ICS 33 Spring 2025
Project 1: *Calling All Stations*

Due date and time: *Wednesday, April 23, 11:59pm*
Final late work deadline: *Monday, April 28, 2:59am*

Git repository: *https://ics.uci.edu/~thornton/ics33/ProjectGuide/Project1/Project1.git*

Introduction

When I first started learning to write programs in the 1980s, software systems were, by and large, written to run on a single computer. It would be necessary for a program to communicate with its computer's main memory, and perhaps with peripheral devices like disk drives, but usually not any farther than that. Communication between computers was much less common, with techniques supporting it much less well-known. While computer networks did exist in those days, they were substantially less common and ran at a tiny fraction of the speed of the networks in place today, which meant that even well-heeled people and organizations couldn't use them in the ways we take for granted today; the bandwidth simply wasn't there at any price.

Today, we have ubiquitous high-speed connectivity, allowing *distributed systems* to be a common part of the real-world software landscape. In a distributed system, programs run on many machines and collaborate to solve a problem. The programs on each machine run separately, which means that they can make simultaneous progress, but this is a double-edged sword.

- Ten identical computers can expend ten times as much raw effort as one can. If all ten are running full tilt, ten times as many computations can be performed in the same amount of time.
- At any given time, each computer can only access what's stored in its own main memory, so if one of them needs information stored only within the other, communication (via a network) will need to take place.

In other words, in terms of raw work, the effect of running on many computers is multiplicative; the problem is that the raw work of one computer will only be useful if it's coordinated with the raw work being done by the others. Just like when multiple people collaborate on a task, if there's no agreement about who will do what, you can end up with duplicate effort, with two people working on the same task. If there are dependencies between tasks — one task can't be started until another is finished — then coordination will be needed, so that things will be done in the right order, and so that the output from the first can be made available as input to the second. So, an important part of any distributed system is coordinating these kinds of things, which requires careful design before the system runs, as well as computation and communication bandwidth for the coordination while it runs.

In this project, we'll scratch the surface of the ideas underlying collaboration in a distributed system, by building and testing a simulation of how information propagates between the devices that make up such a system. Our simulation will focus on the problem of communicating *alerts*, which is to say that we're concerned with one device noticing a problem and then notifying other devices about it. We're not concerned about what the alerts mean, though, nor what should be done about them; we're just interested in the communication aspect of the problem.

Along the way, you'll continue to build on your skills from prerequisite coursework by making the appropriate design decisions (e.g., determining which of Python's built-in data structures might best help you solve the various parts of the problem, deciding how best to organize your program into modules, classes, functions, and so on). Additionally, you'll continue to develop your abilities to test your programs in an automated fashion by writing unit tests, employing a few techniques that we've learned more recently to widen the reach of what can be verified by your automated tests.

A simulation of a distributed system

A *simulation* is a program that executes a simplified model of some reality, usually with the goal of allowing us to explore some aspect of that reality without having to actually experience it. There are lots of reasons why we might want a simulation. We can use one for the purposes of entertainment, like how we might use a flight simulator to allow us to experience the feeling of flight without purchasing an airplane and obtaining a pilot's license; we might instead use one to explore the impact of a potential change to a business process without having to make the change, so we can see ahead of time whether the change is likely to be profitable. (Of course, what we learn from a simulation is only useful if our model matches the reality of what we're simulating — if our model mischaracterizes our customers' actual reaction to the change in our business process, say, then our simulation will mislead us into making poor decisions — but that's a deep topic for other courses.)

In our case, we'd like to explore one aspect of how distributed systems behave without having to obtain multiple machines, write coordinated programs to run on them, and connect them via a computer network. So, rather than setting all of that up, we'll write a program that boils our problem down to its essence, with simulated devices pretending to communicate with each other, according to a set of rules that will be a simplification of reality, but that will allow our program to have a result that we hope will be indicative of reality. (Since we won't ultimately be using that result for anything beyond this project, there's less pressure on us to define our simulation model accurately; it's more important that we have a definitive result, and that we learn something about the nature of how simulations are built and tested.)

The simulation model

Our simulation model is based around a few concepts, which we'll need to understand before we can proceed into the details of how to implement and test our simulation.

