ICS 33 Spring 2025
Project 2: *Learning to Fly*

Due date and time: *Wednesday, May 7, 11:59pm*
Final late work deadline: *Monday, May 12, 2:59am*

Git repository: *https://ics.uci.edu/~thornton/ics33/ProjectGuide/Project2/Project2.git*

Introduction

In lecture, we've explored the idea that it won't always be the case that our Python programs will operate only on objects stored in memory. We'll sometimes need data to be *persistent*, which is to say that we'll need it to remain available even if our program ends and we start it again later. Other times, we'll operate on an amount of data so large that it won't fit in our available memory — or, at the very least, we'll need a small enough percentage of the data at any given time that the cost of allocating so much memory to it would outweigh the benefit. Satisfying either of these requirements necessitates storing the data somewhere that outlives our program, which suggests that we could store it in one or more files instead; we know already that files live on, even after the program that created them has finished running, so they provide a great place to hold data for safe keeping.

As you've seen in prior coursework, though, this introduces a new set of problems to be solved, since file storage operates on a different set of principles than objects in memory in a Python program do. We need to figure out a way to take all of the objects we want to store and "flatten" them into one stream of text or bytes to be stored in a file, then to be able to turn that same information back into the original objects again. And, even if we can accomplish that, if the amount of data will be large, we'll need to figure out a way to solve that problem piecemeal, so we don't have to read an entire file every time we need one piece of information from it, or re-write the entire file every time we need to change one piece of information in it. These are difficult problems indeed, especially for non-experts.

Fortunately, these are such common problems that there are common solutions to them, with *databases* and *database management systems* (DBMSs) acting as the giants whose shoulders we can stand on. Provided that we can describe the shape of our data to a DBMS, it can efficiently automate the underlying file manipulation that occurs when we need to read a small amount of data from it or update something within it, so that we can issue a command that conceptually boils down to "Tell me the name of the user with the most followers" or "Associate a new mobile phone number with this patient's account" and get the necessary effect without needing to know the details of how files will be accessed or changed. As we saw in lecture, SQLite is a *relational* DBMS, which means that if we can describe our data in terms of tables and relationships between rows in those tables, and if we can issue it the necessary SQL statements, SQLite can take care of managing all of the file handling for us, even if the file is much larger than the amount of memory we have available. Furthermore, if we can describe constraints on that data, it can enforce them for us. Among its benefits is its availability in Python's standard library; if we have Python installed, we have SQLite, too, as well as a way to connect a Python program to a SQLite database. So, that makes it a great choice for our initial exploration of relational databases and SQL.

There is a tradeoff at work, though, in the sense that we now have a new problem to solve: When our Python program needs to fetch data from the database or make changes to it, it'll need to construct SQL statements and send them to SQLite, then interpret the result that SQLite sends back. In other words, since SQLite doesn't "speak Python," we'll have to do something to bridge the gap between the way our Python program manipulates data and the way SQLite does.

That tradeoff, ultimately, is the central focus of this project, in which you'll be implementing a program with a graphical user interface that allows a user to search and update some information in a SQLite database. We've provided a substantial starting point — notably, the entire graphical user interface is already implemented — so that you can stay focused on the important parts of the problem. But, of course, there's a tradeoff there, too: When you work on a project where a substantial amount of code is already in place, you'll need to understand enough about it that you can take advantage of what's there, while fitting your new work into it without having to rewrite all of it.

(A situation like this is a realistic analogue for joining an open source project or starting a new job; it's rarely the case that you'll be working in an area where nothing already exists, so setting one's fears and preferences aside and being able to become productive within an existing code base is an essential skill, though it can be quite daunting, even for relatively experienced people. If this project is your first experience with that, great! That's why we're doing it. If you stay on a path that leads to real-world software engineering, it won't be your last experience like this. Start early and give yourself some time to digest what's there, and you'll be in business.)

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.

