CS5001
Spring 2019
HW4 (lists, loops, strings)
Assigned: February 7, 2019
Deadline: February 14, 2019 at 9:00am <===== **2/14 6:00pm due to SNOWPOCALYPSE**

Admin notes (also on piazza...):

- Tuesday 2/12 office hours cancelled after 4:30pm
- Laney's adding office hours Thursday 9-11am, WVH 310

To submit your solution, compress all files together into one .zip file. Login to `handins.ccs.neu.edu` with your CCIS account, click on the appropriate assignment, and upload that single zip file. You may submit multiple times right up until the deadline; we will grade only the most recent submission.

Recent handouts that might be useful: strings, lists, and nested lists.

Your solution will be evaluated according to our CS5001 grading rubric. The written component accounts for 20% of your overall HW4 score; the programming component for 80%. We'll give written feedback when appropriate as well; please come to Laney's office hours with any questions about grading (email to set up a time if no current office hours work for you).

If you apply your late token, your new deadline is **February 18, 2019 at 9:00am**. Email laneys@northeastern.edu to notify us you'll be cashing it in.

Written Component

- Filename: `written.txt`

Written #1
Each of the lists below is defined in "roster notation" -- listing out every single element. Rewrite each line below to create the list using Python list comprehension.

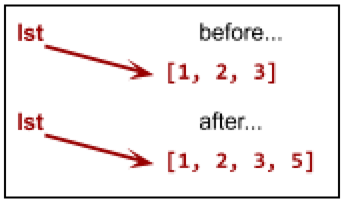
1A `lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

1B `lst = [2, 4, 6, 8, 10, 12, 14, 16, 18]`

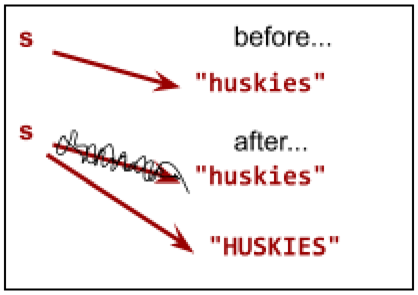
1C `lst = [-1, -2, -3, -4, -5, -6, -7, -8, -9]`

Written #2
Because lists are mutable, they can be modified without using the assignment operator. We can directly manipulate their values in memory, without reassigning that label to another value.

For example: **`lst.append(5)`** puts the value 5 on the back of a list called **`lst`**. Here's what memory looks like:



Strings are immutable, and so the assignment operator needs to be involved in order to modify a string. For example: **`s = s.supper()`** reassigns the uppercase version of a string to the variable **`s`**. Here's what memory looks like:



This difference impacts the way function parameters can be modified within their functions:

- If the parameter is modified directly (like the list example), then the caller's variable is modified as well.
- If the assignment operator gets involved (like the string example) then the caller's variable is unaffected.

With that in mind, what does the following Python program print to the terminal?

```python
def modify_list(lst):
    lst.remove('a')

def modify_string(st):
    st = st.replace('a', 'w')

def modify_int(intgr):
    intgr = intgr / 10

def main():

    my_list = ['a', 'b', 'c']
    modify_list(my_list)
    print(my_list)

    my_string = 'ab'
    modify_string(my_string)
    print(my_string)

    intgr = 20
    modify_int(intgr)
    print(intgr)

main()
```

Written #3

For each of the Python snippets below, what will be printed to the terminal? Be specific with spacing and special characters like punctuation; we want to know exactly what the user will see.

3A

```python
colors = ["red", "green", "blue"]
print(colors)
```

3B

```python
colors = ["red", "green", "blue"]
for c in colors:
   print(c)
```

3C

```python
colors = ["red", "green", "blue"]
for i in range(3):
   print(i, colors[i])
```

3D

```python
colors = ["red", "green", "blue"]
for i in range(3):
      print(i, colors[i], sep=", ")
```

Written #4

For each of the Python snippets below, what will be printed to the terminal?

4A

```python
for x in range(6):
    print(x)
```

4B

```python
for x in range(2, 6):
    print(x)
```

4C

```python
for x in range(2, 30, 5):
    print(x)
```

Programming Component
Make sure you review the Python Style Guide. A percentage of your score for every homework is for writing good, clear, readable code. As with all your CS5001 assignments, user interaction must be friendly and informative.

Programming #1

- Filename: `upc.py`
- `test_upc.py` -- for reference, a program we'll use to test your code as part of your correctness score.