- We're simulating a distributed system made up of *devices*, which we'll assume are Internet-connected electronic devices that are able to communicate with each other individually (i.e., one device can send a message to another device). It's not particularly important to us what the devices do otherwise; we only care that they're able to communicate with one another.
- Each device has a *device ID*, which is a non-negative integer value that uniquely identifies it.
- When one of the devices detects an issue that will potentially affect other devices, it raises an *alert*, with the goal of making all of the other devices aware of the issue. To do that, it sends a message to some subset of the devices, which, in turn, send messages to other devices, and so on, in a process that we call *propagation*. That way, if there are thousands of devices, the alerting device doesn't have to send thousands of messages itself; instead, it can send a message to a few of them, then rely on them to spread the word. (While ours is a dramatic simplification, this is at least a little bit similar to how peer-to-peer protocols like BitTorrent work.)
- Every alert has a *description*, which we can think of as a short string of text that uniquely identifies it, so that we can differentiate alerts from one another. When an alert is propagated from one device to another, the description is left intact, so that every device recognizes it the same way.
- When the issue causing an alert is resolved, a device performs a *cancellation*, notifying other devices that the alert is no longer active. A cancellation will include the description of the alert being canceled, so that devices can correlate a cancellation with an alert. Once a device has been notified that an alert has been canceled, it will never propagate that alert to other devices again.

- Each device is configured to propagate alerts and cancellations to a predefined collection of other devices, which we'll call its *propagation set*.
- Whenever a device receives an alert or cancellation, it immediately notifies all of the devices in its propagation set unless it's previously received a cancellation of that same alert. We'll say that this introduces a *delay* into the process, which is to say that when a device sends a message at some particular time, it will be received at some later time. (As a practical matter, such a delay might partly arise because receiving a message, processing it, and sending it out to another device takes time, but might also be an intentional slowing-down to avoid flooding the network with messages. The longer the delay, the less bandwidth is used in proagating alerts and cancellations, but the longer it takes for faraway devices to become aware of one.)

So, in general, there's a straightforward effect that we can expect from these rules. When a device raises an alert, it begins propagating among the devices, continuing to propagate — even amongst devices that have already been made aware of it — until all of the devices have been notified of its cancellation. Because communication takes time, there will naturally be a delay between the alert being raised and all devices being aware of it, just as some devices will continue propagating canceled alerts until they've been made aware of the cancellation. (We see the same kinds of effects in any kind of distributed organization, whether it's made up of computers or people; some members will be acting on outdated information at least some of the time.)

As an example, we might have four devices, which we'll call Device 1, Device 2, Device 3, and Device 4, for the purposes of discussion. Let's suppose that the propagation sets are configured as follows.

- Device 1 propagates to Device 2 with a delay of 750 milliseconds.
- Device 2 propagates to Device 3 with a delay of 1,250 milliseconds.
- Device 3 propagates to Device 4 with a delay of 500 milliseconds.
- Device 4 propagates to Device 1 with a delay of 1,000 milliseconds.

Now, suppose Device 1 raises an alert at time 0 (i.e., immediately as the simulation begins). What happens next is a cascade of communication amongst devices, which we can simulate by hand to understand how our simulation model is meant to work.

- Initially, at time 0, Device 1 is aware of the alert, but no other devices are aware. It's time for propagation to commence, so Device 1 sends a message to Device 2.

