

Assignment 1: The MewbileTech phone company

IMPORTANT NOTES:

1. START EARLY!

- It's important that you start early on the Assignment so that you get help in a timely manner. We *cannot guarantee answers on the discussion board on the last 12 hours before the deadline!*

2. PRIVATE POSTS

- If you are asking about ideas from your implementation, or include pieces of code, *your Piazza post must be set as private to all Instructors.*
- If it is a general question on A1, please use a public post, so that we answer the same question only once and avoid clutter. Please use the search feature as well to find if your question was asked before.

3. DEBUGGING:

- It is *your responsibility to write tests and debug your code to make sure it works and conforms to specifications.*
- If you need help, *you must show us what you've tried to debug your code first, or what tests you've written, as applicable.*

4. TESTING:

- We want to help you so we give you some very basic sample tests for A1. However, these are *only to help you start testing your code!* If you pass these sample tests, *it only means you're on track, not that you will pass our own tests.*
- You have to write your own (*good!*) tests to make sure your code works. You may not share tests with others, everyone has to get used to writing tests, as it's an important skill to develop.

5. TWO MARKUS ENTRIES:

- Please be careful to *submit the correct files under "A1" and "A1-writing" on MarkUs.* Your code and your Documentation.pdf files need to be submitted *separately*, as indicated further in the handout! You *do not have grace tokens for the documentation part*, to discourage adding the documentation last minute as an after-thought, rather than developing it along with the code!

6. HELP CENTRE:

- Please note the **Help Centre hours** posted on the discussion board, which are additional TA-run office hours we are providing you for help with assignments. Make sure to take advantage of all these supports we give you, as we want everyone to get enough support and to succeed.

Table of contents

- [Assignment 1: The MewbileTech phone company](#)
 - [Table of contents](#)
 - [General guidelines](#)
 - [Learning Goals](#)
 - [Introduction](#)
 - [Phone Company Specifications](#)
 - [Warmup Task A: Get familiar with the input data format](#)
 - [Warmup Task B: Starter code, setup, and initial run](#)
 - [Task B.1: Run the starter code](#)
 - [Task B.2: Understand how data is loaded](#)
 - [Task 1: Reading and recording call events](#)
 - [Task 2: Really recording call events](#)
 - [Task 2.1: Complete class CallHistory](#)
 - [Task 2.2: Complete class PhoneLine](#)
 - [Task 2.3: Complete class Customer](#)
 - [Task 3: Contracts](#)

- [Task 3.1: Implement term contracts](#)
- [Task 3.2: Implement month-to-month contracts](#)
- [Task 3.3: Implement prepaid contracts](#)
- [Task 3.4: Give customers the right type of contract for each phone line](#)
- [Task 3.5: Documentation](#)
- [Task 4: Filtering events and displaying bills](#)
 - [Task 4.1: Implement the filters](#)
 - ["Task" 4.2: Display bills](#)
- [Task 5: Speeding up execution](#)
 - [Task 5.1: Experiment with different amounts of parallelism](#)
 - [Task 5.2: Slow things down to magnify the effect](#)
- [Polish!](#)
- [Submission instructions](#)

General guidelines


- You must complete this assignment individually.
- You must not change the starter code other than by filling in missing method and class bodies as described in your tasks below, or uncommenting the parts we tell you to (marked with "TODO:" comments in the starter code). It's always okay to add helper methods, however.
- The tasks are not designed to be equally difficult, or even in increasing order of difficulty. They are just laid out in logical order.
- Although implementing a complex application can be challenging at first, we will guide you through a progression of tasks, in order to gradually build pieces of your implementation. However, it is your responsibility to read through this handout and the starter code we provide, **carefully**, and to understand how the classes work together in the context of the application.

