COMP 202

Winter 2022

Assignment 4

Due: Tuesday, April 12th, 11:59 p.m.

Please read the entire PDF before starting. You must do this assignment individually.

Part 1:	20 points
Part 2:	40 points
Part 3:	40 points
Bonus:	8 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, allowing the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while grading, then that increases the chance of them giving out partial marks. :)

To get full marks, you must follow all directions below:

- Make sure that all file names and function names are **spelled exactly** as described in this document. Otherwise, a 50% penalty per question will be applied.
- Make sure that your code **runs without errors**. Code with errors will receive a very low mark.
- Write your name and student ID in a comment at the top of all `.py` files you hand in.
- Name your variables appropriately. The purpose of each variable should be obvious from the name.
- **Comment your code.** A comment every line is not needed, but there should be enough comments to fully understand your program.
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing.
- Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.
- **Up to 30% can be removed for bad indentation of your code, omission of comments, and/or poor coding style (as discussed in class).**

Hints & tips

- **Start early.** Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.

- Read these instructions and make sure you understand them thoroughly before you start. Ask questions if anything is unclear!
- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to your TA during office hours if you are having difficulties with programming. Go to an instructor's office hours if you need extra help with understanding a part of the course content.
  - At the same time, beware not to post anything on the discussion board that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved. If you cannot think of a way to ask your question without giving away part of your solution, then please drop by our office hours.
- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. And as you will discover for yourself, reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.
  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

Revisions

April 1:

- Fixed some typos in some function names.
- Fixed the example for `most_contacted_student` function.
- Deleted the extra spaces from `all_students.json`. Make sure to download it again.
- Added a clarification for dealing with empty lists in `build_report` method.
- Deleted the example of the `FileNotFoundError` from `load_file` function – do not consider the case when this error happens, because then there is no need to close the file as it hasn't been opened in the first place.
- No need to deep copy the class attribute `students` in the class `ContactTracker`.
- A disclaimer is added for `ContactTracker` class.
- Assume no cycles can happen for `min_distance_from_patient_zeros`. The description is updated.
- Default value for `students` attribute in the class `ContactTracker` has been added.
- A change in the instructions of exception handling in `load_file` and `write_in_file`.

Warm-up questions (0 points)

*Do **NOT** submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)

In the `bookstore.py` file created in the lecture videos, add a function called `create_bookstore`. The function takes as input a string indicating the name of the bookstore, and a list of strings each containing information about a book. The data related to a book is separated by tabs. The function should create a list of objects of type `Book` and then uses this list and the name to create and return a `Bookstore` object.

Warm-up Question 2 (0 points)

In the same module, add a function called `create_bookstore_from_file`. The function takes as input a filename containing information about all the books in this bookstore. Reading the file line by line, the function creates a list of objects of type `Book`. It then uses this list and the name of the file (without its extension) to create and return a bookstore. To test the function use the file `'City_Lights.txt'`.

Warm-up Question 3 (0 points)

In the `Bookstore` class, add a method called `plot_by_genre`. The method should create a bar plot of all the books in the bookstore by genre. Label the x axis with 'Genres', y axis with 'Number of Books', and title the graph with '<Name of the Bookstore> - Books by genre'. The plot should be saved in a file named '<Name of the Bookstore>_by_genre.png'. Test the method with the bookstore obtained by the previous function.

Assignment 4

The questions in this part of the assignment will be graded.

The main learning objectives for this assignment to apply all the concepts that we have learned during the semester:

- Work with (nested) lists and (nested) dictionaries using nested loops.
- Understand how to test functions that return dictionaries.
- Work with strings.
- Understand how to write a docstring and use doctest.
- Apply what you have learned about file IO.
- Apply what you learned about about exception handling, and `try`, `except`, `finally` blocks.
- Apply what you have learned about OOP – classes, methods, constructors, attributes, objects etc.

Note that this assignment is designed for you to be practicing what you have learned in the videos up to and including Lecture 40 (OOP 5: Class and static methods). For this reason, you are NOT allowed to use anything seen after Lecture 40 or not seen in class at all. You will be heavily penalized if you do so.