Additionally, you'll need one more thing, which you won't find in that repository: a SQLite database containing information about airports from around the world, which your program will be querying and updating. Download the file linked below and store it somewhere, but make a note of where you put it, because you'll need to be able to find it later. (It's fine to store it in your PyCharm project directory, but you can put it anywhere else you'd like. However, regardless of where you put it, do not commit it into your Git repository, as it's not a part of your program and potentially changes every time you run your program.)

- airport.db

Understanding the provided database

This project will ask you to write a program that is primarily tasked with querying and updating a SQLite database that contains information about airports from around the world, mainly from the perspective of pilots flying into and out of them. As is often the case when you first start working in an area that's new to you — unless you're a trained pilot or a flight simulator enthusiast, it's likely that you know little or nothing about airports, runways, radio frequencies, and so on — your first order of business is familiarizing yourself with the problem domain. You don't need to become an immediate expert, but you definitely need to achieve at least a passing familiarity with the important concepts and the common terminology used to describe them. When you'll be using a database as part of your work, you'll also want to acquaint yourself with its *schema* (i.e., the tables, their columns, and the relationships between tables), which can be a great way to figure out what concepts are important and which terminology is common; understanding your data takes you a long way toward understanding a problem domain. If there are terms that are unfamiliar, you might even want to do some side research, so you understand a little bit of the context in which your work fits. (This process of gradual understanding has been necessary in every professional job I've ever started, since each one has been in an area of business very different from the previous ones. One of the great things about software skills is the number of areas in which they're applicable, but this means it's a lot likelier that switching jobs also means dramatically switching contexts.)

So, before you dive into writing any code, it's not a bad idea to take a look around the provided database, a task that PyCharm can help with, since it includes built-in tools for communicating with a SQL-based database like ours.

Connecting to the database in PyCharm

First, recall where you stored the `airport.db` file you downloaded previously, because now you're going to need it. Once you've figured that out, you're ready to connect to it within PyCharm. There are a few steps to follow to do that, which are complicated mainly because there are so many different ways that PyCharm is able to connect to databases, even though what we want to do is pretty simple.

- Open the PyCharm project you created previously, if it's not open already.
- Along the right-hand side of the PyCharm window, you'll see an icon labeled **Database**. Click that.
- That should reveal an area titled **Database**, which is mostly blank, but which has a few buttons along the top of it. Click the button labeled with a + (plus sign), which will drop down a menu, from which you should select **Data Source from Path**.
- A dialog titled **Select Database Location** will pop up, asking you to find the file that contains the database. Find your `airport.db` file, select it, and click **OK**.
- A dialog titled **New Data Source** will pop up.
    - The box labeled **Path:** should already be populated with the path to your database file, so you can leave that as-is.
    - In the dropdown labeled **Driver:**, make sure **SQLite** is selected, if it isn't already.
    - Click **OK**.
- A dialog titled **Data Sources and Drivers** will pop up, in which we'll need to configure how PyCharm will connect to the database.
    - Near the top-left corner, where there's a choice between **Data Sources** and **Drivers**, select **Drivers**, revealing a long list of DBMSs that PyCharm can connect to. Select **SQLite** in that list.
    - In the right-hand area of the window, click **General**, then click the + (plus sign) underneath **Driver Files**. That will reveal a menu, in which you should select **Provided Driver**, then **Xerial SQLiteJDBC**, then **3.43.0**. (If the only choices are slightly different from this version number, that's fine. Choose the most recent one that's at least **3.43.0** and you should be in business.)
    - Next, select **Data Sources** near the top-left corner (instead of **Drivers**).
    - Finally, click **OK** near the bottom-right corner of the dialog.

- In the **Database** area of the PyCharm window, you should now see `airport.db` listed. Expanding it should reveal a schema named **main**. Expanding **main** should reveal a list of **tables**. (You might need to right-click or Ctrl-click **main** and select **Refresh** to reveal this.) Expanding that should reveal the names of the tables in the database.

Now that your PyCharm project is connected to our database, we can now execute SQL statements against it. (The next time you close PyCharm and re-open the same project, this connection should still be available. If it ever disappears, the steps above should allow you to get it back again.)