Learning Goals

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

- read code you didn't write and understand its design and implementation, including:
 - reading the class and method docstrings carefully (including attributes, representation invariants, preconditions, etc.)
 - determining relationships between classes, by applying your knowledge of composition and inheritance
- complete a partial implementation of a class, including:
 - reading the representation invariants to enforce important facts about implementation decisions
 - reading the preconditions to factor in assumptions that they permit
 - writing the required methods
- implement a class from a provided specification, including:
 - defining instance attributes and methods
 - writing the class documentation and method docstrings
 - implementing the class functionality according to specs
 - using inheritance: defining a subclass of another class
- write technical documentation (see also the guidelines provided in class)
- learn about speeding up execution using parallelism

Introduction

MewbileTech (Logo: ) is a new phone company who wants to get a foothold on the Toronto mobile phone market. They are asking for your help in building software to keep track of their historic customer data. They want the program to read in a dataset of customer calls, set geographically in Toronto, and produce a visualization of this past calling activity on a real map of the city. The MewbileTech operator should then be able to filter results, (for example, displaying only calls and texts from a particular customer), or display a detailed bill that a given customer would have been sent during a specific month. We use

the term “to filter” to indicate that we want to display only those calls which match the search criterion.

Please note that this is not a “real-time” system where calls would be recorded as they are made, but rather a system to visualize calls that have already been made. It is intended for use by a MewbileTech operator to analyze customers’ calling patterns.

The classes you will implement include some methods that are never used in the present application, such as a method for cancelling a contract. These provide services that MewbileTech anticipates will be used in future applications.

Phone Company Specifications

A MewbileTech **customer** has a unique id number, and one or more phone lines. Each **phone line** has a unique phone number associated with it (which cannot change once assigned to this phone line), a contract, the call history associated with this phone line, and the monthly bill for every month when the line was in use under the current contract. A phone line can change contracts over time, but the application you are writing does not offer that operation to the MewbileTech operator.

A **call** has a source number, destination number, the time when the call was made, the duration of the call, and the geographic locations (as a pair of longitude+latitude coordinates) of both the source of the call and the destination of the call.

The **call history** of each customer is a record of all outgoing and incoming calls.

A **contract** is an agreement between MewbileTech and a given customer. Contracts come in three types: **Month-to-Month Contract**, **Term Contract**, and **Prepaid Contract**. The type of contract determines things like whether the customer must commit to a term (*i.e.*, a period of time), the rate charged per minute of voice time, and the number of included (free) minutes per month. Details on each contract type are provided in Task 3.

A customer can have each of their phone lines under a different type of contract, but each line is associated with exactly one contract.

As any respectable company, MewbileTech must be able to bill its customers correctly. A monthly **bill** has a number of billed minutes, as well as the free minutes included in the customer’s contract (if any) that have been used in that monthly billing cycle.

Note: Although, as you will see, the application can display the bill for any customer in any month, it does not have any capabilities related to payments. Each bill is generated assuming there was no unpaid carry-over from the previous month.

Warmup Task A: Get familiar with the input data format

Download the [Assignment 1 materials](#) and unzip the zip file into your `assignments/a1/` folder.

The customer data, including calls, are stored in a file that is in a format called JSON. JSON format stores data as key-value pairs, similar to dictionaries in python. If you’re curious to explore this format in depth, we recommend this introduction to the [JSON format](#). However, we will give you all the information you need about how to interpret the structure of the data you will be working with, and a sample dataset in a file called `dataset.json`. This dataset is not real; it was automatically generated. Open it up and review its structure and content. To help you understand the structure, we are also providing a smaller dataset called `data.py` which is formatted to make the structure of the data stand out. Here is some information that will help you find your way around:

- The data contains two keys, `events` and `customers`.
- The value for the `events` key is a list of call and SMS events (aka text messages) for all customers. **NOTE:** We will ignore the SMS events in this assignment. You are welcome to think of how you would extend the MewbileTech application to support SMS events too, but this is out of the scope of this assignment.
- Each event in the list has several key-value pairs, representing the type of event, source and destination number, time when the event occurred, location (longitude/latitude coordinates), etc.
- The value for the `customers` key is a list of customers.
- Each customer in the list has two key-value pairs: the key `lines` is associated with a list of phone lines, and the key `id` is associated with the id of the customer who owns those lines.

