CSCA08H Assignment 2: Maintaining Bridges Infrastructure

Introduction

The Government of Ontario collects a huge amount of data on provincial programs and infrastructure, and much of it is provided as open data sets for public use. In this assignment, we'll work with a particular dataset that contains information about provincially owned and maintained bridges in Ontario. All bridges in Ontario are reviewed every 2 years, and their information is collected to help determine when bridges need inspection and maintenance. In this assignment, you will write several functions to help explore and process real data about the bridges in Ontario, and simulate how inspectors get assigned to various bridges.

The data you'll be working with is provided by the Ontario government and contains information about all bridges in the Ontario highway network, such as the length of the bridges, their condition over various years, and historical information. To simplify your task, we are only giving you part of the provided data. We also "cleaned up" the data for you: made sure it is in a consistent format.

The purpose of this assignment is to give you practice using the programming concepts that you have seen in the course so far, including (but not limited to) strings, lists and list methods, and loops.

This handout explains the problem you are to solve, and the tasks you need to complete for the assignment. Please read it carefully.

Goals of this Assignment

- Continue to use the Function Design Recipe to plan, implement, and test functions.
- Design and write function bodies using lists and list methods, and loops. (You can do this whole assignment with only the concepts from Weeks 1 through 6 of CSCA08.)
- Practice good programming style.

Files to Download

Please download the Assignment 2 Data Files and extract the zip archive. A description of each of the files that we have provided is given in the paragraphs below:

There are two starter code Python files, a sample data file, and a Python program that helps you check (**not fully test!**) your solutions for Python style violations:

- `constants.py`

  This file contains some useful constants that you should use in your solution. You will **not** modify this file.

- `bridge_functions.py`

  This file contains some code that sets up your use of constants from `constants.py`, some helper functions for you to use (see the table of functions to implement below), and **headers and docstrings** (but not bodies) for several functions you are to write, to help you get started. Your job is to complete this file.

- `bridge_data.csv`

  The `bridge_data.csv` file contains example bridge data in comma-separated values (CSV) format. You must not modify this file.

- `a2_checker.py`

  This is a checker program that you should use to check your code. See below for more information about `a2_checker.py`. The checker program does not test the correctness of your functions, so you must do that yourself.

The Data

For this assignment, you will use data from a Comma Separated Value (CSV) file named `bridge_data.csv`. Each row of this file contains the following information about a single bridge:

- bridge label: a label representing the bridge
- bridge name: the name of the bridge
- highway name: the name of the highway (not necessarily unique)
- latitude: the latitude of the centre of the bridge
- longitude: the longitude of the centre of the bridge
- year built: the year the bridge was built
- last major rehab: the year the last major rehabilitation was performed on the bridge
- last minor rehab: the year the last minor rehabilitation was performed on the bridge
- number of spans: the number of spans of the bridge
- span details: the detail of each span.

    - This is in the format:

      ```
      Total=[total length of all spans] (1)=[the length of the first span];(2)=[the length of the second span];
      ```

      and so on for each span of the bridge.
- length: the length of the bridge (this can differ from the total length of all spans!)
- last inspection date: the date that the bridge was last inspected (in MM/DD/YYYY format)
- last inspected index: the **bridge condition index (BCI)** from the last inspection
    - A bridge condition index (BCI) is a numeric score representing the condition of a bridge. The lower the BCI, the worse the condition of the bridge.
- bridge condition indices (BCIs): several numeric score representing the condition of the bridge over time

We have provided a function named `read_data`, which reads a CSV file and returns its contents as a `list[list[str]]`. As you develop your program, you can use the `read_data` function to produce a larger data set for testing your code. See the main block at the end of `bridge_functions.py` for an example.

