Project #1 – Python Programming

NAYA – Data Scientist Professional

# Background

## Terminology

This project is concerned with the implementation of mathematical graphs and their usages. The basic definition of a graph is a set of nodes where some pairs of nodes are connected by edges, and where these edges have weights. Each edge of the graph has a direction so by default a graph is directional.

If there is a directed edge from node A to node B, then we say that B is connected to A (or B is directly reachable from A). Note that this does not necessarily mean that A is neighbor of B (or A is reachable from B).

If it is possible to “travel” from a node A to node B (either directly or through any number of consecutive neighbors), then we say that B is reachable from A and that there is a path connecting A to B. The weight of a path is the sum of the weights of the edges from which the path is made of.

we say a graph is non-directional if all the edges come in “pairs” (namely, if X connected to Y, then Y is also connected X, with the same weight). i.e., there is no meaning to the direction of the edge.

## Illustration

The figure above shows a graph with 10 nodes and 20 edges. Make sure you understand the following statements:

* The graph is directional, although it does include a single pair of nodes that are neighbors of each other.
* Node “5” (“5” is the name of the node) is connected to node “1” with a weight of 20, while node “1” is not connected “5”.
* “1” is reachable only from “8” and “9”.
* “6” is reachable from “9” by several paths, and the minimal weight of such a path is 30.
* There is a path from “2” to “6” ("6" is reachable from "2"), but there is no path from “6” to “2” ("2" is not reachable from "6")

# Part I – The Node class

## Task 1 – Define the class

Implement the Node class with the following properties:

* Attributes
  + name – the “name” of the node
    - name can be any immutable object, most naturally a string or a number. for simplicity, you can assume name is string.
    - neighbors – a dictionary of the nodes that are connected to "self" (meaning the nodes that are directly reachable from "self"). the keys of the dictionary are the nodes' names and the values of the dictionary are the weights of the corresponding edges.
* Methods
  + \_\_init\_\_(self, name)
    - The method does not have to test the validity of name.
  + \_\_str\_\_(self)
  + \_\_len\_\_(self) – returns the number of neighbors
  + \_\_contains\_\_(self, item) – returns whether item is a name of a neighbor (connected noded) of self.
  + \_\_getitem\_\_(self, key) – returns the weight of the neighbor named key. If there is no such neighbor, then the method returns None.
  + \_\_eq\_\_(self, other) – based on the name attribute
  + \_\_ne\_\_(self, other)
  + **Bonus:** add rich comparison methods (lt, le, ge, ge), based on the number of neighbors
  + is\_neighbor(self, name) – equivalent to \_\_contains\_\_().
  + update(self, name, weight) – adds name as a neighbor of self.
    - If name is not a neighbor of self, then it should be added.
    - This method should not allow adding a neighbor with the same name as self.
    - If name is already a neighbor of self, then its existing weight should be updated to new weight
    - **Bonus**: update the weight to the maximum between the existing weight and the new (input) weight.
  + remove\_neighbor(self, name) – removes name from being a neighbor of self.
    - This method should not fail if name is not a neighbor of self.
  + is\_isolated(self) – returns True if self has no neighbors (i.e. there is no node connected to self)

\* note that you can add any auxiliary function you want

## Task 2 – Exemplary usage

### Question 1

Create 10 Node objects according to the figure above, print them (textually, of course).

### Question 2

Make some tests to make sure your implementation works.

### Question 3

How many edges are in the graph, and what is their total weight?

### Question 4

Sort the nodes by the number of their neighbors.

# Part II – The Graph class

## Task 1 – Define the class

Implement the Graph class with the following properties:

* Attributes
  + name – the name of the graph.
  + nodes – this is a dictionary fully descriptive of the graph. Its keys are the names of the nodes, and its values are the nodes instances (of class Node).
* Methods
  + \_\_init\_\_(self, name, nodes=[ ])
    - nodes is an iterable of Node instances.
    - Note: the **input nodes is an iterable** (preferably list) of Node instances, while the **attribute nodes is a dictionary**. This is not a mistake.
  + \_\_str\_\_(self)
    - This method should print the description of all the nodes in the graph.
    - Tip: the built-in function print() is not the only function that calls the Node.\_\_str\_\_() method, but also the built-in function str().
  + \_\_len\_\_(self) – returns the number of nodes in the graph
  + \_\_contains\_\_(self, key) – returns True if key is there is a Node named key in self (in the graph).
    - you can assume key is string.
    - **Bonus**: return True in two cases: (1) If key is a string, then if a node called key is in self, and (2) If key is a Node, then if a node with the same name is in self. Tip: use the built-in function isinstance().
  + \_\_getitem\_\_(self, name) – returns the Node object whose name is name.
    - This method should raise KeyError if name is not in the graph.
  + **Bonus**: \_\_add\_\_(self, other) – returns a new Graph object that includes all the nodes and edges of self and other
    - This method applies the same logic as update\_node().
  + update(self, node) – adds a new node to the graph
    - node is a Node instance.
    - If a node with the same name already exists in self, then this method should update the relevant information with the same logic as Node.update().
    - If node has neighbors that are not already nodes in self, then this method should not create the relevant nodes.
  + remove\_node(self, name) – removes the node name from self.
    - This method should not fail if name is not in self.
    - This method should not remove edges, in which name is a neighbor of other nodes in the graph.
  + is\_edge(self, frm\_name, to\_name) – returns True if to\_name is a neighbor of frm\_name.
    - This method should not fail if either frm\_name is not in self.
  + add\_edge(self, frm\_name, to\_name, weight) – adds an edge making to\_name connected to of frm\_name.
    - This method applies the same logic as Node.update().
    - This method should not fail if either frm\_name or to\_name are not in self.
  + remove\_edge(self, frm\_name, to\_name) – removes to\_name from being connected to frm\_name.
    - This method should not fail if frm\_name is not in self.
    - This method should not fail if to\_name is not connected to frm\_name.
  + get\_edge\_weight(self, frm\_name, to\_name) – returns the weight of the edge between frm\_name and to\_name.
    - This method should not fail if either frm\_name or to\_name are not in self.
    - This method should return None if to\_name is not connected to frm\_name.
  + get\_path\_weight(self, path) – returns the total weight of the given path, where path is an iterable of nodes’ names.
    - This method should return None if the path is not feasible in self.
    - This method should return None if path is an empty iterable or an iterable that contain only single node.
    - Tip: The built-in functions any() and all() regard nonzero numbers as True and None as False.
  + auxiliary function:

def all\_paths(self):

from itertools import chain, combinations, permutations

s = list(self.nodes.keys())

nodes\_sets = list(chain.from\_iterable(combinations(s, r) for r in range(2, len(s)+1)))

paths = list(chain.from\_iterable((permutations(nodes\_set) for nodes\_set in nodes\_sets)))

return paths

* + is\_reachable(self, frm\_name, to\_name) – returns True if to\_name is reachable from frm\_name.
    - This method should not fail if either frm\_name or to\_name are not in self.
    - **Bonus:** do not use the all\_paths auxiliary function. do it smart (:
  + find\_shortest\_path(self, frm\_name, to\_name) – returns the path from frm\_name to to\_name which has the minimum total weight.
    - This method should return None if there is no path between frm\_name and to\_name.
    - **Bonus:** do not use the all\_paths auxiliary function. do it smart (:
      * Note: path finding is usually implemented with recursion. We didn’t learn recursion in our course, so I recommend implementing a non-recursive algorithm like: “breadth-first search” or “depth-first search”.

## Task 2 – Exemplary usage

### Question 1

Make some tests to make sure your implementation works.

### Question 2

Create 3 Graph objects, each contains a different collection of nodes, which together contain all 10 nodes. Use the \_\_add()\_\_ method to create a total graph that contains the entire data of the example.

### Question 3

Sort the nodes by the number of their reachable nodes.

### Question 4

What is the pair of nodes that the shortest path between them has the highest weight?

## Task 3 – The roadmap implementation (BONUS)

The files travelsEW.csv and travelsWE.csv record a large number of travels made by people from five regions in the country, called Center, North, South, East and West.

### Question 1

From each file create a graph whose nodes are the country regions, and whose edges are the roads themselves (if a travel was not recorded between country regions, then it means such road does not exist). The weight of each edge is defined as the average time (in seconds) of all the travels done on that road. When the two graphs are ready, add them together to create the complete graph of the roadmap.

### Question 2

From which region to which region it takes the longest time to travel?

# Part III – Non-directional graph (BONUS)

## Task 1 – define the class

Implement the NonDirectionalGraph class as a sub-class of Graph. The main property of the non-directional graph is that its edges come in pairs, so if an edge is added or removed, the class must make sure the same applies to its counterpart. Make sure you rewrite all relevant methods.

## Task 2 – The social network implementation

The file social.txt describes chronologically the intrigues among 14 friends. Use the data in the file and the classes you’ve defined to answer the following questions.

### Question 1

What was the highest number of simultaneous friendships?

### Question 2

What was the maximum number of friends Reuben had simultaneously?

### Question 3

At the current graph (considering all the data of the file), what is the maximal path between nodes in the graph?

### Question 4

Implement a function called suggest\_friend(graph, node\_name) that returns the name of the node with the highest number of common friends with node\_name, which is not already one of his friends.

Good luck!

# Submission

* the classes definitions should be store in a python file, which will be used as module, named ‘graph\_project.py’
  + the ‘graph\_project.py’ will contain only the classes themself, without any usage (instance creation).
  + see empty example (skeleton) in the project folder
* the usage of this module should be done in a different file, preferably in a jupyter notebook.
  + see a simple notebook example in the project folder, named ‘project\_notebook\_example.ipynb’
* note that the testing of the project is done using unit tests (automatic tests) and not by code-review style tests - meaning I will probably not look at your code.
  + This means that your primary objective is making the code works, regardless if its efficient or not (or if its the best way to solve it).
  + However, notwithstanding, I will probably look at a few random projects deeper.
* you should send the two files (the module file and the usage file) as a reply to the email I’ll send about the project