Warmup Task B: Starter code, setup, and initial run

Task B.1: Run the starter code

Before you go into the starter code, you should know that aside from the python code, sample tests, and the sample

dataset, we provide you with a **diagram (in the file StarterCodeArchitecture.pdf)** to help you visualize some of the key classes. You may have a peek over this diagram and think about relationships between the classes, but don't worry if you don't understand it all in one go. Remember to revisit this diagram as you follow our instructions in the next tasks.

We'll dig in to the code shortly. But first, do the following to make sure your computer is set up properly to run the program.

1. Open the file `application.py` in PyCharm and skim through it. This module contains the main program which drives the entire application. Don't worry if you don't understand everything, we will guide you through this at the right time.
2. Run that file. You will see a new window open up showing a blank map of Toronto. This is all the program does right now, but you are about to change that! You can close the window by pressing the X in the corner, as you would any other window.

Task B.2: Understand how data is loaded

Important: you may not modify anything below `if __name__ == '__main__':` in `application.py` (or any of the other modules).

This main program uses a function called `import_data()`, that we have given you, to import data from a json file. This function uses Python's `json` module to read and parse json data. If you are curious to learn how the `json` module's `load()` function works, you can familiarize yourselves with it [here](#).

`import_data()` takes an open file and stores all its contents in a dictionary, which is then returned. This dictionary contains the `customers` key and the `events` key, as loaded from the input file format you learned about in Task A. Make sure that you understand the structure of the dictionary returned by this function; in the coming tasks, you will need to pull information out of this dictionary.

Next, have a look over the provided `create_customers()` function in `application.py`. This function's role is to create instances of the `Customer` class, using the data loaded from the json file.

This function goes through the loaded data (passed in through the `log` argument) and instantiates `Customer` objects, then return them in a list. In the `log` dictionary, the value corresponding to the `customers` key contains a list of customers, where each customer is stored as a dictionary itself, with the following keys:

- key 'id' corresponds to a value representing the customer id
- key 'lines' corresponds to a list of phone lines for this customer. Each phone line in turn is a list of dictionaries, with the following keys:
 - `number`: the phone number associated with the phone line
 - `contract`: the contract type associated with the phone line

You will implement contracts later, so for now, just leave commented out the lines of code where contracts are instantiated, as mentioned in the "TODO" comments. After you have completed Task 3.4, you will come back and uncomment these lines of code.

Task 1: Reading and recording call events

You are now ready to begin writing code! **A note about doctests:** we've omitted doctests from most of the starter code, and you aren't required to write doctests for this assignment. You will write your tests in `pytest` instead. This is because, in this assignment, initializing objects for testing is more involved than usual, and would make for messy docstrings. However, you will be required to follow all other aspects of the class design recipe.

Your first code-writing task is to implement the `process_event_history()` function in the `application` module. The role of this function is to instantiate `Call` objects and store each of them in the right customer's records.

The parameter `log` is a dictionary containing all calls and SMSs from the dataset. Its key `events` contains a list of call and SMS events, which you will go through, creating `Call` objects out of events of type "call", and ignoring events of type "sms".

Each event you extract from the `log` dictionary is itself a dictionary. A call event has the following keys:

- 'type' corresponds to the type of event ("call" or "sms")
- 'src_number' corresponds to the caller's phone number
- 'dst_number' corresponds to the callee's phone number
- 'time' corresponds to the time when this call was made (for example: "2018-01-02 01:07:10")
- 'duration' corresponds to the call duration (in seconds)
- 'src_loc' corresponds to the caller's location (longitude+latitude)
- 'dst_loc' corresponds to the callee's location (longitude+latitude)

We have provided the first three lines of code in this method, to show you how to extract data from the dictionary and to give you an example of using the `datetime` module. You will need to familiarize yourselves with this module by reading the [datetime documentation](#)

As you go through the list of events and instantiate `Call` objects, you must also record these in the right Customer's records. This must be done by using the `Customer` class API, so please consult it to find the methods for registering incoming or outgoing calls. You will implement some of these Customer methods in the next task so calling them now will have no effect, but put the calls in. As we discussed in lectures, in order to use an API, all you need to know is the interface, not the implementation.

Additionally, as you instantiate new events, every time a new month is detected for the current event you are processing (whether it's a call or SMS!), you must advance all customers to a new month of contract. This is done using the `new_month()` function from the `application.py` module.

