HW1: Introduction to Python

Due Sep 17 by 11am **Points** 100 **Submitting** a file upload
Available Sep 10 at 11am - Sep 18 at 11am 8 days

This assignment was locked Sep 18 at 11am.

Program Goals

- Get familiar with Python syntax
- Get familiar with a Python development/execution environment
- Implement a basic AI algorithm

Be advised:

If you need a tutorial for Python programming, we have released Python and Numpy tutorials on Piazza (https://piazza.com/class/kef2n72f455sp?cid=100) . Please check it out.

Summary

This project includes two separate implementations:

1. State space generation for the water jug puzzle (please write the code in a file called **p1_statespace.py**)
2. Weather prediction from historical data (please write the code in a file called **p1_weather.py**)

We'll walk you through the concepts and provide quite a bit of scaffolding for each of the implementations since you will be expected to complete them both in Python, which many of you do not know or haven't used in a while. **Note that you're responsible for the output format (e.g., the data type, String or Boolean) of each implemented function. Please strictly follow the instructions below.**

IMPORTANT: Please make sure your functions do NOT have any extra debug prints. If you have any testing code, it should be contained within a main() method so it does not start to execute when the file is imported by the grading framework. This will lead to a penalty in your assignment scores.

Part 1: State Space Generation

The code for this portion should be written in a file named **p1_statespace.py**. You will only define the 4 Python functions in your file, following the exact same function signature and having the correct output format as discussed below.

This is a problem of the two water jugs (denoted by 0 and 1) with fixed integer size. We'll use this as an exercise to learn using Python lists.

For the purposes of this assignment, we will be representing each state as a **list of two ints**. Please also maintain a separate list of two ints with the respective maximum capacities of the jugs.

There are three operations you can perform on a jug:

- Fill the jug to capacity from the faucet
- Empty the jug onto the ground
- Pour into another jug until that jug is full or this jug is empty

For this component of implementation, please implement **four (4)** Python functions:

1. **fill(state, max, which)** — returns a copy of `state` which fills the jug corresponding to the index in `which` (0 or 1) to its maximum capacity. Do not modify `state`.
2. **empty(state, max, which)** — returns a copy of `state` which empties the jug corresponding to the index in `which` (0 or 1). Do not modify `state`.
3. **xfer(state, max, source, dest)** — returns a copy of `state` which pours the contents of the jug at index `source` into the jug at index `dest`, until `source` is empty or `dest` is full. Do not modify `state`. xfer is shorthand for transfer.
4. **succ(state, max)** — prints the list of **unique** successor states of the current state in any order. This function will generate the unique successor states of the current state by applying fill, empty, xfer operations on the current state. (Note: You have to apply an operation at every step for generating a successor state.)

Each of these functions should take as an argument the current state of the two water jugs as a list of two integers, as well as the maximum capacities of each jug as a list of two integers.

These functions should work as follows (though be sure to test them more thoroughly yourself). For the demo examples below, we have printed the output from the functions. But in the actual implementation, you have to follow the exact output format which the function expects (either return the state/print all the states) as mentioned above.

```
>>> max = [5,7]
>>> s0 = [0,0]
>>> fill(s0,max,1)
[0,7]
>>> s0
[0,0]
>>> fill(s0,max,0)
[5,0]
>>> s1 = fill(s0,max,1)
>>> xfer(s1,max,1,0)
[5,2]
>>> succ(s0,max)
[0,0]
```

```
[5,0]
[0,7]

>>> max = [5,7]
>>> s0 = [3,1]
>>> succ(s0,max)
[5,1]
[3,7]
[0,1]
[3,0]
[4,0]
[0,4]
```

You may assume our states and capacities will be valid, non-negative integers presented in a valid, two-element list, and indexes will be valid. Error-check if you want the practice, but it will not be part of your score.

Be sure to test your functions with capacities other than 5 and 7! We will test those cases :)

In the event that a function does not modify the state (e.g. you fill a jug which is already at capacity or empty a jug which is already empty), you should still return a list whose contents are identical to the original state.

```
>>> s0 = [0,0]
>>> max = [3,7]
>>> empty(s0,max,1)
[0,0]
>>> s0 == empty(s0,max,1)
True
```

Part 2: Weather Prediction

The code for this portion should be written in a file named **p1_weather.py**.

This implementation will require some basic math, file I/O, and use of some built-in Python data structures, and is a little more involved than the previous one.

You'll be using a version of the **k-Nearest Neighbors algorithm (https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm)** to predict whether we expect it to be raining in Seattle based on various weather conditions.

For this component of implementation, please implement **four (4)** Python functions:

1. **manhattan_distance(data_point1, data_point2)** – return the Manhattan distance between two dictionary data points from the data set.
2. **read_dataset(filename)** – return a list of data point dictionaries read from the specified file.
3. **majority_vote(nearest_neighbors)** – return a prediction of whether it is raining or not based on a majority vote of the list of neighbors.

4. **k_nearest_neighbors(filename, test_point, k, year_interval)** – using the above functions, return the majority vote prediction for whether it's raining or not on the provided test point.