For full marks, in addition to the points listed on page 1, make sure to add the appropriate documentation string (docstring) to *all* the functions you write. The docstring must contain the following:

- The type contract of the function.
- A description of what the function is expected to do.
- At least three (3) examples of calls to the function. You are allowed to use *at most one* example per function from this PDF.

Examples

For each question, we provide several **examples** of how your code should behave. All examples are given as if you were to call the functions from the shell.

When you upload your code to codePost, some of these examples will be run automatically to test that your code outputs the same as given in the example. However, **it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples.** When the time comes to grade your assignment, we will run additional, private tests that will use inputs not seen in the examples. You should make sure that your functions work for all the different possible scenarios. Also, when testing your code, know that mindlessly plugging in various different inputs is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

Furthermore, please note that your code files **should not contain any function calls in the main body of the program** (i.e., outside of any functions). Code that contains function calls in the main body **will automatically fail the tests on codePost and thus be heavily penalized**. It is OK to place function calls in the main body of your code for testing purposes, but if you do so, make certain that you remove them before submitting. You can also test your functions by calling them from the shell. Please review what you have learned in Lecture 14 (Modules) if you'd like to add code to your modules which executes only when you run your files.

Safe Assumptions

For all questions on this assignment, you can safely assume that the **type** of the inputs (both to the functions and those provided to the program by the user) will always be correct. For example, if a function takes as input a string, you can assume that a string will always be provided. At times you will be required to do some input validation, but this requirement will always be clearly stated. Otherwise, your functions should work with any possible input that respect the function's description. For example, if the description says that the function takes as input a positive integer, then it should work with all integers greater than 0. If it mentions an integer, then it should work for *any* integer. Make sure to test your functions for edge cases!

Code Repetition

One of the main principles of software development is DRY: Don't Repeat Yourself. One of the main ways we can avoid repeating ourselves in code is by writing functions, then calling the functions when necessary, instead of repeating the code contained within them. Please pay careful attention in the questions of this assignment to not repeat yourself, and instead call previously-defined functions whenever appropriate. As always, you can also add your own helper functions if need be, with the intention of reducing code repetition as much as possible.

Comments: Comments in a function should be included **sparingly**. They should not be used to describe basic features of the Python language nor when the line itself can read as an English statement. For instance, when using a for loop to go over the lines of a file (`for line in f:`), a comment is not needed since reading the line out loud already indicates exactly what is going on (we are going through each line in the file).

A good rule of thumb is to assume that the reader of the comments is experienced with the Python language. So there is no need to write a comment about creating a variable, using a loop or opening a file. Instead, comments should be used when something logically complicated is going on. For instance, if there is a complicated conditional expression that involves some real-world idea (like the conditions in the `get_cheapest_trip_cost` function of Assignment 1), then it may be helpful to have a comment to explain the logic of what is going on (but even in this case, it may be better to include this logic in the docstring instead of having a comment).

We can also avoid writing comments by using descriptive and informative names for our variables; instead of `x`, we can write `rectangle_area`, and then there is no need to have a comment explaining the purpose of that variable.

Failure to adhere to these guidelines will incur style penalties.

COVID-19 contact tracker

Note: The story and the data provided is completely fictional and made up; it is designed to motivate the assignment, except unfortunately COVID itself. The student IDs and the names are randomly generated.

For the past two weeks there has been a significant surge of COVID-19 cases on McGill campus. As already experienced coders, McGill is asking for your help in protecting its community members, as well as slowing the spread of COVID-19 by designing a system for contact tracing in the student community. Are you up for doing some good for your community?

The analysis of the data provided will be done in 6 stages (the meanings of these stages might differ from application to application):

- Stage 0: Data collection – collecting data in a convenient data format (e.g. txt, xls, csv, pkl, etc.)
- Stage 1: Data input – data is loaded to the program
- Stage 2: Data preparation – data is organized and translated into objects that the program can work with
- Stage 3: Data processing – the data inputted to the computer in the previous stage is processed for interpretation. Processing is done using logical analysis like algorithms.
- Stage 4: Data output/interpretation – the processed data is translated in an easily readable form for non-programmers (e.g. charts, graphs, plain text).
- Stage 5: Data storage – data is stored/exported in a local file/server/cloud.