Formatting the data (function `format_data`)

Here is a sample `list[list[str]]` returned by the provided function `read_data` (we added spacing here for readability that you won't see in Python):

```
[
   ['1 - 32/', 'Highway 24 Underpass at Highway 403', '403', '43.167233', '-80.275567', '1965', '2014', '2009', '4', 'Total=64
   (1)=12;(2)=19;(3)=21;(4)=12;', '65', '04/13/2012', '72.3', '', '72.3', '', '69.5', '', '70', '', '70.3', '', '70.5', '', '70.7', '72.9',
   ''],

   ['1 - 43/', 'WEST STREET UNDERPASS', '403', '43.164531', '-80.251582', '1963', '2014', '2007', '4', 'Total=60.4 (1)=12.2;
   (2)=18;(3)=18;(4)=12.2;', '61', '04/13/2012', '71.5', '', '71.5', '', '68.1', '', '69', '', '69.4', '', '69.4', '', '70.3', '73.3', ''],

   ['2 - 4/', 'STOKES RIVER BRIDGE', '6', '45.036739', '-81.33579', '1958', '2013', '', '1', 'Total=16 (1)=16;', '18.4',
   '08/28/2013', '85.1', '85.1', '', '67.8', '', '67.4', '', '69.2', '70', '70.5', '', '75.1', '', '90.1', '']
]
```

Notice that all of the data in the inner lists are represented as strings. You are to write the function `format_data`, which should make modifications to the list such that it follows this format:

Index	Data	Type	Description
`ID_INDEX`	ID	`int`	The ID of the bridge. The first bridge should have an ID of 1, the second should have an ID of 2, and so on. Your program must generate the ID numbers. Note: the original bridge labels are discarded.
`NAME_INDEX`	Name	`str`	The name of the bridge.
`HIGHWAY_INDEX`	Highway	`str`	The name of the bridge's highway.
`LAT_INDEX`	Latitude	`float`	The latitude of the bridge.
`LON_INDEX`	Longitude	`float`	The longitude of the bridge.
`YEAR_INDEX`	Year Built	`str`	The year the bridge was built. If no year is provided, this will be an empty string.
`LAST_MAJOR_INDEX`	Last Major Rehab	`str`	The year of the last major rehab. If no year is provided, this will be an empty string.
`LAST_MINOR_INDEX`	Last Minor Rehab	`str`	The year of the last minor rehab. If no year is provided, this will be an empty string.
`NUM_SPANS_INDEX`	Number of Spans	`int`	The number of spans on the bridge. A bridge will always have at least 1 span.
`SPAN_DETAILS_INDEX`	Span Details	`list[float]`	The length of each of the spans of the bridge. Example: If the Span Details in the unformatted data was: `Total=64 (1)=12;(2)=19;(3)=21;(4)=12;` And the bridge had 4 spans, then the formatted version should be the following list:

Index	Data	Type	Description
			`[12.0, 19.0, 21.0, 12.0]`
`LENGTH_INDEX`	Bridge Length	`float`	The length of the bridge. If no length is provided, this will be `0.0`.
`LAST_INSPECTED_INDEX`	Last Inspected Date	`str`	The date of the last inspection (in MM/DD/YYYY format). If no date is provided, this will be an empty string.
`BCIS_INDEX`	BCIs	`list[float]`	A list of the BCIs from the most recent to the least. If the bridge has no BCIs, then this should be an empty list.

Bridge data: format of data to be returned by `format_data` for A2.

If the index constants above have values `0`, `1`, `...`, `12`, then after applying the `format_data` function to the example list, the list should look like:

```
THREE_BRIDGES = [

  [1, 'Highway 24 Underpass at Highway 403', '403', 43.167233, -80.275567, '1965', '2014', '2009', 4, [12.0, 19.0, 21.0, 12.0],
  65.0, '04/13/2012', [72.3, 69.5, 70.0, 70.3, 70.5, 70.7, 72.9]],
  [2, 'WEST STREET UNDERPASS', '403', 43.164531, -80.251582, '1963', '2014', '2007', 4, [12.2, 18.0, 18.0, 12.2], 61.0,
  '04/13/2012', [71.5, 68.1, 69.0, 69.4, 69.4, 70.3, 73.3]],
  [3, 'STOKES RIVER BRIDGE', '6', 45.036739, -81.33579, '1958', '2013', '', 1, [16.0], 18.4, '08/28/2013', [85.1, 67.8, 67.4,
  69.2, 70.0, 70.5, 75.1, 90.1]]]

]
```

Before you write the `format_data` function body, please note:

1. To help you structure your code and define helper functions for the `format_data` function, we included several headers and docstrings in the starter code. Please, make sure to **study the starter code carefully** before writing any code for this function.
2. This function is one of the more challenging functions in A2, so we suggest that you don't start with it.