Querying our database in the database console

Once you've connected to your database in PyCharm, another area within the PyCharm window titled **console** should have opened. If not, in the **Database** area, click the icon labeled **Jump to Query Console**, then select **Open Query Console** from the menu that pops up.

In that **console** area, you can type a SQL statement and execute it against the database. Type the SQL statement below into the **console** area, then click the **Execute** button near the top-left corner of the **console** area (or press Ctrl+Enter).

```
SELECT *
FROM airport
WHERE airport_ident = 'KSNA';
```

The **Services** area along the bottom edge of the PyCharm window will be displayed, if it wasn't already, in which you should see the result of your query: all of the columns from one row of a table named `airport`, describing information about Orange County Airport (which is a few miles from UCI).

That's all there is to it.

Exploring our database

From here, your best bet is to explore our database a bit. If you want to see its entire schema, the provided `schema.sql` file (which you'll find in your PyCharm project) shows all of the database's tables, including their names, the names and types of their columns, along with any other constraints (primary keys, foreign keys, and so on).

The data in our database came from community-sourced data provided by OurAirports. They provide the data as a collection of files in the *comma-separated values* (CSV) format, which I've converted into a SQLite database for our uses. Their data dictionary describes the meanings of the data they provide, which tracks pretty closely with the SQLite database that you've been provided, with each of their files having turned into one table in our database, and most of their columns appearing in our database with names that are the same (or, at least, pretty similar).

Let curiosity be your guide for a while. Don't aim to memorize everything you see; just aim for familiarity, as you would any time you're exploring new territory.

The program

Your program is required to be a `tkinter`-based graphical user interface that allows a user to interact with the information stored in the `airport.db` database. It doesn't only provide the ability to visualize the information already in the database; it also provides a means to update it, which means that the database is effectively both an input and an output of your program.

This arrangement, with a program and a database existing alongside each other, is not unusual in practice; the reason we have databases is often specifically so we can have persistent data that's updated gradually over time, with one or more programs used to update it along the way. In such cases, the program and the database are symbiotic; we can't run the program without the database, we can't (as easily) interact with the database without the program, and changes to the design of one will have a commensurate impact on the other. (In your case, the design of the database is a given, so you won't have to worry about changes in the database's design affecting the design or implementation of the program, but that's a realistic concern in real-world programs that evolve over long periods of time; schemas change, and programs have to change to accommodate those changes accordingly.)

Not all students in ICS 33 will necessarily have a prior background in writing graphical user interfaces using `tkinter` — it's covered in varying levels of depth in ICS 32 and ICS H32, depending on both instructor and quarter — so we've provided one in its entirety, though it's largely non-functional as provided, because the "engine" behind it — the code that interacts with the underlying database — is entirely missing and will need to be provided by you. We'll take a look at the details of how to do that a little later in the project write-up.

In terms of functionality, your program will have to meet the following basic requirements.

- Search for continents in the database, given either a continent code, a name, or both, displaying all of the continents that exactly match the given characteristics.
- Add a new continent to the database, by specifying the various data points that describe them (except their primary key).
- Update an existing continent in the database, changing any of the various data points that describe them (except their primary key).
- Search for a country in the database, given either its country code, its name, or both, displaying all of the countries that exactly match the given characteristics.
- Add a new country to the database, by specifying the various data points that describe them (except their primary key).
- Update an existing country in the database, changing any of the various data points that describe them (except their primary key).
- Search for a region (a part of a country) in the database, given either its region code, its local code, its name, or some combination of them, displaying all of the regions that exactly match the given characteristics..
- Add a new region to the database, by specifying the various data points that describe them (except their primary key).
- Update an existing region in the database, changing any of the various data points that describe them (except their primary key).

Notably, not all of the tables in the provided database are represented in the user interface, nor are they required to be, but the three tables that are represented should be sufficient to gain the necessary experience with how you can approach problems like this.

