Assignment 0

Canadian Weather Data

Due date: Friday, February 5, 2021 before 1:00 pm sharp, Toronto time.

You must complete this assignment individually. Partners are permitted only on Assignments 1 and 2.

Learning goals

This assignment is called “Assignment 0” because it is considerably smaller than Assignments 1 and 2. Its purpose is to make sure that you have understood the basics of object-oriented programming in Python and are ready to move on with the more advanced topics in the course.

By the end of this assignment you should be able to:

- Implement a class in Python from a provided interface, including:
  - Using instance attributes that enable the class methods to provide their services to client code
  - Enforcing representation invariants that record important facts about implementation decisions
  - Implementing initializers and other methods
  - Working with instance attributes that involve composition of objects
- Choose unit tests that can convincingly demonstrate that a method works according to its docstring for all valid inputs
- Implement those test cases in pytest
- Interpret the results of doctests and unit tests to assess the correctness of methods

You should also have developed these habits:

- Implementing tests for a method before implementing the method
- Running your test suite throughout the development of your code

General coding guidelines

These guidelines are designed to help you write well-designed code that will adhere to the interfaces we have defined (and thus will be able to pass our test cases). Overall, you mustn’t change the function and class interfaces that we provided in the starter code.
In particular, do not:

- change the parameters, parameter type annotations, or return types in any of the methods or functions you have been given in the starter code.
- add or remove any parameters in any of the methods you have been given in the starter code.

- change the type annotations of any public or private attributes you have been given in the starter code.
- create any new public attributes.
- create any new public methods.
- write a method or function that mutates an object if the docstring doesn't say that it will be mutated.
- add any more import statements to your code, except for imports from the `typing` module.

On the other hand, you are welcome to create new *private* helper methods for the classes you have been given, since this doesn't change the interface. If you create new private methods, you must provide type annotations for every parameter and return value. You must also write a full docstring for each method as described in the **function design recipe (https://www.teach.cs.toronto.edu/~csc148h/winter/notes/python-recap/function_design_recipe.pdf)** .

While writing your code you may assume that all arguments passed to the methods and functions you have been given in the starter code will respect the preconditions and type annotations outlined in the docstrings provided.

Introduction

The internet is full of interesting data that you can explore. In this assignment, you will be developing Python code that can read in historical weather data from files, store that data in appropriate data structures, and then compute some meaningful statistics based on the data.

The program

The code consists of three Python classes:

- An instance of `DailyWeather` is a record of weather facts for a single day (temperature and precipitation).
- An instance of `HistoricalWeather` is a record of weather information on various dates, for a fixed place on Earth.
- An instance of `Country` is a record of weather information on various dates, for various locations in a country.

There are also two top-level functions, that is, functions that are defined outside of any class.

The interface to the functions and to the classes have been designed, and we have decided how data will be represented, but most of the method bodies need to be written.

Input to the program

The input to the program is real weather data, from a Government of Canada website: `https://climate.weather.gc.ca/historical_data/search_historic_data_e.html`.

The data stored is in csv (comma-separated values) files. In a csv file, each line contains values separated by commas. Open the example weather files in PyCharm, or a text editor of your choice, to see the format. If you open these files instead in excel, they will be displayed as a spreadsheet. But remember what's really in the file! For instance, there may be double quotes around the values in the csv files. You'll need to strip these off to get at the content you are interested in.

The data files record, among other things, rainfall, snowfall, and precipitation. You might imagine that precipitation is the sum of rainfall plus snowfall, but the relationship is more complicated than that (and in fact doesn't matter to this assignment).

Near the top of `weather.py`, there is a list of CONSTANTS corresponding to what is stored in each column of a weather data file. You must use these constants in your program rather than "hard coded" values. For instance, the maximum temperature is always stored in column 9, but to extract it from a row you must use the constant `MAX_TEMP` rather than the number 19. (Why do you think this is important?)

Here are some things your program can assume will be true for any data file it is asked to read:

- A weather data file will always contain a header row, which specifies what data each column in the file corresponds to.
- Every data file will always have the same columns in the same order as you see in our sample data files.
- The header row will be followed by zero or more rows of weather data.
- The weather data will be in order from oldest dates to most recent dates.

Missing or ill-formed data

Part of working with real data is dealing with incomplete or ill-formed information. In this assignment, if a row in a weather data file is missing one or more of these values:

- Longitude, Latitude
- Station Name
- Year, Month, Day
- Max Temp, Min Temp, Mean Temp
- Total Rain, Total Snow, Total Precip

then that entire row should **not** be included in the historical weather data. Similarly, if any piece of data is of the wrong sort, then the entire row should not be included. For instance, if we find "boo!" in the Month column, we should ignore that row.

The columns "Total Rain Flag", "Total Snow Flag", and "Total Precip Flag" should either be empty or contain the value "T". ("T" indicates that there were "trace amounts".) If you encounter any value aside from these, it should be considered ill-formed and the entire row should not be included.