Luckily, McGill has already done Stage 0 and for the purposes of this assignment it is providing you with all the data you will need (all the data below is fictional):

- `all_students.json` – the list of all currently registered students (names and IDs) in JSON format.
- `cases.csv` – the list of sick students with the list of students that each sick student had contact with is CSV format.

We will explain these data formats a bit later below.

Stage 3 is responsible for doing logical analysis of the data and it will be handled by a file `contactTracker.py`.

All the remaining stages will be handled by `contactTracingSystem.py` which will be the main file, so it will put everything together.

Remark: You are not allowed to use global variables for this assignment.

`all_students.json` and the data format JSON

JSON (JavaScript Object Notation) is a format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs (dictionary) and lists. It's a common data format used in web applications. `all_students.json` is a list where each element of the list is a dictionary of two key value pairs – a key `"id"` and a value for the student ID, and a key `"name"` and a value for the student name. For example, `{"id": "260745567", "name": "Larry"}`. An example of `all_students.json` content is the following:

```
[
	{"id": "260745567", "name": "Larry"},
	{"id": "260761234", "name": "Zach"},
	{"id": "260785442", "name": "George"}
]
```

In this example, the JSON data is formatted, you can see that each dictionary starts in a new line and after one tabular (\t) symbol. You may assume the JSON file given to you will have the same format and spacing.

`cases.csv` and the data format CSV

The `cases.csv` is stored in CSV (comma separated value) files. Many operating systems, by default, open the files ending with the .csv extension in a spreadsheet program like Excel or Pages and when saving them will change their contents in a way that disrupts this assignment. Please don't open these files in such a program, and don't alter them in any way. You can open them in a text editor, like Notepad, TextEdit or Sublime, to see their contents.

Each line in `cases.csv` will be of the form *<Sick_Student_1>,<Contact_1>,<Contact_2>,...,<Contact_n>*, where *<Sick_Student_...>,<Contact_...>* are placeholders for specific student's student IDs.

An example what `cases.csv` can look like:

```
260745567, 260761234, 260245896
260761234, 260785442
```

Each line will always begin with exactly one person, followed by one or more other people – you can assume for convenience that each sick student had contact with at least one other student. Also, the sick student cannot appear in its own contact list. A comma and a space follow each ID except for the last one **We assume a student is sick if and only if their ID appears in first column of this file.**

In order to make the files easier to work with you can assume that the file does not have spaces or new lines before the first sick student ID and after the last contact ID. There will be one space after the comma, and there is no comma after the last contact ID in each line.

Part 1: Student class

Since the basic objects we are going to work with while performing the analysis are the students, it will be convenient to have a defined type of `Student`. Define a class Student in a `student.py` file. The Student object has three instance attributes named:

- `student_id` – string which is the McGill ID of the student.
- `name` – string which is the name of the student.
- `is_sick` – bool attribute, that is set to **`False`** by default. The attributes value can be used to help with the analysis later but it doesn't have to represent the reality at any given time, i.e. it doesn't have to be **`True`** when the student is in the sick students list. *You are not going to tested be for storing the value of this attribute right. You are only going to be tested for giving it a correct default value when a Student object is just created.*

Besides the definition of the class, include the following functions in the file `student.py`. In addition, you can also define your own helper functions. The methods below are instance methods unless otherwise stated.