Persistence

It's worth noting that persistence is not just a concept we're discussing generally; it's a requirement. Changes made to the underlying data must be persistent, which is to say that if you make a change via the user interface, quit the program, start the program again, and open the same database, the changes made in the previous run of the program will still be present in the database. (That's one of the main reasons why we use databases, after all.)

Interlude: Organizing Python modules using packages

Because so much code is provided in this project, one of the things you'll need to do early on, before you start working on implementing the project, is to acquaint yourself with what's already there. Immediately, you'll see that the provided code is organized differently than you might have seen in your past work; it's made up of what are called *Python packages*. Since Python packages are likely to be new territory for you, let's stop briefly and talk about what problem they solve and the necessary details about their mechanics.

When we want to split up a collection of functions and classes, so that similar ones are grouped together and dissimilar ones are kept separate, we use *modules* to do that. Each module is written in a separate file (with a name ending in `.py`), and we use Python's `import` statement to allow the code in one module to make use of things written in other modules. We use naming conventions like leading underscores to indicate that some things are protected and shouldn't be used outside of their own module. All of that allows us to avoid writing large programs in a single file, which has all kinds of advantages.

But what do we do when there are so many modules that they become unwieldy to manage? The provided code spans fifteen or so modules, and it's not uncommon for one module to need to import things from multiple of them. That's a lot to complexity to track in our heads at once — more so for you, too, because you're new to this project and you weren't the original author of the provided code! — so it would be nice if there was a way of organizing all of these modules somehow, allowing us to achieve two goals.

- Grouping modules into directories, so that strongly-related modules are grouped together and less strongly-releated modules are kept separate. We want the same thing here that we wanted when we were grouping functions and classes into modules: *high cohesion* and *low coupling*. But now we want it from the perspective of modules instead of individual functions or classes; the modules in one directory are strongly related to each other, while the modules in different directories solve different kinds of problems. This way, when we're looking for the part of our program that's related to one broad kind of problem — a user interface, a network protocol, or whatever — we'll have an idea where to look.
- Providing a way for all of the modules in a directory to appear outwardly like they were a single module, so that we could import an entire directory's worth of modules at once, instead of importing each one independently. This way, when we're working on code in one part of a program, we won't have to remember every detail of which functions are defined in which modules in some other part of our program; we'll just need to know that "When I want something to do with the user interface, I want something in the `p2app/views` directory."

Python packages are meant to achieve those two goals. A Python package is a directory underneath the root directory of our project, in which there are multiple `.py` files. The relative path to that directory indicates the package's name. In the provided code for this project, for example, you'll find three packages named `p2app.engine`, `p2app.events`, and `p2app.views`. But there's a little more to the story than just the organization of files into a directory structure; the `import` statement gives us some tools to use packages more effectively, too.

When we want to import a single module from within a package in our project, we do so by specifying a sort of "path" to it. For example, if we wanted to import the module `main.py` from the `p2app/views` directory, we could do so this way.

```
import p2app.views.main
```

After having done that, we can access what's in that module in the usual way. For example, if that module contained a function named `foo`, we could call it like this.

```
p2app.views.main.foo()
```

When we want to import an entire package all at once, we do so by specifying the name of the directory. For example, if we wanted to import the entire `p2app.views` packages, we could write this.

```
import p2app.views
```

But what did we get? The `p2app.views` package contains several different modules, each containing functions and classes; so, which ones did we just import? The answer lies in one more detail: A directory that we want to be able to treat as a package like this needs to contain a file named `__init__.py`, whose job is to specify what the contents of the package are. Whatever it imports becomes what's imported when we say `import p2app.views`. For example, the contents of `p2app/views/__init__.py` in the provided code is a single line.

```
from .main import MainView      # The ".main" notation means "The module called main in
                                # the directory we're currently in."
```

Consequently, when we say `import p2app.views`, what we get is exactly one definition: the class `MainView` from `p2app/views/main.py`. Nothing else in any of the package's modules is made available this way, mainly because everything else in that package is meant to be used only within that package. (That doesn't stop us from importing those other modules individually and using what's in them — as usual, Python enforces few rules about access restrictions — but, for most uses, we'd simply import a package we need and be done with it.)

