Contents

1 Basic Test Results

```
Starting tests...
Wed Dec 30 16:32:40 IST 2015
91aded68c354559306aece0b9a2d4be2cbc626b8  -


Archive:  /tmp/bodek.m4tG4o/intro2cs/ex10/elinorperl/presubmission/submission
  inflating: src/ex10.py
  inflating: src/README

Testing README...
Done testing README...

Running presubmit tests...
18 passed tests out of 18
result_code    jaccard    18    1
52 passed tests out of 52
result_code    friends    52    1
18 passed tests out of 18
result_code    path    18    1
13 passed tests out of 13
result_code    article    13    1
11 passed tests out of 11
result_code    pagerank    11    1
4 passed tests out of 4
result_code    readfile    4    1
29 passed tests out of 29
result_code    network    29    1
Done running presubmit tests

Tests completed

Additional notes:

There will be additional tests which will not be published in advance.
```

2 README

```
elinorperl
329577464
Elinor Perl

I discussed the exercise with Talya Adams, Eli Corn, Nophar Sarel, and
 Ben Jacobi

=========================================
=  README for ex10: WikiNetwork  =
=========================================


==================
=  Description:  =
==================

In this exercise, I defined two classes "Article",
and "WikiNetwork", each playing intergral part in building a network of
articles. Articles was defined according to the attributes enabling me to
 build the Wikinetwork.

======================
=  Special Comments  =
======================

I used stackoverflow.com

Questions asked in the exercise:

1. 1) 'United_States'
   2) 'France'
   3) 'Europe'

2. 1) 'United_States' - 'Driving_on_the_left_or_right'
   2) 'Israel' - 'Yemen'
   3) 'United_Kingdom' - 'Scotland'
   4) 'Algebra' - 'Calculus'
   5) 'World_War_II' - 'Adolf_Hitler'

3. Percentage friends of distance 1 from "Christopher_Columbus" -
 0.9172308364271675
   Percentage friends of distance 2 from "DNA" - 13.758462546407513
   Percentage friends of distance 3 from "History" - 61.694693164446385
```