Our friend the modulo operator can be useful in surprising ways -- like making sure we don't accidentally overpay at a checkout register. Every item you purchase has a Universal Product Code (UPC), which you'll see on a sticker or otherwise attached to it, along with a barcode used for scanning. Like this:

Most of these digits are chosen to specify the manufacturer, product type, country, etc., but one of them is set aside to be the "check digit." After all the other numbers are set, the check digit is chosen such that a calculation we apply to the numbers will end up being a multiple of 10.

Why are UPCs designed this way? Because then, if you're typing one up on a cash register or in a database for an online store, if you make a mistake... it's super easy to tell. (Same thing happens with credit cards. If you type in your credit card number in a website, you get notified right away that it's invalid without anyone having to contact your bank. Try it sometime, for fun! You get that feedback right away, without checking your credit limit or verifying your name/expiration date or anything.)

For UPC numbers, the validation algorithm proceeds on the number **from right to left:**

- Digits in even positions, including zero: no change
- Digits in odd positions: *3
- Sum the results

If it's a valid UPC number, this result is a multiple of 10.

The UPC number above is 0 7 3 8 5 4 0 0 8 0 8 9.

We apply the algorithm (the number above is written from left to right, but the algorithm goes right to left, so we say 9 is at position 0, 8 is at position 1, etc.):

$$\begin{aligned} & 9 + 3 \cdot 8 + 0 + 3 \cdot 8 + 0 + 3 \cdot 0 + 4 + 3 \cdot 5 + 8 + 3 \cdot 3 + 7 + 3 \cdot 0 \\ = \quad & 9 + 24 + 24 + 4 + 15 + 8 + 9 + 7 \\ = \quad & 100 \end{aligned}$$

And we're done! We've verified that this, in fact, a valid UPC number.

For this assignment, write a Python function that determines whether a given list of integers represents a valid UPC. Use a loop to apply the algorithm described here. UPC codes can be different lengths, so don't assume it'll always be exactly 12 digits.

Special case: If the input is less than 2 digits long, or if all the digits have value 0, the code is **not valid.**

Your function must meet the following specifications:

- Name: **`is_valid_upc`**
- Input: List of integers, a possible UPC number
- Returns: Boolean, indicating whether the given input is valid or not

Testing Your Program
For this part of the assignment, your functions must match the spec exactly -- same name, parameters, and return type. This is so we can test your code with testing code we've already written, which will contribute to the correctness part of your grade.

AMAZING points (in addition to have well-written code and meeting the specs of the assignment):

- Your function must have a maximum of 30 lines. Break it up into smaller pieces if necessary, but the initial function called from the test suite must still match the specs.
- Deal with the special cases first and the common case second, clearly spelled out and separated by vertical space in your function.

Programming #2

- Filename (we're providing the driver; don't submit it): `mbta_directions.py`
- Filename (for your functions): `mbta.py`
- Filename (for your test suite): `mbta_test.py`

For this assignment, you'll be the function writer. We've provided the driver on the course website, `mbta_directions.py`, and you'll need to define (and test) functions that match our function calls. Don't modify the provided driver! We'll test your solution using the original driver we provided. Note that the name of your module and the names of your functions must match exactly for our driver to run.

Functions (defined in mbta.py)

We're doing a simplified version of getting directions on the T -- red line only, Ashmont branch only. As you can see in the driver, the user provides a start and end station, and we tell them:

- Which direction to go (Ashmont or Alewife).
- Number of stops to get there (a positive number).
- Station name must match exactly (unforgiving about upper/lowercase and abbreviations).

Our driver needs the following functions in your mbta module (again, names are important and need to match exactly for our driver to work):

1. `is_valid_station`
    - Input: station name, a string.
    - Returns: boolean, True if station is in the red line, False otherwise.
2. `get_direction`
    - Input: start and end stations, both strings
    - Returns: first stop or last stop on the red line.
        i. If either station doesn't exist on the red line, return the string `"no destination found"`.
        ii. If both stations are the same, return `"no destination found"`.
3. `get_num_stops`
    - Input: start and end stations, both strings
    - Returns: number of stops from start to end, a positive integer. If either station doesn't exist on the red line, or if both stations are the same, return 0.