- **`__init__`**: the constructor for the class, takes three inputs to initiate the attributes of the class – the student id (string), the student name (string) and a boolean variable indicating whether the student is sick. The latter has a default value **`False`**. Note the order of input parameters has to be the same as they are listed above. When setting the values for attributes the constructor also has to check whether the student is a valid McGill ID (by a function is_valid_student_id defined later). If the ID is not valid, for example 12a6b7O5s, then a **`ValueError`** has to be raised with a message `'The student ID 12a6b7O5s is not a valid ID'`.
- **`__str__`**: returns the following string for the object.

```
>>> larry = Student('260745567', 'Larry', True)
>>> str(larry)
'Larry (260745567)'

>>> alice = Student('260245896', 'Alice', False)
>>> str(alice)
'Alice (260245896)'
```

- **`__repr__`**: returns the following string for the object. It is the same string that **`__str__`** builds and returns, so you can just call function. **`__repr__`** overwrites the printable representation of the object (note the last command in the shell)

```
>>> larry = Student('260745567', 'Larry', True)
>>> repr(larry)
'Larry (260745567)'

>>> larry
Larry (260745567)
```

- **`is_valid_id`** is a static method: takes an input string for a student id and checks whether it is a valid McGill ID. An ID string is valid if it is a has 9 digits where the first three digits are 260.

```
>>> Student.is_valid_id('260745567')
True

>>> Student.is_valid_id('2601543s')
False
```

- `from_JSON` is a class method: constructs and return a Student object from the input parameter string that has the data of the student in JSON (in the format as it is given in `all_students.json` file).

```
>>> larry = Student.from_JSON('{"id": "260745567", "name": "Larry"}')
>>> str(larry)
'Larry (260745567)'

>>> alice = Student.from_JSON('{"id": "260245896", "name": "Alice"}')
>>> str(alice)
'Alice (260245896)'
```

Part 2: ContactTracker class

***Disclaimer**: In order to be able to test the methods in this class the data from the initial files should be loaded first, which means the functions from Stage 1 and 2 from Part 3 should be implemented first.*

This is the class that will be responsible for the logical analysis part. Define a class `ContactTracker` in the `contactTracker.py` file. It has two attributes:

- `students` – it is a class attribute, which is a list of Student objects. It should be set to represent the list of all students. The default value should be empty list.
- `cases_with_contacts` – it is an instance attribute. The attribute is a dictionary keeping the mapping between a sick student and its contacts. In particular, the keys in the dictionary are the IDs of sick students and for each sick student the value is the list of student IDs that the student has been in contact with. For example, for `cases.csv` the dictionary looks like:

```
{'260808934': ['260840155', '260248711', '260996175', '260476020', '260561504'],
'260996175': ['260248711', '260476020'],
'260675874': ['260840155'],
'260476020': ['260758421'],
'260386543': ['260996175', '260248711', '260476020', '260561504'],
'260248711': ['260212160', '260970944'],
'260840155': ['260970944'],
'260561504': ['260476020', '260248711'], '260970944': ['260212160']}
```

Besides the definition of the class, include the following functions in the file `contactTracker.py`. In addition, you can also define your own helper functions. All the functions that return a list, they can return in any ordering – the order of the elements inside the list is not going to be checked.

- `__init__`: the constructor for the class, takes two inputs to initiate the attributes of the class – a list of `Student` objects and a dictionary to initiate the instance attribute `cases_with_contacts` (the input dictionary should have the same structure as the attribute `cases_with_contacts`). The attribute `students` should be set only once for the entire class of `ContactTracker`, (as the list of registered students mostly doesn't change in a semester) – it should be the same list for all the instances of the class. Thus, in the constructor you should only set it if hasn't been set already by another instance of the class. The constructor should also check that all the student IDs appearing in `cases_with_contacts` can be found in the registered students list (stored in `students` attribute). If such a student ID cannot be found, for example `'260245896'`, then raise a `ValueError` with a message `'A student with id 260245896 either doesn't exist or is not reported as sick.'` Finally, the initializations of the attribute `cases_with_contacts` should be done by deep copying.

  Note: Although we require you to define two attributes for this class, you can define and use more attributes if that would make the logic of the next functions easier. You may initiate those attributes in the constructor, but the constructor should initialize at minimum the fixed parameters that we listed above.

The methods below are instance methods unless otherwise stated. They should be implemented by using in part the two attributes discussed above.

All the examples below assume that the same variable `contact_tracker` of type `ContactTracker` had been created from the data files `all_students.json` and `cases.csv` that are provided with this pdf. `contact_tracker` variable should be created outside the definition of `ContactTracker` class (for example it can be created in Part 3).