- Device 2 receives the message from Device 1 at time 750 (i.e., 750 milliseconds after the start of the simulation). At this point, both Device 1 and Device 2 are aware of the alert, but no other devices are. Device 2 now needs to propagate the alert, so it sends a message to Device 3.
- At time 2,000, Device 3 receives a message from Device 2 indicating the alert (1,250 milliseconds after it was sent), which triggers Device 3 to propagate it.
- Let's suppose at time 2,200, Device 1 cancels the alert. So, at this point, Device 1 is aware that the alert is canceled, but no other device is aware. It sends a message to Device 2 indicating the cancellation.
  - At this point, there are two messages in flight: Device 3 has sent the alert to Device 4 (sent at time 2,000), while Device 1 has sent the cancellation to Device 2 (sent at time 2,200).
- At time 2,500, Device 4 receives the message from Device 3 indicating the alert (500 milliseconds after Device 3 sent it). Since Device 4 is unaware of the cancellation of the alert, it propagates the alert to Device 1.
  - At this point, three devices — 2, 3, and 4 — believe that there is an alert, while Device 1 is still the only one currently aware that it's been canceled.
- At time 2,950, Device 2 receives the message from Device 1 indicating the cancellation (750 milliseconds after Device 1 sent it). Device 2 propagates the cancellation to Device 3.
  - Two devices (Device 1 and Device 2) now know about the cancellation, while the other two (Device 3 and Device 4) still believe there is an alert.
  - There are now two messages in flight: Device 4 has sent the alert to Device 1 (sent at time 2,500) and Device 2 has sent the cancellation to Device 3 (sent at time 2,950).
- At time 3,500, Device 1 receives the message from Device 4 indicating the alert (1,000 milliseconds after Device 4 sent it). Because Device 1 is aware that the alert has been canceled, it doesn't propagate the alert.
- At time 4,200, Device 3 receives the message from Device 2 indicating the cancellation (1,250 milliseconds after Device 2 sent it). Because this is news to Device 3, it propagates the cancellation to Device 4.
- At time 4,700, Device 4 receives the message from Device 3 indicating the cancellation (500 milliseconds after Device 3 sent it). Because it hadn't seen it before, Device 4 propagates the cancellation to Device 1.
- At time 5,700, Device 1 receives the message from Device 4 indicating the cancellation (1,000 milliseconds after Device 4 sent it). Since this is a cancellation that Device 1 is already aware of, the message isn't propagated any further.
- There are no longer any messages in flight, so, until there's another alert, all will remain quiet.

There are a couple of things worth noting about this example.

- Because each device is dealing only with limited information — it can only know what it's been told by other devices — the alert continued to be propagated for a while even after the original device canceled it. While we could limit this effect by propagating the message more widely (e.g., each device could send more than one message, for example) or more quickly (e.g., by reducing the delays somehow), this is essentially unavoidable in a distributed system where each device is acting only on its own local knowledge.
- The simplicity of our propagation rules — propagate anything you haven't seen canceled before — leads to some wasteful behavior, such as propagating the cancellation back to the original device that issued it (which will surely know about it already). In a practical situation, this might cause us to redesign our protocol a bit, but we'll skip that problem here, except to take note of how our simulation is already giving us actionable advice about our design; that's part of what makes simulations (even "simulation-by-hand," like we're doing here) so useful, by helping us to see potential downsides of our ideas before we've committed the time and effort to fully implement them.

Tightening up our simulation model

Because we want our program to have a deterministic outcome — given a particular input, we always want the same events to occur at the same times — we'll need to carefully consider what we should do about the problem of things being scheduled to happen at the same time. There are three scenarios to consider.

- What do we do when two alerts arrive at the same device simultaneously? Essentially, this is a non-issue. Regardless of the order in which we process them, we'd do the same thing with them either way, either forwarding them or not, depending on whether a cancellation had previously been received.
- What do we do when two cancellations arrive at the same device simultaneously? This, too is a non-issue. We'd put both of them on the list of alerts we'd consider to be canceled going forward, but the order in which we processed them would otherwise be irrelevant.
- What do we do when a cancellation and an alert arrive at the same device simultaneously? This is where it gets more interesting:
  - If the descriptions are different, the order in which we process them is irrelevant.
  - If the descriptions are the same, then the processing of the cancellation could potentially impact whether or not we forward the alert. If we processed the alert first, it would have been forwarded before we processed the cancellation; if we processed the cancellation first, we'd avoid forwarding the alert when we processed it later.

There's a simple rule we can employ to keep the last of these scenarios from ever being an issue for us, though: When a cancellation arrives at a device at time *n*, it only affects the propagation of subsequent alerts or cancellations arriving at that device from time $n + 1$ onwards. So, we'll employ that rule, and otherwise allow multiple events scheduled simultaneously to be processed in any order.

Getting started

Near the top of this project write-up, you'll find a link to a Git repository that provides the starting point for this project. Using the instructions you followed in Project 0 (in the section titled *Starting a new project*), create a new PyCharm project using that Git repository; you'll do your work within that PyCharm project.

You won't find that there's much provided code: Just a skeletal `project1.py` file, an empty `test_project1.py` file in which you could write unit tests for the code in `project1.py`, one sample input and one sample output file, and a `.gitignore` file to help you curate what does (and doesn't) end up committed into your Git repository. Requirements about how you organize your solution will follow a little later in this write-up.

The program

Your program will begin by reading one line of input from the Python shell (e.g., via a call to the built-in `input` function), specifying the path to an *input file* that describes the work that your program will be doing. Do not print a prompt or anything else; read this input before writing any output.