The Red Line

The list of valid stations should live in your file mbta.py.

```
RED_LINE = ["Ashmont", "Shawmut", "Fields Corner", "Savin Hill", "JFK/UMass",
            "Andrew", "Broadway", "South Station", "Downtown Crossing",
            "Park St", "Charles/MGH", "Kendall", "Central", "Harvard",
            "Porter", "Davis", "Alewife"]
```

The test suite

Your test suite `mbta_test.py` should have a main function (`mbta.py` should not).

Your test suite should test all 3 functions, each with enough variety of inputs that you're confident all possible scenarios will work. Of course, the driver we've provided is the end result for this assignment, but you might put these functions out there for anyone to use. Convince yourself that they all work, no matter how the driver ends up using your functions.

When I run your test suite, I should see the following printed on the terminal:

- The details of running each individual test on each of the three functions:
    - The input to the function,
    - the expected output, and
    - the outcome of the test (SUCCESS or FAILURE)
- The total number of tests that succeeded, and the total number that failed

AMAZING points

- Forget the part about station names matching exactly. Accept upper/lowercase station names, no matter how the user chooses to capitalize (e.g., "Ashmont" is the same as "ashmont" is the same as "ASHMONT" is the same as "ashmoNT".)

Programming #3

- Filename (we're providing; don't submit it): **`kmeans_viz.py`**
- Filename (for your driver): **`kmeans_driver.py`**

For this assignment, you'll be writing the driver for a program. We've provided functions that you'll call within your driver to make a visual representation of your solution. Don't modify the functions! We'll test your solution by running it with the original file we provided.

Data mining is a field within computer science where we attempt to make sense out of a whole bunch of input. It's often described as transforming data into information.

You'll implement a data mining algorithm called *k-means clustering*, a process by which data is organized into a small number (*k*) of clusters of similar values. Your algorithm will process a set of 2-dimensional points, assigning each of them to a cluster. We've provided on the course website in **`kmeans_viz.py`**, with functions you'll call to visually represent your clusters.

We're doing only the first step of what would be a full-on clustering program, but it still works OK. Here's what your program needs to do:

- Set the number of clusters (i.e., the value of *k* in *k-means clustering*). We'll use 4.
- Choose 4 random points to be your "centroids." Our centroids will be chosen from our initial data. Choose 4 points at random from the input data set.
- For each point in the data set, find the closest centroid. Assign that point to the centroid. Once all the points are assigned to a centroid, we have our clusters!
- Draw the centroids, and the clusters, using the functions defined in **`kmeans_viz.py.`**

What's the closest centroid?
The closest centroid for a given point is chosen using the Euclidean distance algorithm, but extended to be more general than the one we saw in HW2. A given point can have an arbitrary number of coordinates, not just *x* and *y*. The generalized Euclidean formula is:

$$d(a,\ b) = \sqrt{\sum_j (a_j - b_j)^2}$$

For example, the Euclidean distance between (-1, 2, 3) and (4, 0, -3) is

$$\sqrt{(4-(-1))^2 + (0-2)^2 + ((-3)-3)^2}$$

$$= \sqrt{5^2 + (-2)^2 + (-6)^2}$$

$$= \sqrt{25+4+36}$$

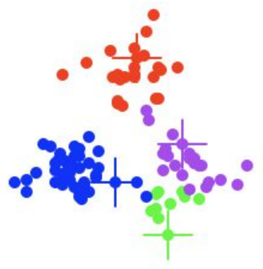
$$= \sqrt{65}$$

=8.0622

In case of a tie (i.e., two or more centroids are equidistant from a given point), choose any one of them.

Visualizing the Data
Invoke the functions we've defined in **`kmeans_viz.py`**. You need to invoke only two functions: **`draw_centroids`** and **`draw_assignment`**.

Because the centroids are randomly chosen, your output will look different every time. But in general, we draw the centroids as plusses and points as circles, so the clusters are easy to see; here's an example:



The way the data is organized is important for the visualization functions that you'll invoke. Each function, of course, has its name, parameters, return type, and description in a comment, but we're recapping here parameters that are commonly used in the visualization module.

- **`data`**, a list of lists, each containing exactly 2 floats to represent a 2-dimensional point. For this program, *data* will contain 100 points. (This should be the same as input data, below.)
- **`centroids`**, a list of lists, each containing exactly 2 floats to represent a 2-dimensional point. For this program, *centroids* will contain 4 points, because we're making 4 clusters.
- **`assignment`**, a list of lists, each containing exactly 2 integers to represent the indices of *data* and *centroids*, respectively. For example, if assignment contains [[8, 0], [9, 3]], then we would say that `data[8]` is assigned to `centroid[0]`, and `data[9]` is assigned to `centroid[3]`.
- **`colors`**, a list of strings that is the same length as **`centroids`**.