- `get_contacts_by_student_id`: takes a `student_id` of a sick student as an input and returns the list of `Student` objects that the sick student has been in contact with. If such a `student_id` cannot be found, for example `'260245896'`, then raise a `ValueError` with a message `'A student with id 260245896 either doesn't exist or is not reported as sick.'`

```
>>> students_list = contact_tracker.get_contacts_by_student_id('260996175')
>>> print(students_list)
[Mark (260248711), Leanne (260476020)]
```

- **`get_all_contacts`**: takes no parameters and returns a dictionary where the keys are the student IDs of sick students. For each sick student the value in the dictionary is the list of `Student` objects corresponding to the students that the sick student had contact with.

```
>>> print(contact_tracker.get_all_contacts())
{'260808934': [Paul (260840155), Mark (260248711), Carol (260996175), Leanne (260476020),
Will (260561504)],
'260996175': [Mark (260248711), Leanne (260476020)],
'260675874': [Paul (260840155)],
'260476020': [Sarai (260758421)],
'260386543': [Carol (260996175), Mark (260248711), Leanne (260476020), Will (260561504)],
'260248711': [Philip (260212160), Zach (260970944)],
'260840155': [Zach (260970944)],
'260561504': [Leanne (260476020), Mark (260248711)],
'260970944': [Philip (260212160)]}
```

- **`patient_zeros`**: takes no parameters and returns a list of sick students (`Student` objects) who are the possible patient zero(s). In this setting, we define the **patient zero** to be a sick student who didn't contract the virus from any McGill student, meaning the student didn't appear in any other sick student's contact list. Assume there is always a patient zero in the dataset.

```
>>> print(contact_tracker.patient_zeros())
[Bob (260808934), Farley (260675874),  Larry (260386543)]
```

- **`potential_sick_students`**: takes no parameters and returns a list of students (`Student` objects) who are not reported to be sick, but might be sick because they appear in a sick student's contact list. If there aren't any such students, the function should return an empty list.

```
>>> print(contact_tracker.potential_sick_students())
[Philip (260212160), Sarai (260758421)]
```

- **`sick_from_another_student`**: takes no parameters and returns a list of sick students (`Student` objects) who got sick from another student, i.e. they appeared in a contact list of a sick student (that's probably how they got sick). In other words, these are the sick students in the contact list who are neither patient zero(s) nor potentially sick. If there aren't any, the function should return an empty list.

```
>>> print(contact_tracker.sick_from_another_student())
[Carol (260996175), Leanne (260476020), Mark (260248711),
Paul (260840155), Will (260561504), Zach (260970944)]
```

- **`most_viral_students`**: takes no parameters and returns a list of the most viral student(s) (`Student` objects). The **most viral student** is the student who contacted the largest number of other students, meaning the student who had the longest list of contacts. There can be multiple most viral students; in that case, all the most viral students should be included.

```
>>> print(contact_tracker.most_viral_students())
[Bob (260808934)]
```

- **`most_contacted_student`**: takes no parameters and returns a list of the most contacted student(s) (`Student` objects). The **most contacted student** is the student who is not reported as sick, but has been in contact with the most amount of sick students, so most likely the student is also sick and thus should contacted by contact tracers. Note that there can be more than one most contacted students; in that case, all the most contacted students should be included. If there aren't any, the function should return an empty list.

```
>>> print(contact_tracker.most_contacted_student())
[Philip (260212160)]
```

- `ultra_spreaders`: takes no parameters and returns the list of spreader students (`Student` objects).