If the specified file doesn't exist, your program will then print one line of output `FILE NOT FOUND` and then terminate. Otherwise, your program will have a simulation to run, with everything driven by what's in that input file, so let's take a look at what you can expect to be in it.

The input

Your program's input file consists of lines of text. Each file will fit one of six characteristics, described below.

- A line can begin with the word `LENGTH`, followed by a space, followed by a positive integer value. This line establishes the length of the simulation, specified in milliseconds. So, for example, the line `LENGTH 600000` would mean that the simulation will run for 600,000 simulation milliseconds, numbered 0 through 599,999, with the simulation ending (and nothing else happening) at time 600,000.

- A line can begin with the word `DEVICE`, followed by a space, followed by a non-negative integer value. These lines establish the existence of one device in our simulation; the integer is the device ID that uniquely identifies the device.
- A line can begin with the word `PROPAGATE`, followed by three non-negative integer values, separated by spaces. These lines describe a rule for propagating an alert or cancellation from one device to another. The three integers specify, as follows, three things:
  - A device ID that may subsequently receive an alert or cancellation.
  - A device ID to which the received alert or cancellation should be propagated.
  - A delay, expressed as a positive integer number of milliseconds, indicating how long it will be before the propagated alert or cancellation will be received by the second device. ("Positive" means that it will never be zero.)

  So, for example, the line `PROPAGATE 1 2 100` indicates that device 1 will propagate any alert or cancellation to device 2, and that device 2 will receive it 100 milliseconds later.
- A line can begin with the word `ALERT`, followed by three values, separated by spaces.
  - A device ID that will be scheduled to raise an alert at a particular time during the simulation.
  - An arbitrary string of (non-space) characters specifying the *description* of the alert.
  - The simulation time at which the alert will be *raised*, which means that the specified device will begin the process of propagating it. This time is given as a non-negative integer number of milliseconds after the start of the simulation. ("Non-negative" means that it might be zero.)

  So, for example, the line `ALERT 1 OhNo 5000` indicates that device 1 will raise an alert with the text `OhNo` at simulation time 5000.
- A line can begin with the word `CANCEL`, followed by three values, separated by spaces.
  - A device ID that will be scheduled to cancel an alert at a particular time during the simulation.
  - An arbitrary string of (non-space) characters specifying the *description* of the alert being canceled.
  - The simulation time at which the cancellation will occur, which means that the specified device will begin the process of propagating the cancellation. This time is given as a non-negative integer number of milliseconds after the start of the simulation. ("Non-negative" means that it might be zero.)

  So, for example, the line `CANCEL 1 OhNo 6000` indicates that device 1 will cancel an alert with the text `OhNo` at simulation time 6000.

- A line can be blank or consist only of spaces. These lines are used for readability of the input file, and can be ignored by your program when detected.
- A line can begin with a # character, in which case it can be followed by zero or more characters of text with no restrictions. These lines are used as comments in the input file — similar to comments in a Python program — and can be ignored by your program when detected.

There is no restriction about the order in which you'll find the lines of the file. Any of those kinds of lines can appear anywhere in the file.

However, you can assume that the input will be formatted according to those rules; we won't be testing your program with input files that don't meet those requirements, so your program can do anything (including crash) if given such an input file.

There are also a few other assumptions you can safely make about the input (i.e., we won't test scenarios where these assumptions aren't true, so we don't care how — or if — your program handles them).

- The simulation's length will always be specified exactly once.
- No time listed in the input will occur after the scheduled end of the simulation.
- No pair of devices listed in the input file will have the same device ID.
- No propagation rule will list the same two device IDs (i.e., devices never propagate to themselves).
- No pair of propagation rules will list the same two device IDs in the same order (i.e., if there's a rule specifying that device ID 1 propagates alerts to device ID 2, it is definitive).
- No pair of alerts will have the same description.
- No pair of cancellations will have the same description.
- No propagation rules, alerts, or cancellations will make reference to device IDs that are not also defined elsewhere in the input file.

The output

As your simulation progresses, your program will produce output that describes messages as they're sent and received, printing that output to the standard output. There are four interesting events in our simulation, each of which warrants a line of output.