There's a little more to the story of packages in Python than this, but Python's own detailed documentation about its package feature tells it as least as well as I could; you can find that documentation at the link below.

- Python's documentation for the package feature

An overview of the program's design

Now that we've discussed how Python packages allow a large program to be organized, we can proceed with building an understanding of the design of the provided program. The program is organized into four major areas.

- A package named `p2app.views`, which defines a `tkinter`-based graphical user interface giving a user the ability to view and edit the information in the `airport.db` database. All of the necessary functionality is already in place, so you will not need to make any modifications to this package.
- A package named `p2app.engine`, in which you'll write the part of the program necessary to communicate with the `airport.db`, so that the provided graphical user interface can obtain and modify the information in that database. Almost all of the necessary code is missing from this package, so this is where you'll be working.
- A package named `p2app.events`, which provides the tools necessary to allow the `p2app.views` package to communicate with the `p2app.engine` package. When the user interface needs to engine to perform some operation, it uses events to do it; when the engine needs to communicate its results back to the user interface, it uses events to do it. All of the necessary functionality is already in place, so you will not need to make any modifications to this package.

- A "main" module named `project2.py` that initializes the necessary parts of the program, then launches the user interface. You will not need to make any modifications to this module.

How the user interface communicates with the engine

We've provided a complete implementation of this program except for the engine, whose role is to arbitrate access to the database, so that the user interface can be unaware of how any of that functionality is implemented. There are at least two good reasons why the program is designed that way.

- The usual benefits of designing software in a way that keeps separate things separate certainly apply here. When we handle different kinds of problems in different areas of a program — communicating only to the extent necessary and relying on the fewest number of details of the other areas as possible — we benefit by being able to think about the parts of our program in isolation, and by being able to change one without having to make cascading changes to the others.
- My goal was to provide a fully-implemented user interface, given that not everyone in this course would have a sufficient background in `tkinter` to be able to build it (and given that it was not the learning objective), while leaving the database-handling problem open-ended enough for each of you to implement it on your own and design it as you see fit. To make that possible, it was necessary to design this program in a way that allowed the user interface to know as little as possible about how you designed your solution.

Consequently, you can think of the communication between `p2app.views` and `p2app.engine` as being done entirely by sending *events* back and forth. When the user interface needs something done, `p2app.views` sends out an event. Subsequently, `p2app.engine` receives that event and processes it, then sends out one or more events indicating what the results are. Those events are, in turn, received by `p2app.views` and cause the user interface to change in some way.

While the idea sounds like new territory, our implementation of that idea is actually straightforward Python. When `p2app.views` sends out an event, the `Engine.process_event` method is called, and its `event` parameter will be the event that was sent. In return, `Engine.process_event` generates any resulting events, which are received by the user interface again. (By *generates*, I mean that `Engine.process_event` is a *generator function*, as we discussed in the Generators notes.) So, ultimately, the entire interaction between the two packages boils down to a sequence of calls to a generator function: the user interface calling it with an event, and the engine yielding its results (if any).

Like any communication protocol, though, we can't just send events willy-nilly and expect things to work. Instead, an agreement is necessary about which events are sent in which circumstances. You can expect that the user interface will hold up its end of the bargain and send the right events when the user clicks buttons, selects menu items, and so on, but its response to those clicks and selections will only be as good as the quality of the events sent back by the engine. What follows, then, is a protocol for how the two packages communicate events with each other. (Of course, the engine will also need to do things when receiving events; what's described below is how it communicates the necessary results back to the user interface.)

Situation	*Event sent by `p2app.views`*	*Event(s) sent back by `p2app.engine`*
Application-level events		
User quits the application	`QuitInitiatedEvent`	`EndApplicationEvent`
User opens a database file	`OpenDatabaseEvent`	• `DatabaseOpenedEvent`, when the database was opened successfully • `DatabaseOpenFailedEvent`, when opening the database failed, with a user-friendly error message
User closes the currently-open database file	`CloseDatabaseEvent`	`DatabaseClosedEvent`
Continent-related events		
User initiates a search for continents	`StartContinentSearchEvent`	• One `ContinentSearchResultEvent` for each continent found in the search. • If no continents were found, no events are returned.