To help you understand this better, we define a few concepts which we will use in this assignment:

1. **Advancing to a new month** is the action of moving to a new month of contract, which must be done for the purposes of billing. The `new_month()` function in this module advances the month for every customer for each of their phonelines, according to the respective contract types (remember that customers can have multiple phone lines each, and that each phone line can be under a different type of contract).
2. **Gap month** is defined as a month with no activity for any customer (no calls or SMSs), despite there being activity in the month prior to the gap month, as well as in the month after the gap month.

We are making several simplifying assumptions, and enforcing them via preconditions in the code:

- The input data is guaranteed to be sorted chronologically. Therefore, advancing to a new month of contract can be safely done. For example, once you encounter an event from February 2019, you are guaranteed the next events you process cannot be from a month before February 2019.
- There is no gap month in the input data. This implies that for the timespan between the first event and the last event in the dataset, there is no month with zero activity from all customers. In other words, while one customer could have zero activity during a specific month X, there is at least one other customer who had some activity during that month. Therefore, according to the definition of advancing a month, customers with zero activity for a specific month will still get a Bill instance created. This bill will just have whatever baseline cost the corresponding contract specifies (more on this when we get to contracts in a later task).

Important: We are providing you with a `find_customer_by_number()` function. You must use it to figure out which customer a number belongs to. (Our autotesting depends on this, and you will lose marks if you don't.)

Check your work: Write pytest tests to make sure your code so far works according to specifications. It is your responsibility to write good tests!

Task 2: Really recording call events

Your code from Task 1 has called some methods that are not yet implemented. In this task, you'll write those methods, so that call information will be recorded inside objects created by the program.

Task 2.1: Complete class `CallHistory`

Open `callhistory.py` and read the docstrings for the `CallHistory` class and its methods. We have already partially implemented this class and designed some of its methods for you; do *not* modify the attributes or initializers here. Your job is to complete the following methods, according to the specifications we've given you in the code:

- `register_outgoing_call`
- `register_incoming_call`

Task 2.2: Complete class `PhoneLine`

Next, open `phoneline.py` and read the documentation we've provided for the `PhoneLine` class. You will have to review the starter code we provided for customer billing in `bill.py` as well as the `Contract` class in `contract.py`. You will see that `Contract` is an abstract class representing a generic contract; we will add specific contract types in a later task.

After you understand the connections between these three classes, your task is to implement the following methods in the `PhoneLine` class, according to the specifications from the docstrings:

- `make_call`

- `receive_call`

Task 2.3: Complete class `Customer`

Finally, open `customer.py` and read the docstrings for the `Customer` class and its methods. We have already partially implemented this class, so make sure *not* to modify the attributes or initializers here. You must complete the following methods though, according to the specifications we've given you in the code:

- `make_call`
- `receive_call`

Hint: make sure to add the calls to the correct phone line of the customer. You should be familiar at this point with the `PhoneLine` API.

Visualize your work: At this point you should be able to visualize calls within the application. Run the application in `application.py` and make sure that the calls are now being displayed. At the start the visualization displays all the Calls (round phone images), scattered on the map, with lines indicating the connection between the source and destination of that Call!

Your application should not crash at this point but filter actions will not work until you finish task 4.

Check your work: Write `pytest` tests to make sure your code so far works according to specifications. It is your responsibility to write good tests!

Task 3: Contracts

As we saw earlier, the starter code includes a `Contract` class in `contract.py`, which contains a basic set of attributes and methods. Each contract contains at the very least a start date and a `Bill` object.

We are making some simplifying assumptions:

- regardless of the contract type, the billing cycle starts on the 1st of the month and ends on the last day of the month.
- incoming calls are free.
- for any given contract type, the rate charged for a minute is always the same. In other words, all contracts of the same type charge the same rate per minute, but different contract types may charge different rates.