- A device receives an alert.
  - `@100: #1 RECEIVED ALERT FROM #2: OhNo` means that, at simulation time 100, device 1 received the alert with description `OhNo` from device 2.
- A device sends or propagates an alert.
  - `@50: #2 SENT ALERT TO #1: OhNo` means that, at simulation time 50, device 2 sent the alert `OhNo` to device 1.
- A device receives a cancellation.
  - `@200: #1 RECEIVED CANCELLATION FROM #2: OhNo` means that, at time 200, device 1 received a cancellation of the alert with description `OhNo` from device 2.
- A device sends or propagates a cancellation.
  - `@150: #2 SENT CANCELLATION TO #1: OhNo` means that, at time 150, device 2 sent a cancellation of the alert with description `OhNo` to device 1.
- The simulation ends.
  - `@600: END` means that, at time 600, the simulation ended.

Your program's output must exactly match this output specification, character-for-character, with a newline on the end of every line (including the last one). Spelling, capitalization, and spacing are all relevant.

When two events are scheduled to occur at the same time, the order in which your program prints the lines of output describing them is irrelevant. Earlier events must be printed before later ones, but simultaneously occurring events can appear in the output in any order.

How the simulation ends

Your simulation will continue running for its scheduled duration, ending at the scheduled time, regardless of whether there are still messages propagating.

If the simulation is scheduled to end at a particular time, nothing else happens within that same simulation millisecond — nor after that time — even if scheduled. For example, if the simulation length is described in the input as `LENGTH 600000`, that means there are 600,000 simulation milliseconds, numbered from 0 through 599,999. The only thing that happens at time 600,000 is the ending of the simulation.

Sample input and output

After you create a PyCharm project from the provided Git bundle, you'll find a directory named `sample` that contains two files, `sample_input.txt` and `sample_output.txt`, which are a sample input and output, respectively. The `sample_input.txt` file represents the example scenario described in the section titled *The simulation model* above. If you run your program using `sample_input.txt` as the input file, its output should exactly match (on a character-for-character basis) what's in `sample_output.txt`.

We will not be providing additional samples; beyond this, testing is your responsibility, as it usually is when you develop programs.

The meaning of time in your simulation

It's worth noting that your goal here is building a simulation, which is defined generally as a way of exploring a possible reality without actually having to experience that reality. What we learn from a simulation might inform our real-world decisions — if, for example, a simulation tells us that an idea is a good one, and we believe our simulation's model reasonably reflects reality, then we might be inclined to follow through with our idea.

In our case, our goal is not to experience the passage of time in our simulation; it's to measure the possible passage of time in the world that we're simulating, while obtaining a result as quickly as possible. For that reason, when an event is scheduled to take place at a certain time in our simulation, that has no bearing on when it happens in real time. A simulation of eight hours of real-world time shouldn't take eight hours to run; it should run as quickly as our program can determine the outcome. So, don't fall into the trap of trying to figure out how to pause your program for a certain number of milliseconds, or schedule functions to be called at a certain time; that's not the goal here.

There's one more thing to consider in your design, as well. Since we're measuring time at the fine grain of one millisecond, but since nothing will happen during most of the milliseconds in our simulation, you'll want to be sure that your simulation "skips ahead" whenever possible. In other words, if the current time is 5000 and there are no events scheduled to occur until time 6000, your simulation should immediately proceed to time 6000, rather than looping 1000 times — each time concluding that nothing needs to happen — in order to get there. This dramatically improves your program's ability to run sparse simulations over long stretches of simulation time (i.e., long simulations in which relatively little happens).

Design, organization, and testing requirements

Building on your prior background in designing and implementing Python programs (e.g., from ICS 32 or ICS H32), we'll leave many of the details of how to design your program unspecified; there are lots of good ways to approach the problem, so rather than require everyone to do exactly the same thing, we'll allow each of you to discover your own solution.

That said, we do have some requirements that need to be met, and that will inform at least some of your design choices. We will not be willing or able to pre-grade your work, and we won't generally be answering quesitons such as "If I make this decision choice, how will that affect my grade?" The requirements are listed below and they — along with what's specified in the section titled *Limitations* below and the general grading guidelines given on the Project Guide page — are definitive.

Your main module

You must have a Python module named `project1.py`, which provides a way to execute your program in whole; executing `project1.py` executes the program. Since you expect this module to be executed in this way, it would naturally need to have an `if __name__ == '__main__':` statement at the end of it, for reasons described in prior coursework. Note that the provided Git repository will already contain this file (and the `if __name__ == '__main__':` statement.