User loads a continent from the database to edit it	`LoadContinentEvent`	`ContinentLoadedEvent`, containing the loaded information about the continent
User saves a new continent into the database	`SaveNewContinentEvent`	• If saving the continent succeeded, `ContinentSavedEvent`, containing the complete information about the saved continent • If saving the continent failed, `SaveContinentFailedEvent` with a user-friendly error message.
User saves a modified continent into the database	`SaveContinentEvent`	• If saving the continent succeeded, `ContinentSavedEvent`, containing the complete information about the saved continent • If saving the continent failed, `SaveContinentFailedEvent` with a user-friendly error message.
Country-related events		
User initiates a search for countries	`StartCountrySearchEvent`	• One `CountrySearchResultEvent` for each country found in the search. • If no countries were found, no events are returned.
User loads a country from the database to edit it	`LoadCountryEvent`	`CountryLoadedEvent`, containing the loaded information about the country

User saves a new country into the database	`SaveNewCountryEvent`	• If saving the country succeeded, `CountrySavedEvent`, containing the complete information about the saved country • If saving the country failed, `SaveCountryFailedEvent` with a user-friendly error message.
User saves a modified country into the database	`SaveCountryEvent`	• If saving the country succeeded, `CountrySavedEvent`, containing the complete information about the saved country • If saving the country failed, `SaveCountryFailedEvent` with a user-friendly error message.
Region-related events		
User initiates a search for regions	`StartRegionSearchEvent`	• One `RegionSearchResultEvent` for each region found in the search. • If no regions were found, no events are returned.
User loads a region from the database to edit it	`LoadRegionEvent`	`RegionLoadedEvent`, containing the loaded information about the region
User saves a new region into the database	`SaveNewRegionEvent`	• If saving the region succeeded, `RegionSavedEvent`, containing the complete information about the saved region • If saving the region failed, `SaveRegionFailedEvent` with a user-friendly error message.
User saves a modified region into the	`SaveRegionEvent`	• If saving the region succeeded, `RegionSavedEvent`, containing the complete information about the saved region

database		• If saving the region failed, `SaveRegionFailedEvent` with a user-friendly error message.

In cases in which errors occur and there is no event specifically defined for it (e.g., when loading a continent fails), the engine should yield one `ErrorEvent`.

The various event types listed here are defined in the `p2events` package; refer to the Python modules in that package for details like the names and types of their attributes.

Debugging the events being sent and received

When you launch the program, you'll find that it has a **Debug** menu on which there is one selection: **Show Events**. Ordinarily, this feature is disabled; selecting that menu choice will cause it to be enabled. The next time you look at the **Debug** menu, you'll find a checkmark next to **Show Events**. Selecting it again disables the feature once again.

When the **Show Events** feature is enabled, all of the events sent from `p2app.views` and `p2app.engine` and sent back from `p2app.engine` to `p2app.views` are logged to the standard output. While your program is running, you'll find them in the **Run** area in PyCharm, where any other standard output would normally appear.

How our work fits into a present-day context

It's worth noting that the design you see here bears some similarities to the design you'd see in a present-day Internet-based application, such as a web or mobile application, even though ours is built on a somewhat more primitive collection of technologies that runs entirely as a single program on one computer. In an Internet-based application, it's not uncommon to have the same division of labor that we have in our application, though an Internet-based application would divide that labor amongst separate programs running on separate computers, with computer networks used to communicate between them.