The interaction between contracts and bills is something you have to piece together carefully, so we will give you some guidance to see the “big picture”, as follows:

- whenever a new month is encountered in the historic data from the input dataset, we “advance” to a new month of contract all customers and all phone lines. You did this already in `application.py`, by calling the `new_month()` function from that module in the `process_event_history()`, whenever the timestamp of an event indicates a new month is encountered.
- given that historic records get loaded gradually, the `bill` attribute of a contract always represents the “latest” bill (the one for the month we are currently loading events for). Once the data is fully processed, the `bill` is basically corresponds to the last month encountered in the input data.
- notice that the contract instance has the information on how much a call should be charged, whether a call should be free or billed, etc. Therefore, you must ensure that each monthly bill can get this information, whenever you advance the month.

To understand the interaction between contracts and bills, ask yourself these questions:

- is this the first month of the contract?
- do you get new free minutes for the contract type?
- how do you handle billing a call?
- what is specific to each contract type?

Your task: implement classes for each of the three types of customer contracts: month-to-month, term, and prepaid. The new classes must be called `MTMContract`, `TermContract`, and `PrepaidContract` respectively.

To prepare for this task, you need to do two things:

- First, review all the variables we have provided in module `contract` that store predefined rates and fees. *You must not modify these.*
- Then, make sure you understand the `Bill` class, defined in module `bill`. Pay particular attention to methods `set_rates()` and `add_fixed_cost()`. They will be important when you need to create a new monthly bill.

Task 3.1: Implement term contracts

A **term contract** is a type of `Contract` with a specific start date and end date, and which requires a commitment until the end date. A term contract comes with an initial large **term deposit** added to the bill of the first month of the contract. If the customer cancels the contract early, the deposit is forfeited. If the contract is carried to term, the customer gets back the term deposit. That is, if the contract gets cancelled **after** the end date of the contract, then the term deposit is returned to the customer, minus that month's cost.

The perks of the term contract consist of:

- having a lower monthly cost than a month-to-month contract
- lower calling rates, and
- a number of free minutes included each month, which refresh when a new month starts. Free minutes are used up first, so the customer only gets billed for minutes of voice time once the freebies have been used up.

To prepare, have a look over the `TERM_DEPOSIT` fee defined in `contract.py`, as well as the corresponding rate per minute of voice time.

You can assume that the bill is paid on time each month by the customer, so you don't have to worry about carrying over the previous month's bill for a Term Contract. It is important to note that a customer in a Term Contract can continue with the same contract past its end date (at the same rate and conditions), and can do so until the contract is explicitly cancelled. When the term is over (and the contract hasn't been cancelled), instead of adding a new term deposit and refunding the old one, the deposit simply gets carried over and does not get refunded until the contract gets cancelled.

Your task: Implement class `TermContract` as a subclass of `Contract`. The `TermContract` methods must follow the same interface as the `Contract` class. Consider each aspect of a term contract carefully!

Task 3.2: Implement month-to-month contracts

The **month-to-month contract** is a `Contract` with no end date and no initial term deposit. This type of contract has higher rates for calls than a term contract, and comes with no free minutes included, but also involves no term commitment. Have a look at `contract.py` for all the rates per minute for each type of contract.

Just like for a Term Contract, you can assume that the bill is paid on time each month by the customer, so you don't have to worry about carrying over an unpaid bill.

Your task: Implement the `MTMContract` class as a subclass of `Contract`. The `MTMContract` class methods must follow the same interface as the `Contract` class. Consider each aspect of this contract carefully!

Task 3.3: Implement prepaid contracts

A **prepaid contract** has a start date but does not have an end date, and it comes with no included minutes. It has an associated **balance**, which is the amount of money the customer owes. If the balance is negative, this indicates that the customer has this much credit, that is, has prepaid this much. The customer must prepay some amount when signing up for the prepaid contract, but it can be any amount.

When a new month is started, the balance from the previous month gets carried over into the new month, and if there is less than a \$10 credit left in the account, the balance must get a top-up of \$25 in credit (again, keep in mind that a negative amount indicates a credit). When cancelling the contract, if the contract still has left some credit on it (a negative balance), then the amount left is forfeited. If the balance is positive, this should be returned by the `cancel_contract()` method, so that an application can notify the customer about a remaining balance to be paid (although our application does not do this).