Your Tasks

Your functions will fall into three categories: functions for data formatting, functions for data queries, and functions for data modification. The functions for data queries and modification will all work with data formatted as described above.

Function name: (Parameter types) -> Return type	Full Description (paraphrase to get a proper docstring description)
`get_bridge: (list[list], int) -> list`	The first parameter represents the bridge data, and the second parameter represents the id of a bridge. The function should return the data of the bridge with the given id. If there is no bridge with the given id, the function should return the empty list. For example, when called with the example `THREE_BRIDGES` as the first argument and `1` as the second argument, the function should return: `[1, 'Highway 24 Underpass at Highway 403', '403', 43.167233, -80.275567, '1965', '2014', '2009', 4, [12.0, 19.0, 21.0, 12.0], 65, '04/13/2012', [72.3, 69.5, 70.0, 70.3, 70.5, 70.7, 72.9]]`
`get_average_bci: (list[list], int) -> float`	The first parameter represents the bridge data, and the second parameter represents the id of a bridge. The function should return the average BCI of the bridge with the given id. If there is no bridge with the given id, the function should return zero. If there are no BCIs for the bridge with the given id, the function should return zero. For example, when called with the example `THREE_BRIDGES` as the first argument and `1` as the second argument, the function should return approximately `70.8857`.
`get_total_length_on_hwy: (list[list], str) -> float`	The first parameter represents the bridge data, and the second parameter represents the highway. The function should return the total length of bridges on that highway. If there are no bridges on the highway, the function should return zero. For example, when called with the example `THREE_BRIDGES` as the first argument and `'403'` as the second argument, the function should return `126.0`.
`get_distance_between: (list, list) -> float`	The two parameters represent two bridges. The function should return the distance in kilometres, rounded to the nearest metre (i.e., 3 decimal places), between them. For example, if the first argument is the bridge from `THREE_BRIDGES` with id 1, i.e., the bridge `[1, 'Highway 24 Underpass at Highway 403', '403', 43.167233, -80.275567, '1965', '2014', '2009', 4, [12.0,`