- One program where the user interface is drawn and where the user interacts with it. This might run within a web browser, or it might be a native application for the target platform (say, Windows, macOS, iOS, or Android). In

present-day parlance, this program is often called the *front end* of such an application, and is conceptually equivalent to our `p2app.views` package.
- One Internet-facing program to which the user interface makes requests when it needs to obtain or change information. This is likely running on a server somewhere, and could be written in any programming language or for any operating system, as long as there's a way for the front end to communicate with it; some kind of socket-based communication, as you might have done in ICS 32 or ICS H32, would likely be used for that purpose. It's common to call this program the *back end* of the application, which would be conceptually equivalent to our `p2app.engine` package (i.e., the part we're asking you to write), with our `p2app.events` package approximating the means of communication between the front and back ends (which nowadays would often be done using HTTP requests and responses).
- The database would likely also be a separate program, likely running on a different server from the back end — and, for reasons of security, probably isolated in a way that makes it inaccessible to the open Internet, meaning that the back end can communicate with it, but arbitrary machines on the Internet can't. The back end would communicate with the database similarly to how we're doing it, by sending SQL statements to it and interpreting the results; behind the scenes, though, those SQL statements and their results would be sent across a computer network, and libraries likely implementing most or all of those details for us. In that sense, our use of `sqlite3` is very much like what we would do if we were talking with a separate database server, except that we connect to a file instead of a host and a port.

So, all in all, you won't have written an Internet-based application when you're done with this project, but you'll have experienced a design that's surprisingly similar in its arrangement.

Design, organization, and testing requirements

There is no need to change any code except in the `p2app.engine` package, which is almost entirely empty. You can change any of the provided code in any way you'd like to meet this project's requirements, though we strongly recommend leaving all of the provided code except the `p2app.engine` package as-is, as we will not be willing to spend the time to help you debug your changes if you decide to re-design the parts that have been provided in their entirety; the user interface is not the learning objective of this project, so you'll want to approach changes in that area with caution (and you'll need to accept the burden of researching and debugging those changes on your own, if you decide to go that route).

You'll need to design the code in the `p2app.engine` package like any other large-scale things we write, with single-purpose functions and divided into separate modules where appropriate.

Your main module

You must have a Python module named `project2.py`, which provides a way to execute your program in whole; executing `project2.py` executes the program. In fact, we've provided one already, and there should be no need to change it, so your best bet is to leave it as-is unless you have a good reason to change it.

Working and testing incrementally

As you did in previous projects, you are required to do your work incrementally, to test it incrementally, and to commit your work periodically into a Git repository, which you will be bundling and submitting to us.

Unlike in previous projects, however, the territory in which we'll be working is much less amenable to the kinds of unit tests that we've been writing. The necessary techniques to work around these limitations are beyond the scope of this course, so there is no requirement that you write any unit tests in this project; we are not requiring it and we will not be deducting points when it's missing. You certainly can write unit tests, though, and you might even want to write them for areas of your code that don't interact directly with the database, so that you know they work the way you expect. You'll likely find, though, that unit testing the parts of your code that interact with the database will not be amenable to unit testing in the ways you've learned previously, since the tests would rely on a particular database set up in a particular way; the kinds of tools that automate this sort of testing are beyond the scope of this course.

Of course, it should go without saying that you'll still need to test your work, and you'll still be best off doing that incrementally. The difference, though, is that you'll likely have to do the majority of that job manually — in the graphical user interface, or from the Python shell — this time around, and there will be nothing for you to submit that verifies that you've done it. Still, if you haven't tested your work thoroughly, there's a pretty good chance you'll submit a program that fails in ways you didn't expect, which will affect how we test it after submission and, of course, will negatively impact your score on this project.

Using techniques not yet taught in the course

If you look through the provided code, particularly in the `p2app.views` package, you'll find that it uses some techniques (e.g., inheritance) that have not yet been taught in the course. You will not need those techniques in your own implementation, though you aren't forbidden from using them; still, you won't need to use Python techniques that haven't either been taught in the course, in prerequisite coursework, or (in the case of packages) in this project write-up.

Respecting data integrity

When we discussed Databases in class, we talked about *foreign keys*, which are a mechanism that relational databases use to allow us to store a value in one row that describes some other row in the database. Relational database management systems will generally have a mechanism to enforce the integrity of foreign key constraints, which is to say that they can prevent us from storing foreign keys that relate to non-existent rows, or from changing rows that will violate foreign key relationships in other places; SQLite is no exception to this, though SQLite has this feature turned off by default, so we'll need to turn it on.

