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 `call_history.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: 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!

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 uncommenting the piece of code which instantiates contracts.

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 December 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.

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: 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 **c**ustomer id
- d: show only calls (made or received) with at least (or at most) a given **d**uration
- l: show only calls (to and from) a given **l**ocation
- 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 `Drawable`s, 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.

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`, 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. 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. 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 colleagues 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.