Aside from the requirement that there be a `project1.py` file that is your "main" module, you can change any of the provided code in any way you'd like to meet this project's requirements.

Modules other than the main module

This is a project that is large enough that it will benefit from being divided into separate modules, each focusing on one kind of functionality, as opposed to jamming all of it into a single file or, worse yet, a single function. As you no doubt learned in prerequisite coursework, when you divide a large program into smaller pieces that are relatively isolated from one another, your ability to work on a large-sized problem without losing control of its complexity dramatically increases. So, we'll expect to see your program divided into multiple modules, with each one focused on one kind of problem. We don't have specific requirements around exactly where you draw the line, but we do have the requirement that, in general, you "keep separate things separate."

If deciding what functionality belongs in which modules is territory that feels unfamiliar to you, the Modules notes from my ICS H32 course are a good primer on some of the ways you might think about that problem. Pay special attention to how terms like *high cohesion* and *low coupling* are defined there.

Working and testing incrementally

As you did in Project 0, you are required to do your work incrementally, to test it incrementally (i.e., as you write new functions, you'll be implementing unit tests for them), and to commit your work periodically into a Git repository, which you will be bundling and submitting to us.

As in Project 0, we don't have a specific requirement about how many commits you make, or how big a "feature" is, but your general goal is to commit when you've reached stable ground — a new feature is working, and you've tested it (ideally with unit tests). We'll expect to see a history of these kinds of incremental commits.

Test coverage

Your goal in this project is to reach the highest percentage of test coverage you can. Our goal here can't as easily be 100% as it was in Project 0 — mainly because there are aspects of what you're doing here that may not be testable with techniques that you've learned to date — but you'll nonetheless need to test whatever can be tested. Challenge yourself to cover everything that you can cover using the techniques that you've learned to date.

To that end, every function that lacks full coverage (i.e., for which at least one line or at least branch is left uncovered) will require a comment above it, briefly explaining why there is missing coverage. These comments won't ever need to be more than a sentence or two, but the objective is to reflect on why you weren't able to write unit tests that covered it. Doing that may actually help you to realize why you actually can test at least some of the code; by splitting a complex function into separate, smaller pieces, you can often take a function for which you know no technique to test it, and isolate the untestable portion of it in its own separate function or module. (You might also find that splitting up complexity makes an untractably large number of tests turn into a much smaller number of them instead. Automated testing is as much a design problem as it is a testing problem, and there's no better way to learn that lesson than to set a high test coverage goal and challenge yourself to meet it.)

We will be neither willing nor able to negotiate individually with you about explicitly what must be tested and what can be safely left untested, as this will depend heavily on the design decisions you're making. However, it's safe to say that there are relatively few things we're doing here for which no testing techniques exist, and we will be aware, while grading your work, of what aspects of this problem are likelier to be untestable (using techniques taught in this course or in prerequisite coursework) than others.

Avoiding patching and mocking

It's worth noting that Python provides a library named `unittest.mock`, which provides tools for doing what is sometimes called *patching* or *mocking*, with which we ask Python to temporarily replace a part of our implementation (or Python's standard library) — say, a function or a class — with a "fake" version that returns a result that we've predetermined. While this isn't a technique that's much use when running our program normally, it can open up some possibilities within unit tests, where we're testing parts of our program individually and (largely) out of context from the rest of the program. For example, we might replace a function of ours that reads input from a socket with one that returns a hard-coded result instead, so that our test doesn't rely on having connected a socket to a server somewhere.

The problem with this technique is that it relies on the specific details of one's implementation never changing. If, in a unit test, we're saying something to the effect of "Please replace the function `read_boo_age` with one that returns `13`," we've set ourselves up for this test to fail as soon as the underlying function changes. If we change the name of `read_boo_age`, or if we change the type of its result (e.g., we have it return a string instead of an integer, or a more complex object containing an integer), this test will fail.

In my experience, indiscriminate use of mocking techniques like this leads to unit tests that are extremely brittle, so that almost any change to a program's design causes numerous test failures, or so that the actual testing objective isn't met, because all we're really testing is the "fake" behavior that we've already hard-coded. In the hands of experienced users, mocking is a valid and useful technique, but, for it to be useful, it has to be seen as an "in case of emergency, break glass" technique. Unfortunately, it's also a convenient technique — it can be used to quickly solve a lot of problems, including those it's not particularly suitable for — which means that a lot of students (and a lot of professionals who haven't experienced the brittleness of a heavily mocked set of unit tests before) have a tendency to use the technique as a crutch: Every time they have trouble writing a test, they mock something, so that it becomes easier to write. (But, of course, when we write tests, the goal is not for the tests to be easy to write; it's for the tests to assert a truth about how our program should behave — ideally, one that will stand the test of time as our program evolves — then validate that it really is the truth.)