1. Manhattan distance

The Manhattan distance function in 2d is defined as:

$$d = |x_1 - x_2| + |y_1 - y_2|$$

However, in our case, we're taking the distance between points in three-dimensional space, where those dimensions are the precipitation amount (PRCP), maximum temperature (TMAX), and minimum temperature for the day (TMIN). You can ignore the date when doing this calculation.

```
>>> manhattan_distance({'DATE': '1951-05-19', 'TMAX': 66.0, 'PRCP': 0.0, 'TMIN': 43.0, 'RAIN': 'FALSE'},{'DAT
E': '1951-01-27', 'TMAX': 33.0, 'PRCP': 0.0, 'TMIN': 19.0, 'RAIN': 'FALSE'})
=> 57.0
>>> manhattan_distance({'DATE': '2015-08-12', 'TMAX': 83.0, 'PRCP': 0.3, 'TMIN': 62.0, 'RAIN': 'TRUE'}, {'DAT
E': '2014-05-19', 'TMAX': 70.0, 'PRCP': 0.0, 'TMIN': 50.0, 'RAIN': 'FALSE'})
=> 25.3
```

2. Read dataset

Okay, so why do the data points in the previous example code look like that? Because you'll be reading them in from **rain.txt**.

Each line of the file looks something like this:

```
1948-01-01 0.47 51 42 TRUE
```

where the first entry is the DATE, the second entry is the PRCP (precipitation), the third entry is the TMAX (maximum temperature), the fourth entry is the TMIN (minimum temperature) and the last entry is a boolean representing RAIN (what we'll be predicting for new data points).

In this function, you must read the file in line by line, split each line by its spaces, and create a dictionary **with the keys listed above** and the values on the line, where the numeric values have been converted to floats. The function should return a list with one dictionary for each line in the file.

The sample line provided above, for example, should produce the following dictionary:

```
{'DATE': '1948-01-01', 'TMAX': 51.0, 'PRCP': 0.47, 'TMIN': 42.0, 'RAIN': 'TRUE'}
```

We won't show you the whole result of testing this function, but here are some things you can try:

```
>>> dataset = read_dataset('rain.txt')
>>> len(dataset)
=> 25548
```

```
>>> dataset[0]
=> {'DATE': '1948-01-01', 'TMAX': 51.0, 'PRCP': 0.47, 'TMIN': 42.0, 'RAIN': 'TRUE'}
```

3. Majority vote

Very simply, a k-Nearest Neighbor (or kNN) algorithm looks at the classes assigned to the k points closest to the query point, and says that the query point has the same label as most of its neighbors (in this case, whether it was raining or not). This function takes in a list of those neighbors and returns the **string** representing whether it's raining or not. (Note: this function does not return a boolean! Booleans in Python look like `True` or `False`. We want all-caps.)

If a tie occurs, default to 'TRUE' as your answer. This is Seattle, after all.

```
>>> majority_vote([{'DATE': '2015-08-12', 'TMAX': 83.0, 'PRCP': 0.3, 'TMIN': 62.0, 'RAIN': 'TRUE'},
{'DATE': '2014-05-19', 'TMAX': 70.0, 'PRCP': 0.0, 'TMIN': 50.0, 'RAIN': 'FALSE'},
{'DATE': '2014-12-05', 'TMAX': 55.0, 'PRCP': 0.12, 'TMIN': 44.0, 'RAIN': 'TRUE'},
{'DATE': '1954-09-08', 'TMAX': 71.0, 'PRCP': 0.02, 'TMIN': 55.0, 'RAIN': 'TRUE'},
{'DATE': '2014-08-27', 'TMAX': 84.0, 'PRCP': 0.0, 'TMIN': 61.0, 'RAIN': 'FALSE'}])
=> 'TRUE'
```

4. k-Nearest Neighbors

In this function, you'll be given the path to a correctly-formatted file (to read using your function), a test point dictionary that contains all of the same keys (except the label 'RAIN'), the value of k indicating how many neighbors to select and the value of year interval that will be used to determine the valid neighbors.

Based on the given year interval, filter out the invalid data points that are interval or more years away from the input test point in terms of the year values. For example, the data points of the year 1965 and the year 1985 are both invalid for the test point of the year 1975 if the input year interval is 10.

By using your Manhattan distance, find the **closest k valid neighbors**. If the number of the valid neighbors is smaller than the input value k, just keep as many valid neighbors as possible. If there isn't any valid neighbor, default to 'TRUE' as your answer. Finally, return their majority vote on whether it's raining or not in the test point.

Note that the following example is just for the purpose of input/output format illustration. The output of this example isn't necessarily correct.

```
>>> k_nearest_neighbors('rain.txt', {'DATE': '1948-01-01', 'TMAX': 51.0, 'PRCP': 0.47, 'TMIN': 42.0}, 2, 10)
=> 'TRUE'
>>> k_nearest_neighbors('rain.txt', {'DATE': '1948-01-01', 'TMAX': 51.0, 'PRCP': 0.47, 'TMIN': 42.0}, 2, 10)
=> 'FALSE'
```

Submission

Please submit both **p1_statespace.py** and **p1_weather.py** at the same time so the TAs can grade your homework easily. You may submit more than once; unless you specify otherwise, we will grade your last submission.

HW1

Criteria	Ratings		Pts
State Space: fill() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
State Space: empty() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
State Space: xfer() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
State Space: succ() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
State Space: general formatting, commenting, or file name issues	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Weather: manhattan_distance() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Weather: read_dataset() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Weather: majority_vote() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Weather: k_nearest_neighbors() works as specified	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Weather: general formatting, commenting, or file name issues	**10.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
			Total Points: 100.0