Read the function comments to be sure you're using them correctly and your inputs are formatted the way the functions need. The functions will not change, and you cannot add your own to **`kmeans_viz.py.`**

Input Data
No input comes from the user. Your initial dataset is this list of lists, containing 100 2-dimensional points (copy-paste this into your code):

```
DATA = [ [-32.97, -21.06], [9.01, -31.63], [-20.35, 28.73], [-0.18, 26.73],
         [-25.05, -9.56], [-0.13, 23.83], [19.88, -18.32], [17.49, -14.09],
         [17.85, 27.17], [-30.94, -8.85], [4.81, 42.22], [-4.59, 11.18],
         [9.96, -35.64], [24.72, -11.39], [14.44, -43.31], [-10.49, 33.55],
         [4.24, 31.54], [-27.12, -17.34], [25.24, -12.61], [20.26, -4.7],
         [-16.4, -19.22], [-15.31, -7.65], [-26.61,    -20.31], [15.22,
-30.33],
         [-29.3, -12.42], [-50.24, -21.18], [-32.67, -13.11], [-30.47, -17.6],
         [-23.25, -6.72], [23.08, -9.34], [-25.44, -6.09], [-37.91, -4.55],
         [0.14, 34.76], [7.93, 49.21], [-6.76, 12.14], [-19.13, -2.24],
         [12.65, -7.23], [11.25, 25.98], [-9.03, 22.77], [9.29, -26.2],
         [15.83, -1.45], [-22.98, -27.37], [-25.12, -23.35], [21.12, -26.68],
         [20.39, -24.66], [26.69, -28.45], [-45.42, -25.22], [-8.37, -21.09],
         [11.52, -16.15], [7.43, -32.89], [-31.94, -11.86], [14.48, -10.08],
         [0.63, -20.52], [9.86, 13.79], [-28.87, -17.15], [-29.67, -22.44],
         [-20.94, -22.59], [11.85, -9.23], [30.86, -21.06], [-3.8, 22.54],
         [-5.84, 21.71], [-7.01, 23.65], [22.5, -11.17], [-25.71, -14.13],
         [-32.62, -15.93], [-7.27, 12.77], [26.57, -13.77], [9.94, 26.95],
         [-22.45, -23.18], [-34.7, -5.62], [29.53, -22.88], [0.7, 31.02],
         [-22.52, -10.02], [-23.36, -14.54], [-19.44, -12.94], [-0.5, 23.36],
         [-45.27, -19.8], [8.95, 13.63], [47.16, -14.46], [5.57, 4.85],
         [-19.03, -25.41], [28.16, -13.86], [-15.42, -14.68], [10.19, -25.08],
         [0.44, 23.65], [-20.71, -20.94], [35.91, -20.07], [42.81, -21.88],
         [5.1, 9.33], [-15.8, -18.47], [5.39, -26.82], [-40.53, -17.16],
         [-29.54, 23.72], [7.8, 23.4], [-22.19, -27.76], [-23.48, -25.01],
         [-21.2, -21.74], [23.14, -24.14], [-28.13, -13.04], [-24.38, -6.79] ]
```

Driver
When I run your driver, I shouldn't see any output other than the Turtle graphics that plot your centroids and data points. I should see a slightly-different one each time, because the centroids are randomly chosen.

Each centroid must have a unique color, and all of its corresponding data points should be drawn with that color. In our example, we used red, green, blue, and purple, but you can choose any four turtle-friendly colors you like.

No test suite is required, because you are the function caller in this scenario. You can define functions within your driver, and you should make sure they work, though!

AMAZING points

- Instead of stopping after assigning points to randomly-chosen centroids as described here, do one more step:
    - After all points have been assigned to a centroid, re-compute the centroid of each cluster. This should be the point x_a, y_a where x_a is the average x-value of all the points in the cluster and y_a is the average y-value of all the points in the cluster.
    - Now, the centroid for each cluster is an actual centrally-located point, instead of something randomly chosen.
    - Do one more iteration of assigning data points to the new centroids -- you should get better clusters this way!