Consequently, the `unittest.mock` library is entirely off-limits in this project (and, indeed, in this entire course). Later in this course, we'll discuss alternative techniques that can be used to open up the range of what can be tested, while also introducing us a little more carefully to the tradeoffs we're making when we use techniques like that. But, for the time being, let's learn to stand on our own two feet, rather than learning on that crutch.

Sanity-checking your output

We are providing a tool that you can use to sanity check whether you've followed the basic requirements above. It will only give you a "passing" result in these circmustances.

- It's possible to run your program by executing a correctly named module (`project1.py`), spelled and capitalized correctly.
- Executing that module is enough to execute your program.
- Your program reads its input and generates character-for-character correct input for one simple test scenario.

It should be noted that there are many additional tests you'll be want to perform, and that there are many additional tests that we'll be using when we grade your project. The way to understand the sanity checker's output is to think of it this way: Just because the sanity checker says your program passes doesn't mean it's close to perfect, but if you cannot get the sanity checker to report that your program passes, it surely will not pass all of our automated tests (and may well fail all of them).

You'll find the sanity checker in your project directory, in a file named `project1_sanitycheck.py`. Run that program like you would any other, and it will report a result.

Limitations

You can use the Python standard library where appropriate in this project, and you can certainly depend on the code that we've provided in the Git bundle, but you will otherwise not be able to use code written by anyone else other than you. Notably, this includes third-party libraries (i.e., those that are not part of Python's standard library), which are strictly off-limits in this course except where specifically allowed. Colloquially, if we have to install something other than Python, Git, and PyCharm in order for your program to work, it's considered off-limits, unless specific permission is given in a particular project. This project offers no such permission.

Note, too, the discussion above about `unittest.mock`, which is off-limits in this course, even though it's part of Python's standard library. (We will not re-state this in every assignment, but do be aware of it.)

Preparing your submission

When you're ready to submit your work, run the provided `prepare_submission.py` script, as you did in Project 0, which will create a Git bundle from the Git repository in your project directory; that Git bundle will be your submission.

Verifying your bundle before submission

If you're feeling unsure of whether your bundle is complete and correct, you can verify it by creating a new PyCharm project from it, as you did in Project 0. (You'll want to create this project in a different directory from your project directory, so it's separate and isolated.) Afterward, you should see the files in their final form, and the **Git** tab in PyCharm should show your entire commit history. If so, you're in business; go ahead and submit your work.

Deliverables

Submit your `project1.bundle` file (and no others) to Canvas. There are a few rules to be aware of.

- When grading your program, we'll grade only the most recent submission. We will not negotiate about which submission will be graded, or, for example, grade multiple of your submissions and "take the highest score."
- When grading your program, we'll grade only the most recent commit on the `main` branch, except to the extent that we'll examine prior commits when evaluating your overall process. We will not negotiate about which commit will be graded, or, for example, grade multiple of your commits and "take the highest score."
- You're responsible for submitting the version of your project that you want graded prior to the deadline. Contacting us afterward and telling us that you accidentally submitted the wrong version will not be grounds for a resubmission under any circumstances.
- You're responsible for making a submission in order to receive credit, which means you'll want to be sure that you've remembered to submit your work and verified in Canvas that it's been received. A later claim of having forgotten to submit your work or having misremembered the due date will not be grounds for a resubmission under any circumstances.

- The determination of whether your work has been submitted before the deadline is the time it was submitted to Canvas. Neither timestamps on local copies of your files nor timestamps on commits in your Git repository or in other places (e.g., emails or online storage) are considered evidence of completion prior to the deadline under any circumstances.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link. Beyond the late work deadline described there, we will no longer accept submissions.

What do I do if Canvas adjusts my filename?

Canvas will sometimes modify your filenames when you submit them (e.g., by adding a numbering scheme like **-1** or a long sequence of hexadecimal digits to its name). In general, this is fine; as long as the file you submitted has the correct name prior to submission, we'll be able to obtain it with that same name, even if Canvas adjusts it.