph, nodes)` that takes a graph. It modifies the graph by adding edges between all the nodes, except for self-edges. It also adds any of these nodes that are not already in the graph. For easier testing, it also returns the modified graph.

   This will be similar to the `make_complete()` function from class, but instead of creating a new graph, it changes an existing graph.

5. **(5 points)**

   Write a function `create_graph_and_partition(nodes)` that returns three values: a graph consisting of the given nodes, and two sets of nodes. Those two sets of nodes should form a partition of the graph. You can assume that the input is a set of length at least 2.

   Note that the output of this function should be easily testable by your `is_partition()` function.

6. **(5 points)**

   Write a function `create_graph_without_partition(nodes)` that returns a graph consisting of the given nodes. The graph should not be able to be partitioned. You can assume that the input is a set of length at least 1.