Your tasks

Save a0.zip (https://q.utoronto.ca/courses/204422/files/11998319?wrap=1) (https://q.utoronto.ca/courses/204422/files/11998319/download?download_frd=1) to your computer and extract its contents. Copy all the extracted files and folders into your `csc148/assignments/a0` folder. `a0.zip` includes:

- `weather.py`: Starter code that you will complete and **submit**.
- `a0_starter_tests.py`: Some basic tests cases that you should add to in order to test your own code.
- `sample_data`: A folder containing some small example data files.
- For reference: A document describing the class design recipe that we use in this course, and the code example that it uses.

Your tasks are to:

1. Implement the methods defined in class `DailyWeather`
2. Implement the methods defined in class `HistoricalWeather`
3. Implement the methods defined in class `Country`
4. Finally, complete the code that will allow your program to read data from files. We have written function `load_country`, but you need to complete it's helper function, `load_data`. Don't forget the note above about missing or ill-formed data.

Each method or function that you need to write is marked with a comment saying "TODO". When you write the method or function, remove that comment.

There are many ways we could have chosen to represent the data, and you may find it interesting to consider alternatives. But you must use the attributes as we defined them, and mustn't add any new attributes.

Because these classes use composition, a method in one class will call methods in another. But it must not call private methods or access private attributes. Respecting privacy is part of good code design.

Be sure that you follow all of the other coding guidelines listed above.

If a docstring does not specify what to do in a particular scenario, you may decide what the method will do (we won't test that scenario). You are welcome, and encouraged, to add private helper methods as you see fit. However, **you must not add any public methods**.

Tips and suggestions

- The method docstrings are written to be as precise as possible, and that sometimes makes them challenging to read. It will help a great deal if you make a small example of data that the method might have to handle, and then use that to figure out what the method is supposed to do. "Be the

method" yourself by enacting it on that data before you try to write code that will do it. You can just write the data as a small table on a piece of paper, and include only the relevant columns.

- In method `contiguous_precipitation`, you will likely find `timedelta` to be helpful. Here is an example of its use:

```
>>> d1 = date(2021, 5, 12)
>>> d2 = date(2021, 5, 23)
>>> # Is d2 11 days after d1?  Yes!
>>> d1 + timedelta(11) == d2
True
```

- In method `load_data`, every value you pull out of a line of data could be empty, or it could be inappropriate (for example, 'boo!' where you are expecting an integer). If you try to turn either kind of input into an `int`, you will get an error. It would be tedious to use an if-statement to check every single time. You can use a Python feature called `try-except` to make this much easier. Below is an example, where we are trying to sum up the values in a list of integers. This code can handle the problematic list (currently commented out) without any difficulty. It adds as many numbers as it can, and bails out gracefully if it encounters something it can't convert to an integer.

```
    # stuff = ['31', '8', '', 'junk!!', '52']
    stuff = ['5', '31', '94', '52']
    total = 0
    # Try to do the code inside here.  Don't try to stop it from raising
    # a ValueError. If it does, jump immediately to the "except" clause.
    try:
        for item in stuff:
            total += int(item)
    except ValueError:
        # This is where we end up if a ValueError is raised.
        pass
    # Whether or not we end up in the "except" clause, the program continues
    # here.
    print(total)
```

About testing

We have provided several things to help you test your code:

- the doctests in the starter code,
- unit tests, written using pytest, in `a0_starter_tests.py`, and
- a slightly larger set of tests that you can run via MarkUs.

However, we have further hidden tests that we will use to assess your code. Your assignment grade will be based on the autotesting that we do, and *you can be sure we'll try to break your code as hard as we can*, so you should also! To test your own code thoroughly, add more tests to `a0_starter_tests.py`.

The most efficient way to produce code that works is to create and run your test cases as early as possible, and to re-run them after every change to your code. A very disciplined approach is to design and implement your unit tests for a method before you write the method. (This is called "test-

driven development".) You can do this one method at a time, getting each one working before moving on to the next. Don't forget that each method has doctests that you can run – even if you have written the code before writing its unit tests.

Note: We will not directly test any helper methods that you define. (Think about why this wouldn't be possible.)

Generating reports

In the main block, we have included two examples of code that uses the classes you are working on. One of the examples calls your methods and produces a historical weather report in file `report.md` (md is for "markdown", which is a notation for describing formatting very concisely). Once you have completed all of your tasks above, you will be able to run the weather module and produce this file. If you open it in PyCharm, PyCharm will understand the markdown and show you something like this:

Location	record high for Dec 25	
YORK	13.6	
THUNDER BAY	1.9	

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the Python style guidelines that PyCharm points out. Fix them!
- In each module, run the provided `python_ta.check_all()` code to check for errors. Fix them! PythonTA is also included in the tests we provided on MarkUs. The full feedback will be in a file added to your submission called pyta_feedback.txt.
- Check your docstrings for any helper methods you wrote to make sure they are precise and complete, and that they follow the conventions of the Function Design Recipe and the Class Design Recipe.
- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as calls to the `print` function.
- Remove any `pass` statement where you have added the necessary code.
- Remove the word "TODO" wherever/whenever you have completed the task.

- Take pride in your gorgeous code!

Submission instructions

Submit your file `weather.py` on MarkUs. No other files need to be submitted. We strongly recommend that you submit early and often. We will grade the latest version you submit within the permitted submission period.

Be sure to run the tests we've provided within MarkUs one last time before the due date. This will make sure that you didn't accidentally submit the wrong version of your code, or worse yet, the starter code!