Your task: Implement the `PrepaidContract` class as a subclass of `Contract`. The `PrepaidContract` class methods must follow the same interface as the `Contract` class. Consider each aspect of this contract carefully!

Task 3.4: Give customers the right type of contract for each phone line

Now go back and update the code in the `create_customers()` function in the main application module, by following the two **TODO lists** found within the function.

As you might notice, a dataset does not contain the start date (or end date, where applicable) for a contract, nor does it specify an initial balance for a `PrepaidContract`. In the code we provided you, we use the following (arbitrarily chosen) values:

- all `MTMContracts` start on the 25th of December 2017.
- all `TermContracts` start on the 25th of December 2017 and end of the 25th of June 2019.
- all `PrepaidContracts` start on the 25th of December 2017.

Note: Changing contracts is not supported by the user interface of the application, however, you might be wondering how some other application could support it. We are making the simplifying assumption that in order to change contracts, a customer must first cancel the corresponding phone line at the end of the contract, and then an operator can help the customer create a new phone line with the same number but under a new contract.

Visualize your work: At this point you should still be able to visualize your work within the application without your application crashing. Filter actions will still not work until you finish task 4.

Check your work: Write pytest tests to make sure your code so far works according to specifications. It is your responsibility to write good tests!

Task 3.5: Documentation

Although it is implicitly expected that you will document your code, we will explicitly require you that you do so, and to additionally write a graded piece of documentation for all the contract classes that you implemented. See [Documentation task](#) for further details.

Note: This task is separate from your code. You will have to submit a separate file called `Documentation.pdf`, under the A1-writing submission on MarkUs.

Note: We want you to document your code as you go, so unlike for the A1 assignment on MarkUs (where you submit your code!), **you may not use grace tokens for submitting this `Documentation.pdf` file under the A1-writing assignment submission on MarkUs.**

Task 4: Filtering events and displaying bills

Now that you have the data loaded and recorded in the system, as well as contracts implemented, we are ready to add the filtering feature, which will allow the user to filter the calls that are displayed, for example so that they see only calls from a certain customer.

First, let's understand the various filters and what they will do. If you run the application, the menu bar in the visualization window informs the application user of the supported types of actions. The filtering actions are invoked by pressing the following keys:

- c: show only calls (to and from) phone lines belonging to a given customer id
- d: show only calls (made or received) with at least (or at most) a given duration
- l: show only calls (to and from) a given location
- r: reset all filters applied so far

Once the corresponding key is pressed, filters that require additional information ask the user to enter it. For example, the 'c' filter asks the user to enter the desired customer id.

Note: The first time you press a key to apply a filter, you will see a window pop-up with a welcome message from the MewbileTech management system. This window will appear on top of your filter pop-up, so you must close the welcome pop-up *before* you can proceed to applying a filter. Why does this happen? This pop-up window was intended to be displayed when you first launch the application, rather than when you press a filter key. However, the MewbileTech cat decided to meddle with the application functionality and we're stuck with this behaviour, which you can safely ignore. :)

The user can apply one filter after another. The visualization starts by showing all the calls in the system, for all customers and all their phone lines, and then each filter gradually narrows down the search by only querying the remaining records left after applying the previous filter. For example, you can apply a 'c' filter, followed by an 'l' filter, and then an 'd' filter, in order to determine all the calls made or received by a certain customer, in a given location, with a duration of over or under a number of seconds. The 'r' switch resets all the filters applied so far, and restarts the search on the entire dataset.

This is what the code *will* do. *You* need to implement all of the incomplete filter classes. Let's make it so!

Note: If you are curious to know how the actual visualization process works, you are welcome to look into `visualizer.py`, but you do not need to understand the `visualizer` module in order to solve this assignment. However, if you are curious to explore the `visualizer.py` module, we recommend that you read the docstring of the `render_drawables()` method of the `Visualizer` class. The main block in the `application` module sends it a list of `Drawables`, which it then draws them on the pygame window. You might wish to also look at the `handle_window_events()` method, which takes input from the application user and acts accordingly. You will notice a whole bunch of filter objects, of various types, for example, `DurationFilter`, `LocationFilter`, `CustomerFilter`, etc.