  An **ultra spreader** is a sick student who **only** has had contact with potentially sick students (no student in their contact list is sick yet). If there aren't any, the function should return an empty list.

```
>>> print(contact_tracker.ultra_spreaders())
[Leanne (260476020), Zach (260970944)]
```

- `non_spreaders`: takes no parameters and returns the list of non-spreader students (`Student` objects). A **non-spreader** is a sick student that had contact **only** with other sick students. If there aren't any, the function should return an empty list.

```
>>> print(contact_tracker.non_spreaders())
[Bob (260808934), Carol (260996175), Farley (260675874),  Larry (260386543),
Paul (260840155), Will (260561504)]
```

(Extra 8 points) Bonus methods (do not define the methods below if you are not going to implement them):

- `min_distance_from_patient_zeros` (6 points): takes a `student_id` of a student as an input and returns an int corresponding to the students minimum distance from a patient zero. If such a `student_id` cannot be found, for example `'260245896'`, then raise a `ValueError` with a message `'A student with id 260245896 either doesn't exist or is not reported as sick.'` If the student is a patient zero themselves, then the minimum distance is 0, otherwise if the student appears in a patient zero's contact list, then the minimum distance between the student and a patient zero is 1, and so on. You can assume that each student is either a patient zero or has a path to at least one patient zero.

  You can also assume that there are no *cycles* of any length in `cases_with_contacts`. For example, if the sick student B appears in the sick student A's contact list (I will denote it by $A \to B$, reads A infected B), and the student A appears in B's contact list ($B \to A$), that would be cycle of length 2 ($A \to B \to A$) because in 2 arrows you can get from A to A. This cannot happen. Similarly, there will be a cycle of length 3 if we have $A \to B \to C \to A$, meaning the sick student B appears in the sick student A's contact list ($A \to B$), the sick student C appears in B's contact list ($B \to C$), and finally, A appears in C's contact list ($C \to A$). This cannot happen either. Cycles of any length, $A \to B \to C \to \ldots \to A$ cannot appear.

```
>>> contact_tracker.min_distances_from_patient_zeros('260808934')
0

>>> contact_tracker.min_distances_from_patient_zeros('260476020')
1

>>> contact_tracker.min_distances_from_patient_zeros('260970944')
2
```

- `all_min_distances_from_patient_zeros`(2 points): takes no parameters and returns a dictionary where the keys are all the IDs of all the students (from `students` attribute). The value for each student ID is the corresponding student's minimum distance from patient zero.

```
>>> print(contact_tracker.all_min_distances_from_patient_zeros())
{'260808934': 0, '260840155': 1, '260248711': 1, '260996175': 1, '260476020': 1,
'260561504': 1, '260675874': 0, '260758421': 2, '260386543': 0, '260212160': 2,
'260970944': 2}
```

Part 3: `contactTracingSystem` – putting all together

Implement the following functions in `contactTracingSystem.py` file. In addition, you can also define your own helper functions.

Stage 1: Data input

- **`load_file`**: takes a string parameter `file_name`, opens the file with the given file name and returns its content as a string. If any error occurs when trying to *read* the file after the file has been opened, then the function should close the file before the exception is raised. If the file didn't open successfully, do not handle that case in this function.

  Note: No examples are required for the docstring for this function.

Stage 2: Data preparation

- **`JSON_to_students`**: takes a string in JSON format containing students' list (in the format that is indicated in `cases.csv`) and returns a list of `Student` objects. The function assumes that the parameter is a valid JSON object.

  Hint: Use the **`from_JSON`** function from the `Student` class to parse the json for each student.

```
>>> data = load_file("all_students.json")
>>> print(JSON_to_students(data))
[Bob (260808934), Paul (260840155), Mark (260248711), Carol (260996175), Leanne (260476020),
Will (260561504), Farley (260675874), Sarai (260758421), Larry (260386543),
Philip (260212160), Zach (260970944)]
```

- **`csv_to_dictionary`**: takes a string in CSV format and returns a dictionary where the keys are the sick students' ids, and for each sick student the value is the list of student IDs that the sick student had contact with.

```
>>> data = load_file("cases.csv")
>>> print(csv_to_dictionary(data))
{'260808934': ['260840155', '260248711', '260996175', '260476020', '260561504'],
'260996175': ['260248711', '260476020'],
'260675874': ['260840155'],
'260476020': ['260758421'],
'260386543': ['260996175', '260248711', '260476020', '260561504'],
'260248711': ['260212160', '260970944'],
'260840155': ['260970944'],
'260561504': ['260476020', '260248711'], '260970944': ['260212160']}
```

Stage 4: Data output/interpretation

- **`build_report`**: takes an object of type `ContactTracker`, constructs and returns string that is the entire text in the file `contact_tracing_report_example.txt`. The report is built by calling all the functions of `ContactTracker` in the order that they are listed in the assignment description above. (If the two bonus functions are implemented, then they should appear at the bottom of the report.)