	`19.0, 21.0, 12.0], 65.0, '04/13/2012', [72.3, 69.5, 70.0, 70.3, 70.5, 70.7, 72.9]]`, and the second argument is the bridge from `THREE_BRIDGES` with id 2, i.e. the bridge `[2, 'WEST STREET UNDERPASS', '403', 43.164531, -80.251582, '1963', '2014', '2007', 4, [12.2, 18.0, 18.0, 12.2], 61.0, '04/13/2012', [71.5, 68.1, 69.0, 69.4, 69.4, 70.3, 73.3]]`, then the function should return `1.968`. *Hint*: use a provided helper function `calculate_distance` to calculate the distance.
`get_closest_bridge: (list[list], int) -> int`	The first parameter represents the bridge data, and the second parameter represents the id of a bridge. The function should return the id of a bridge that has the shortest distance to the bridge with the given id. (The function should not return the bridge with the given id itself, i.e., we do not consider a bridge to be closest to itself!) You may assume that the bridge with the given id is present in the bridge data, and that there are at least two bridge in the bridge data. For example, when called with the example `THREE_BRIDGES` as the first argument and `2` as the second argument, the function should return `1`. *Hint*: use the function `get_distance_between`.
`get_bridges_in_radius: (list[list], float, float, float) -> list[int]`	The first parameter represents the bridge data, the second and third parameters represent the location (latitude and longitude), and the fourth parameter represents a radius/distance. The function should return a list of ids of all bridges that are within the given distance of the given location. For example, when called with the example `THREE_BRIDGES`, `43.10`, `-80.15`, and `50` as arguments, the function should return the list `[1, 2]`.
`get_bridges_with_bci_below: (list[list], list[int], float) -> list[int]`	The first parameter represents the bridge data, the second parameter represents a list of bridge ids, and the third parameter represents a BCI. The function should return a list of ids of all bridges whose ids are in the given list of ids and whose BCI is less than or equal to the given BCI. For example, when called with the example `THREE_BRIDGES`, the list of ids `[1, 2]`, and the BCI limit `72`, the function should return the list `[2]`.
`get_bridges_containing: (list[list], str) -> list[int]`	The first parameter represents the bridge data, and the second parameter represents a search string. The function should return a list of ids of all bridges whose names contain the search string. The search should be case-insensitive. For example, when called with the example `THREE_BRIDGES` and the search string `'underpass'`, the function should return the list `[1, 2]`. Note that when called with the example `THREE_BRIDGES` and the search string `'pass'`, the function should also return the list `[1, 2]`, i.e. the function looks for part of the name, not necessarily the entire name.
`inspect_bridges: (list[list], list[int], str, float) -> None`	The first parameter represents the bridge data, the second parameter represents a list of bridge IDs, the third parameter represents a date, and the fourth parameter represents a BCI. The function should update the bridges with the given IDs with the new given date and the new given BCI score for a new inspection. If there is no corresponding bridge for one or more of the given bridge ids in the given data, the function has no effect for that bridge id. For example, if the function is called with the example `THREE_BRIDGES`, a list of one ID `[1]`, date `'09/15/2023'`, and BCI `71.9`, then after the call the bridge data `THREE_BRIDGES` should contain all the same data, except the first bridge record should become: `[1, 'Highway 24 Underpass at Highway 403', '403', 43.167233, -80.275567, '1965', '2014', '2009', 4, [12.0, 19.0, 21.0, 12.0], 65, '09/15/2023', [71.9, 72.3, 69.5, 70.0, 70.3, 70.5, 70.7, 72.9]]`
`add_rehab: (list[list], int, str, bool) -> None`	The first parameter represents the bridge data, the second parameter represents a bridge ID, the third parameter represents a date, and the fourth parameter represents whether the rehab is major (argument is `True`) or minor (argument is `False`). The function should update the bridge with the given ID with the new rehab year record: year of major rehab, if the last argument is `True`, and year of minor rehab if the last argument is `False`. If there is no bridge with the given id in the given bridge data, the function has no effect. For example, if the function is called with the example `THREE_BRIDGES`, the ID `1`, date `'09/15/2023'`, and `False`, then after the call the bridge data `THREE_BRIDGES` should contain all the same data, except the first bridge record should become: `[1, 'Highway 24 Underpass at Highway 403', '403', 43.167233, -80.275567, '1965', '2014', '2023', 4, [12.0, 19.0, 21.0, 12.0], 65, '04/13/2012', [72.3, 69.5, 70.0, 70.3, 70.5, 70.7, 72.9]],`
`format_data: (list[list[str]]) -> None`	See Formatting the data.
`assign_inspectors: (list[list], list[list[float]], int) -> list[list[int]]`	The first parameter represents the bridge data, the second parameter represents a list of locations of inspectors (pairs of latitude and longitude), and the third parameter represents the maximum number of bridges that should be assigned to any inspector. The function should return a list of bridge IDs to be assigned to each inspector. See Assigning inspectors for the rules and the algorithm.

	For example, when called with the example `THREE_BRIDGES`, the list of inspector locations `[[43.20, -80.35], [45.0368, -81.34]]`, and the bridge limit of `2`, the function should return the list `[[1, 2], [3]]`.

Functions to write for A2

Assigning Inspectors (function `assign_inspectors`)

Bridge condition indices (BCI) represent the condition of a bridge on a past inspection, and we use the most recent BCI to prioritise certain bridges. There are three levels of priorities, with their upper limits represented by the constants `HIGH_PRIORITY_BCI`, `MEDIUM_PRIORITY_BCI`, `LOW_PRIORITY_BCI`, constrained by `HIGH_PRIORITY_BCI < MEDIUM_PRIORITY_BCI < LOW_PRIORITY_BCI`.

For example, if `HIGH_PRIORITY_BCI` is 60, then all bridges with their most recent BCI less than or equal to 60 are considered 'high priority'. Similarly, if `MEDIUM_PRIORITY_BCI` is 70, then all bridges with a BCI less than or equal to 70 (but > 60) are considered 'medium priority'.

When assigning bridges to inspectors, we want to prioritise nearby high priority bridges within a large radius of the inspector over medium priority bridges that are closer, and both of those are prioritised over low priority bridges in an even smaller radius.