Task 4.1: Implement the filters

In module `filter`, review the `Filter` class and its documentation carefully. The actual filtering work is done in the `apply()`

method, which acts accordingly for each type of filter. Your task is to implement all the `apply()` methods in the filter classes which subclass `Filter`.

Each of the filters has its own definition of what is a valid user input. This is displayed to the user in the application as a visual prompt (the `__str__()` method returns the visual prompt message). Nevertheless, a user might still enter incorrectly formatted input, or may enter invalid input (e.g., a customer id which does not exist). For each filter, you must enforce “sanity” checks in the `apply()` method to ensure that your code will not crash if given an incorrect or invalid input. We want you to consider how malformed or invalid inputs (as given in the `filter_string` argument) can impact your implementation of the `apply()` method, and to make your code robust. We are not explicitly telling you what cases to consider because we want to encourage you to consider robustness during your implementation. Ultimately, if during execution of your program a cat walks across your keyboard (don't they love to?) your code should not crash and it should do the right thing.

There are two ways to check for incorrect inputs: - ensure that an incorrect input cannot crash your code by using explicit validity checks (e.g., using if statements), or - use try/except blocks to catch whichever Exception may be raised by your code due to an incorrect or invalid input. For the latter, revisit the lecture prep on Exceptions for an example of how to handle exceptions.

Visualize your work: run the `application.py` and try applying each filter. As you apply more and more filters, some calls will disappear from the map. Keep in mind that although visualization is nice for seeing your code at work, it is not a thorough way to test your code! It is your responsibility to thoroughly test your code by making sure you write pytest test cases for a variety of corner cases. And don't forget the cat test. (In the absence of a cat, your elbow will do.) No input should make the code crash. - **Important notes for Mac users:** While your code will be tested on a linux machine, you are welcome to use your own computer to work on the assignment as well. However, for those of you running this assignment on macOS, the visualization libraries that our code imports have a known bug where you cannot enter text into the filter dialogue box. A workaround we have found for this bug is that when you first launch the application, click away to another window and then back into the application window. You must do this before performing the first filter action.

Check your work: Write pytest tests to make sure your code so far works according to specifications. It is your responsibility to write good tests!

“Task” 4.2: Display bills

There is one more user action we have not discussed yet: “m” is used to display the monthly bill that a customer would have received for a given month. There is no code for you to write. We have provided you with the code for this operation. Review it, and try displaying bills in the visualization window.

Task 5: Speeding up execution

This task is optional, but it covers really cool concepts, so we strongly recommend that you do it. We are foreshadowing some concepts you will learn in later courses, but about which it is useful to form an intuition early on.

You will see later on in this course how the choice of data structures and algorithms can make a big difference in your program's efficiency. Another way of speeding up your program is to use multiple resources in parallel.

Let's consider your computer's main processing resource: the central processing unit (CPU). Your computer's CPU has multiple processors or “cores,” each capable of computations independent of the other(s). Most programs you have written so far run entirely sequentially – one instruction after another – until they are finished. However, what if we could find pieces of your program which can be run completely independently? If so, that would enable your CPU to run those pieces at the same time, but on different cores. As a result, execution should be much faster. For example, if your code processes 4 million data items in 40 seconds, and we break down this computation into 4 equal pieces of data, each with 1 million data items, and process all of them in parallel, then the entire computation should only take 10 seconds. In reality we might not get this exact speed-up, because it depends on the specific computation and many other aspects of computing in parallel. When you learn the intricacies of parallelism in later courses, you will be able to analyze the performance of your program under a variety of factors, but we want to give you a flavour of what parallel computing means so that you have the right mental model early on.

To summarize, if we can identify some work that can be done at the same time as other pieces of work, we can speed up our program's execution. Similarly, if we can find *data* which can be broken down into independent pieces, then we can perform computations on each piece at the same time as computations on other pieces.

Let's see what this could look like in our MewbileTech application. When we filter through a list of calls, we go through the list and determine which entries match our filter string. What if we could partition the data into N chunks and launch N parallel computing entities (which we call **threads of execution**, or simply **threads**), each of them responsible for filtering through one of the chunks? The results from each thread could then be “united” at the end and displayed as usual. Given that each thread will process its own separate chunk of data, our filtering should be carried out N times faster, right? Let's try

it out!