When your program connects to the `airport.db` database, it can do so using a technique similar to what we described in the Databases notes. However, by default, SQLite will ignore foreign key constraints, so your program will need to execute one SQL statement right after it connects.

```
PRAGMA foreign_keys = ON;
```

That statement will take effect for the duration of the connection, but this will have to be done every time a connection is established; otherwise, SQLite will ignore foreign key constraints.

(A "pragma" in a programming language is generally a way to configure the behavior of a language processor in some way. In this case, we're asking SQLite to respect foreign key constraints.)

How NULLs in the database relate to the user interface

When presenting or editing a value in the user interface that corresponds to a database column that allows `NULL` values to be stored in it, we have three related design problems to consider.

- How do we display `NULL` values in the user interface?
- How does a user enter a `NULL` value in the user interface?
- How well can we differentiate between values that are `NULL` and those that have some other "empty" value (e.g., an empty string)?

As a simplifying measure, in our program, we'll approach these problems by using `schema.sql` as a starting point, which is to say that the rules we'll follow will be determined by the types and constraints on the corresponding database columns.

- For `TEXT` columns that have a `NOT NULL` constraint, the user would enter an empty string by leaving the corresponding text entry widget blank.
- For `TEXT` columns that allow `NULL` values, when the user enters an empty string, it is considered to be `NULL`. (One consequence of this is that there's no way for the user to enter an empty string, a limitation we'll accept to keep things simple.)
- For columns whose types are other than `TEXT` that allow `NULL` values, empty values in the user interface correspond to `NULL` values in the database.

Avoiding SQL injection

The main objective of our engine is to act as an intermediary between a Python program and a SQLite database. The `sqlite3` database is doing some of the heavy lifting for us, because, for example, we don't have to be aware of how the database is organized within the file in which it's stored, nor the details of what happens behind the scenes when SQL statements are executed. But we nonetheless bear some of burden ourselves, because SQLite is only as good as the SQL statements we ask it to execute on our behalf, and it's our engine that's determining what those SQL statements are.

A critically important issue to consider is where those SQL statements come from. In particular, at least some of what your engine does is driven by input that came from the user interface; when a user wants to search for a country with a particular name, that name is part of the SQL statement we'll need to use to find it. That name might contain characters that have special meaning in SQL — or even SQL code, if a user is particularly nefarious — and so it'll be up to us to mitigate problems like this. There are two things we want to be sure of.

- Legal user input should not cause a failure just because it contained characters that happen to have special meaning in SQL. (For example, it's legal to search for a region whose name has a `'` character in it, even though `'` is also a

character used in SQL to indicate the beginning or the end of a string literal.)
- In no way should user input be allowed to cause things to happen in our database other than the functionality we're providing. (In other words, we need our program not to fall prey to a SQL injection attack.)

If this concept sounds unfamiliar to you, your best bet is to check out the section of the Databases notes titled *Avoiding injection attacks*, which discusses this problem — and what we might do about it — in some detail. In short, it's not a difficult problem to solve once we know it's there, but it's easy to fall into the trap if you're not aware of it. We'll be expecting you, in this project, not to fall into the trap.

Limitations

You can use the Python standard library where appropriate in this project, 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); 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.

Preparing your submission

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

Keeping the database out of your submission

The database is not meant to be included in your submission, since we'll be testing your submission with our own database(s) that, while they contain the same tables as the original database, may not contain precisely the same data — particularly if you've made updates to yours along the way. This means a couple of things.

- You will not want to commit changes to the database to your Git repository.
- If it indeed has not been committed to your Git repository, it will not be included in your bundle, and, consequently, will not be part of your submission.

Note, too, that the starting point includes a file named `.gitignore` that should have the effect of automatically preventing you from committing the database to your Git repository. This is intentional, as we do not want you to submit it.

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 `project2.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.