3 ex10.py

```python
import copy
import math
from operator import itemgetter


def read_article_links(file_name):
    """
    This function opens the file we'd like to access and arranges the articles
    in pairs of tuples, seperated by tab, each tuple seperated by a line,
    creating a list of articles in this format.
    """
    articles = []
    f = open(file_name, 'r')
    file = f.read().split('\n')
    for line in file:
        new_articles = line.split('\t')
        articles.append((tuple(new_articles)))
    del articles[-1]
    return articles


class Article:
    """
    Article is an object defined inside a network with characteristics based on
     the network's needs - Along with it's name, neighbors (articles that
     directly relate to the intial article - object) play an integral part
     in this class.
    """
    def __init__(self, name):
        """
        A constructor for our object - article, includes names and neighbor
        list and it's money us later with Wikinetwork
        """
        self.__name = name
        self.collection = []
        self.starting_money = 1.
        self.updated_money = 0
        self.entry_degree = 0


    def get_name(self):
        """
        Get function - calls name from __init__, enables accessibility to other
        classes
        """
        return self.__name

    def add_neighbor(self, neighbor):
        """
        Adds the neighbors to our collecion.
        """
        if neighbor not in self.collection:
            self.collection.append(neighbor)
            neighbor.update_entry_degree()

    def get_neighbors(self):
        """
        Get function - calls neighbors from __init__, enables accessibility to
        other classes
        """
        return self.collection

    def get_entry_degree(self):
        """
        Returns the entry degree of neighbors to an article
        """
        return self.entry_degree

    def update_entry_degree(self):
        """
        Updates the entry degree of each neighbor by adding a degree.
        """
        self.entry_degree += 1

    def article_entry(self):
        """
        Compares the degree values of each neighbor in the collection list
        returning the high degree, if there are multiple neighbors with the
        same entry degree, the function arranges it in alphabetical order.
        """
        highest_degree = self.collection[0]
        for neighbor in self.collection[1:]:
            if highest_degree.get_entry_degree() < neighbor.get_entry_degree():
                highest_degree = neighbor
            if highest_degree.get_entry_degree() == neighbor.get_entry_degree():
                if highest_degree.get_name() > neighbor.get_name():
                    highest_degree = neighbor
        return highest_degree


    def get_starting_money(self):
        """
         Get function - returns the starting money.
        """
        return self.starting_money

    def get_updated_money(self):
        """
        Get function - returns the starting money.
        """
        return self.updated_money

    def set_starting_money(self, distribute):
        """
        Updates the starting money, according to the input
        """
        self.starting_money = distribute

    def set_updated_money (self, new_money):
        """
        Updates the money each time by adding the input money.
        """
        self.updated_money += new_money

    def __repr__(self):
        """
        Returns name and neighbors in a tuple.
        """
        neighbors = []
        for neighbor in self.collection:
            neighbors.append(neighbor.get_name())
        article_decription = self.__name, neighbors
        return str(article_decription)

    def __len__(self):
        """
        returns the length of our neighbors
```

```python
        """
        return len(self.collection)

    def __contains__(self, article):
        """
        returns a Boolean value, if an article can be found in our neighbor
        collection.
        """
        if article in self.collection:
            return True
        else:
            return False

class WikiNetwork:
    """
    WikiNetwork operates the network of articles built based on articles
    that were built in the former class.
    """

    def __init__(self, linked_list=[]):
        """
        The Wikinetwork constructor - gets a list of articles, and builds
        a dictionary from it and updates it's network according to the linked
        lists items
        """
        self.article_dic = {}
        self.update_network(linked_list)

    def update_network(self, linked_list=[]):
        """
        This function updates the dictionary as long as the article doesn't
        already appear in it, and afterwards, updates the articles neighbors.
        """

        for article1, article2 in linked_list:
            if article1 not in self.article_dic:
                self.article_dic[article1] = Article(article1)
            if article2 not in self.article_dic:
                self.article_dic[article2] = Article(article2)
            self.article_dic[article1].add_neighbor(self.article_dic[article2])

    def get_articles(self):
        """
        Returns a list of the articles from our dictionary.
        """
        articles = []
        for value in self.article_dic.values():
            articles.append(value)
        return articles

    def get_titles(self):
        """
        Returns a list of the name of our articles from the dictionary.
        """
        return [name for name in self.article_dic.keys()]

    def __contains__(self, article_name):
        """
        Boolean function to check if the article name can be found in our
        dictionary.
        """
        if article_name in self.article_dic.keys():
            return True
        else:
            return False

    def __len__(self):
        """
        Returns the length of article list.
        """
        return len(self.get_articles())

    def __repr__(self):
        """
        Returns the dictionary in a string.
        """
        return str(self.article_dic)

    def __getitem__(self, article_name):
        """
        Checks it article_name is found in the dictionary and returns its
        object if it is, otherwise rasing a key error and acting as python
        would to a problem.
        """
        if article_name in self.article_dic.keys():
            return self.article_dic[article_name]
        else:
            raise KeyError(article_name)

    def sorted_list(self, list):
        """
        Sorts the list given by value and leaving only the key, taking into
        account if there are multiple items of the same value - it will
        sort them alphabetically.
        """
        return [key for key, value in sorted(list,
                                             key=lambda x: (-(x[1]),x[0]))]

    def page_rank(self, iters, d=0.9):
        """
        Page_rank repeats the same process for the amount of iters that were
        input. It updates the "money" for each article and neighbor according
        the given equation and resets it once all the appropriate actions
        were take to acquire its page rank value and moves on the the next
        iterator process. Afterward creating a list of the article name and
        value in tuples, and sorting it.
        """
        page_rank_list = []
        for i in range(iters):
            for article, value in self.article_dic.items():
                distribution = (value.get_starting_money()*d) / len(value)
                for neighbor in value.get_neighbors():
                    neighbor.set_updated_money(distribution)
            for article, value in self.article_dic.items():
                value.set_starting_money(value.get_updated_money()+(1-d))
                value.set_updated_money(0)
        for keys, values in self.article_dic.items():
            page_rank_list.append((keys, values.get_starting_money()))
        return self.sorted_list(page_rank_list)

    def jaccard_index_code(self, A, B):
        """
        Using the to sets that were input, and as long as the denominator
        meets the domain (not zero), applies the sets to the jaccard index
        equation.
        """
        if len(A.union(B)) != 0:
            return abs(len(A.intersection(B))) / abs(float(len(A.union(B))))
        else:
            return 0

    def jaccard_index(self, article_name):
        """
        Jaccard index checks if the input article is found in our dictionary,
        if so proceeds to go through the items in our dictionary. Each value
        being the jaccard code, and afterwards added it to a new list which
        is sorted by values and if there are multiple values with the same
        jaccard index, alphabetically.
        """
        jaccard_dic = {}
        jaccard_list = []
        if article_name in self.article_dic.keys():
            if len(self[article_name]) > 0:
                article = self[article_name]
                for key, value in self.article_dic.items():
                        jaccard_dic[key] = self.jaccard_index_code \
                            (set(article.get_neighbors()),
                             set(value.get_neighbors()))
                for keys, values in jaccard_dic.items():
                    jaccard_list.append((keys, values))
                return self.sorted_list(jaccard_list)
        return None


    def travel_path_iterator(self, article_name):
        """
        If article name appears in our dictionary, using the generator
        yields the first article and moves on to its neighbors as long as the
        article has neighbors and the neighbor hasn't been visited yet, and
        calls to the function of the best article entry from class article,
        yielding the highest ranking article in its path of incoming
        neighbor (stopping when it has reaching a neighbor with no incoming
        neighbors).
        """
        if article_name in self.article_dic:
            article = self.article_dic[article_name]
            yield article_name
            visited_list = []
            while len(article.get_neighbors()) > 0 and article_name not in \
                    visited_list:
                visited_list.append(article_name)
                best = article.article_entry()
                yield best.get_name()
                article = best
                article_name = article.get_name()
            raise StopIteration
        else:
            return []

    def friends_depth_helper(self, friends_depths, depth, counter):
        """
        This is a helper recursion function to get to the neighbor depth.
        starting with our condition, once the counter reaches the depth
        amount, it will return the list, recursively repeating the function
        adding the the counter and friend_depths each time.
        """
        if counter == depth:
            return friends_depths
        else:
            for friend in friends_depths:
                friends_depths = friends_depths | \
                                    set(friend.get_neighbors())
            return friends_depths | self.friends_depth_helper \
                (friends_depths, depth, counter + 1)


    def friends_by_depth(self, article_name, depth):
        """
        This function makes friends_depth list into a set, therefore not
        repeating any element twice, calling onto the helper function with
        the friends_depth set, the starting depth and starting our counter at
        0, creating a new list with the names of the articles.
        """
        article_list = []
        if article_name in self.article_dic:
            article_object = self.article_dic[article_name]
            friends_depth = set()
            friends_depth.add(article_object)
            helper = self.friends_depth_helper(friends_depth, depth, 0)
            for article in helper:
                article_list.append(article.get_name())
            return article_list
        else:
            return None
```