Task 5.1: Experiment with different amounts of parallelism

We have ignored the `Visualizer` class so far, but now it's time to revisit a tiny piece of it, specifically the `handle_window_events()` method in the `Visualizer` class. We have already implemented the thread creation and distribution of work to each thread for you; have a look over the `threading_wrapper()` function, if you are interested. However, your task will be much simpler, and does not require a deep understanding of the threading API.

We want you to vary the number of threads, run the application each time and apply some filters (for example, apply the D filter with a duration under 100 seconds), to get a sense of the benefits of computing filtering in parallel.

To do so, find a variable called `NUM_THREADS` located at the top of the file (line 56), currently defined as the value 1. This is the number of chunks that the data is split into, as well as the number of threads launched to filter each of the chunks of data (each thread processes one chunk). Increase this variable's value by doubling it from 1 to 2, 4, then 8, and run the application with each value of the variable (note that you will have to restart the application for the changes to take effect each time). Use the same filter each time, to make sure the measured execution times are comparable. Do you notice any difference in execution time as the number of threads increases?

If you don't notice much difference, don't worry, you're not doing anything wrong! This is solely because, with a tiny dataset, the execution is typically so fast that filtering in parallel won't make much difference.

Task 5.2: Slow things down to magnify the effect

In order to notice faster execution times, either the dataset must be much larger, or the filtering operation has to be much more time consuming than just a basic comparison operation. With a very large dataset, it might take you too long to load the data and experiment with different numbers of threads, so let's try to make the filtering operation more time consuming instead.

Go into `filter.py` and find the code for the `DurationFilter`. In your implementation of its `apply()` method, locate where you check which calls are over or under a given duration (this will likely be WITHIN a for loop). Before making each comparison operation, add a short time delay to make it take longer. To do so, add a `time.sleep(T)`, where T is a time in seconds. We recommend a very short delay, for example, 0.02 seconds; this should be a reasonable number for the size of the dataset in `dataset.json`.

Run your application again, while varying the number of threads to 1, 2, 4, and 8 for each run. Do you notice a difference now? Try with higher values like 16, 32, or 64 threads. Feel free to discuss your findings with your classmates and try to think about what your results tell you.

Since this task is optional, there are no marks assigned to getting this right, but we want you to reflect on what you notice and get some intuition behind parallel execution. We will discuss this briefly in class once the assignment deadline has passed, and you can learn about this in greater detail in later courses.

Polish!

Take some time to polish up your work. 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!
- Check your docstrings 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 print function calls.
- Remove any `pass` statement where you have added the necessary code.
- Remove the "TODO" comment wherever you have completed the task.
- Take pride in your gorgeous code!
- Make sure you have written your `Documentation.pdf` file!

Submission instructions

1. **DOES YOUR CODE RUN?!** Does it pass your thorough test suite (the additional tests you have to write, not just the sample tests)?
2. Login to MarkUs and find the A1 and A1-writing entries.
3. Submit the files `customer.py`, `phoneline.py`, `contract.py`, `callhistory.py`, `filter.py`, and `application.py` under

- A1. Don't submit any other files under A1.
4. Submit only the `Documentation.pdf` under A1-writing.
 5. On a lab machine (or your own machine, if you followed the software setup steps), download all of the files you submitted into a brand-new folder, and test your code once more, thoroughly. *Your code will be tested on the Lab machines, so it must run in that environment or a similar one (see the Software Setup page).*
 6. Congratulations, you are finished with your first **major** assignment in CSC148! You are now one step closer to being a wizard/witch who masters parser-tongue. :)



Mathematical & Computational Sciences
UNIVERSITY OF TORONTO
MISSISSAUGA

For general course-related questions, please use the discussion board.

For individual questions, accommodations, etc., please contact
the **csc148h5-2022-instructors at cs.toronto.edu** email.

Make sure to include CSC148 in the subject, and to
state your name and UtorID in the email body.