The radii are specified by the constants `HIGH_PRIORITY_RADIUS`, `MEDIUM_PRIORITY_RADIUS`, and `LOW_PRIORITY_RADIUS`.

You are told the maximum number of bridges to assign per inspector. The way we want to assign bridges to inspectors is as follows:

1. High priority bridges with a distance <= `HIGH_PRIORITY_RADIUS` from the inspector.
2. If (1) assigned fewer than the maximum number of bridges, then we go on to assign medium priority bridges with a distance <= `MEDIUM_PRIORITY_RADIUS` from the inspector.
3. If (1) and (2) still assigned fewer than the given maximum number of bridges, then we go on to assign low priority bridges with a distance <= `LOW_PRIORITY_RADIUS` from the inspector.

One inspector at a time, we assign the maximum number of bridges to each inspector (or fewer than the maximum number of bridges, if all bridges have already been assigned). Inspectors should be assigned bridges based on the order they appear in the list (e.g., the first inspector in the list should be assigned up to the maximum number of bridges first, the second inspector should be assigned up to the maximum number of bridges next, and so on).

Bridges are assigned to an inspector using as many bridges that fulfil (1) as possible, followed by (2), and then (3) if there are still fewer than the maximum number of bridges assigned to that inspector. If there are multiple bridges with the same priority and radius, we choose the bridge with the lowest ID (e.g., if there are two low priority bridges with IDs 3 and 4, we would assign the bridge with ID 3 first). Each bridge should be assigned to at most one inspector.

Hint: You may want to use the `get_bridges_in_radius` and `get_bridges_with_bci_below` functions to help you with assigning inspectors.

A2 Checker

We are providing a checker module (`a2_checker.py`) that tests two things:

- whether your code follows the Python style guidelines, and
- whether your functions are named correctly, have the correct number of parameters, and return the correct types.

To run the checker, open `a2_checker.py` and run it. Note: the checker file should be in the **same** directory as your `bridge_functions.py`, as provided in the starter code zip file. Be sure to scroll up to the top and read all messages!

If the checker passes for both style and types:

- Your code follows the style guidelines.
- Your function names, number of parameters, and return types match the assignment specification. **This does not mean that your code works correctly in all situations.** We will run a *different* set of tests on your code once you hand it in, so be sure to thoroughly test your code yourself before submitting.

If the checker fails, carefully read the message provided:

- It may have failed because your code did not follow the style guidelines. Review the error description(s) and fix the code style. Please see the PyTA documentation for more information about errors.
- It may have failed because:
  - you are missing one or more functions,
  - one or more of your functions are misnamed,
  - one or more of your functions have incorrect number or types of parameters, or
  - one of more of your functions return values of types that do not match the assignment specification.

  Read the error messages to identify the problematic functions, review the function specifications in the handout, and fix your code.

Make sure the checker passes before submitting.

Testing your Code

It is strongly recommended that you test each function as you write it. (You might be tempted to just plow ahead and write all of the functions and hope everything works, but then it will be really difficult to determine whether your program is working correctly.)

As usual, follow the Function Design Recipe. Once you've implemented a function, run it on the examples in the docstring. Here are a few tips:

- Be careful that you test the right thing. Some functions return values; others modify data in-place. Be clear on what the functions are doing before determining whether your tests work.
- Can you think of any special cases for your functions? Will each function always work, or are there special cases to consider? Test each function carefully.
- Once you are happy with the behaviour of a function, move to the next function, implement it, and test it.

Remember to run the checker frequently as you work!

Marking

These are the aspects of your work that may be marked for A2:

- **Coding style (20%):**
  - Make sure that you follow Python style guidelines that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we do not provide an exhaustive list of style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with one exception: *docstrings will be evaluated separately*. For each occurrence of a PyTA error, one mark (out of 20) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.
  - All functions, including helper functions, should have complete docstrings including preconditions when you think they are necessary and at least two valid examples.
- **Correctness (80%):**

  Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.

  Note that for full marks **all docstring examples you provide should run without producing errors**.

What to Hand In

The very last thing you do before submitting should be to run the checker program one last time.

Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit `bridge_functions.py` on MarkUs by following the instructions on the course website. Remember that spelling of filenames, including case, counts: your file must be named exactly as above.