  If any of the functions from the `ContactTracker` class return an empty list, then the report should say `none` in the place it had to list the names of the students. For example, if aren't any potential sick students, the report for that line (line 13 in `contact_tracing_report_example.txt`) should read: `Potential sick students: none` Similarly, if a sick student haven't had contact with anyone (for example if the students name and ID `Alice (260245896)`), then in the report under `Contact Records:` the line for the student should read: `Alice (260245896) had contact with none`

  Note: No examples are required for the docstring for this function.

Stage 5: Data storage

- `write_in_file`: takes two parameters – a string `file_name` and a string `text`, and does not return anything. The function writes the value of `text` in the specified file. If the file exists the function must overwrite it, otherwise must create a new one. If any error occurs when trying to *write* in the file after the file has been opened successfully, then the function should close the file before the exception is raised. If the file didn't open successfully, do not handle that case in this function.

  Note: No examples are required for the docstring for this function.

Put everything together:

- `main`: takes no parameters and returns nothing. The function's job is to go through the process of data analysis through all the stages in the order specified above: it should take the data from the initial files, create an instance of `ContactTracker` class, build the report and write the final report in a file called `contact_tracing_report.txt`. Finally, the function is also responsible for the following – if any of the initial files `all_students.json` or `cases.csv` is not found, your program should not raise an exception. Python normally raises a `FileNotFoundError` exception in this case; if this exception is encountered while opening these files, then your program should catch the error and print the message `"Sorry, the file " + file_name + " could not be found."`, where the `file_name` is the name of the file that is missing. Afterwards, the program should exit without building the report. Note, the `FileNotFoundError` error should be caught in this function, and not in the functions `load_file` and `write_in_file`.

  Note: There will be no public tests to test this function. This function is for you to test your own code. Only the final content of `contact_tracing_report.txt` generated by your code will be tested. Thus, you are not required to write examples in this function's docstring.

What To Submit

You must submit all your files on codePost (https://codepost.io/). The files you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`student.py`
`contactTracker.py`
`contactTracingSystem.py`
`README.txt` In this file, you can tell the TA about any issues you ran into while doing this assignment. If you point out an error that you know occurs in your program, it may lead the TA to give you more partial credit.
Remember that this assignment like all others is an `individual` assignment and must represent the entirety of your own work. You are permitted to verbally discuss it with your peers, as long as no written notes are taken. If you do discuss it with anyone, please make note of those people in this `README.txt` file. If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.

You may make as many submissions as you like, but we will only grade your final submission (all prior ones are automatically deleted).

Note: If you are having trouble, make sure the names of your files are exactly as written above.

Assignment debriefing

In the week following the due date for this assignment, you will be asked to meet with a TA for a 10-15 minute meeting. In this meeting, the TA will grade your submission and discuss with you what you should improve for future assignments.

Only your code will determine your grade. You will not be able to provide any clarifications or extra information in order to improve your grade. However, you will have the opportunity to ask for clarifications regarding your grade.

You may also be asked during the meeting to explain portions of your code. Answers to these questions will again not be used to determine your grade, but inability to explain your code may be used as evidence to support a charge of plagiarism later in the term.

Details on how to schedule a meeting with the TA will be shared with you close to the due date of the assignment.

If you do not attend the meeting, you will not receive a grade for your assignment (i.e., you will receive a 0), which has been our policy for all assignments in this class, and was specified in the Course Syllabus.

Acknowledgements

This assignment is built on Ben Stephenson's and Jonathan Hudson's "Food Webs or the Zombie Apocalypse" assignment, which was selected as a Nifty Assignment at SIGCSE 2022. We thank them for the idea of ContactTracker class and for the data set.