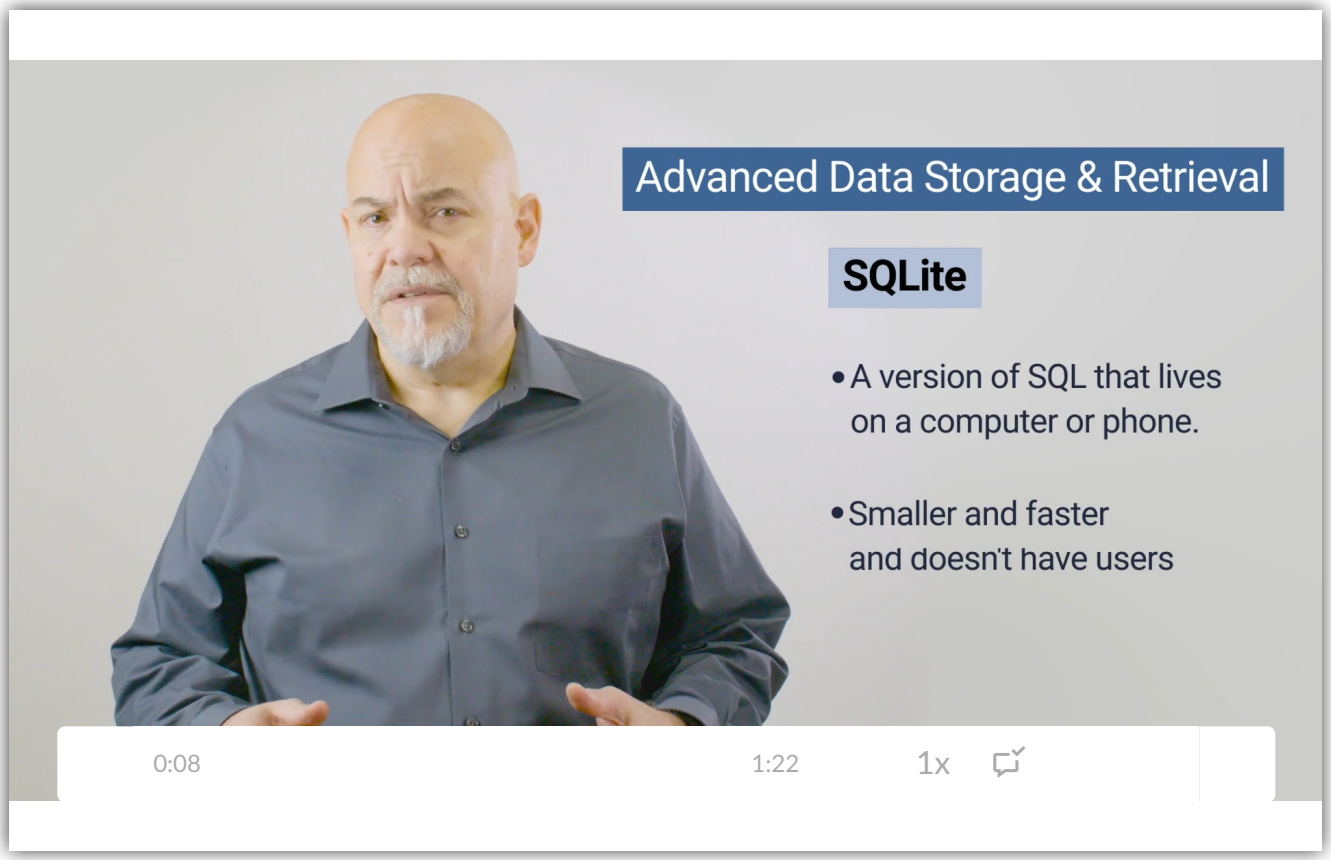
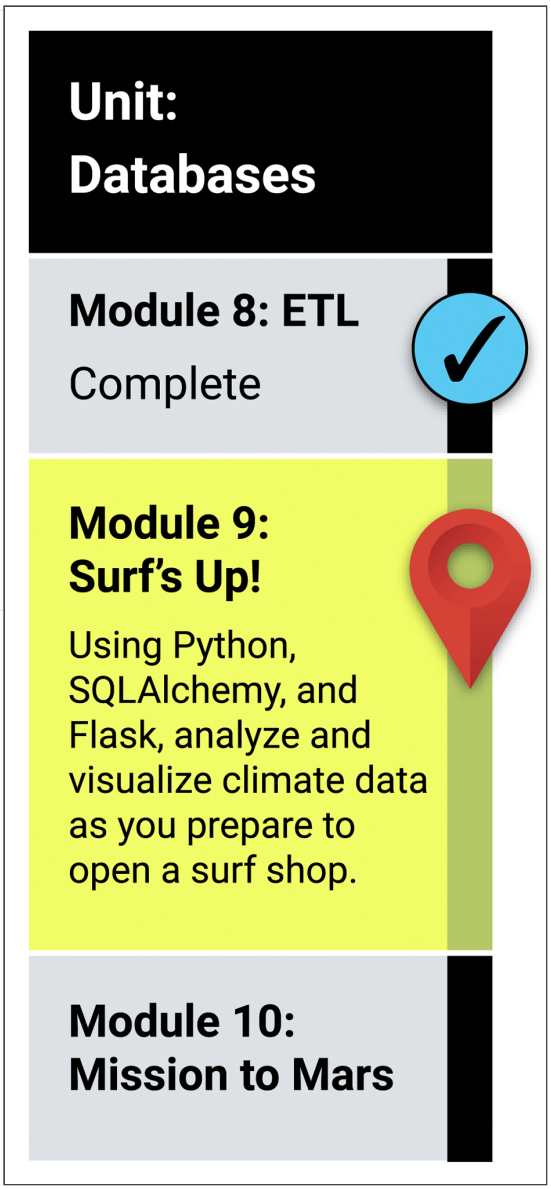
9.0.1: Exploring Weather Data



9.0.2: Module 9 Roadmap

Looking Ahead

In this module, you'll spend time with new tools such as SQLite, SQLAlchemy, and Flask to build on your knowledge of SQL database structures and querying methods. You'll also write and execute Python code in a Jupyter notebook and create graphs using Python.



What You Will Learn

By the end of this module, you will be able to:

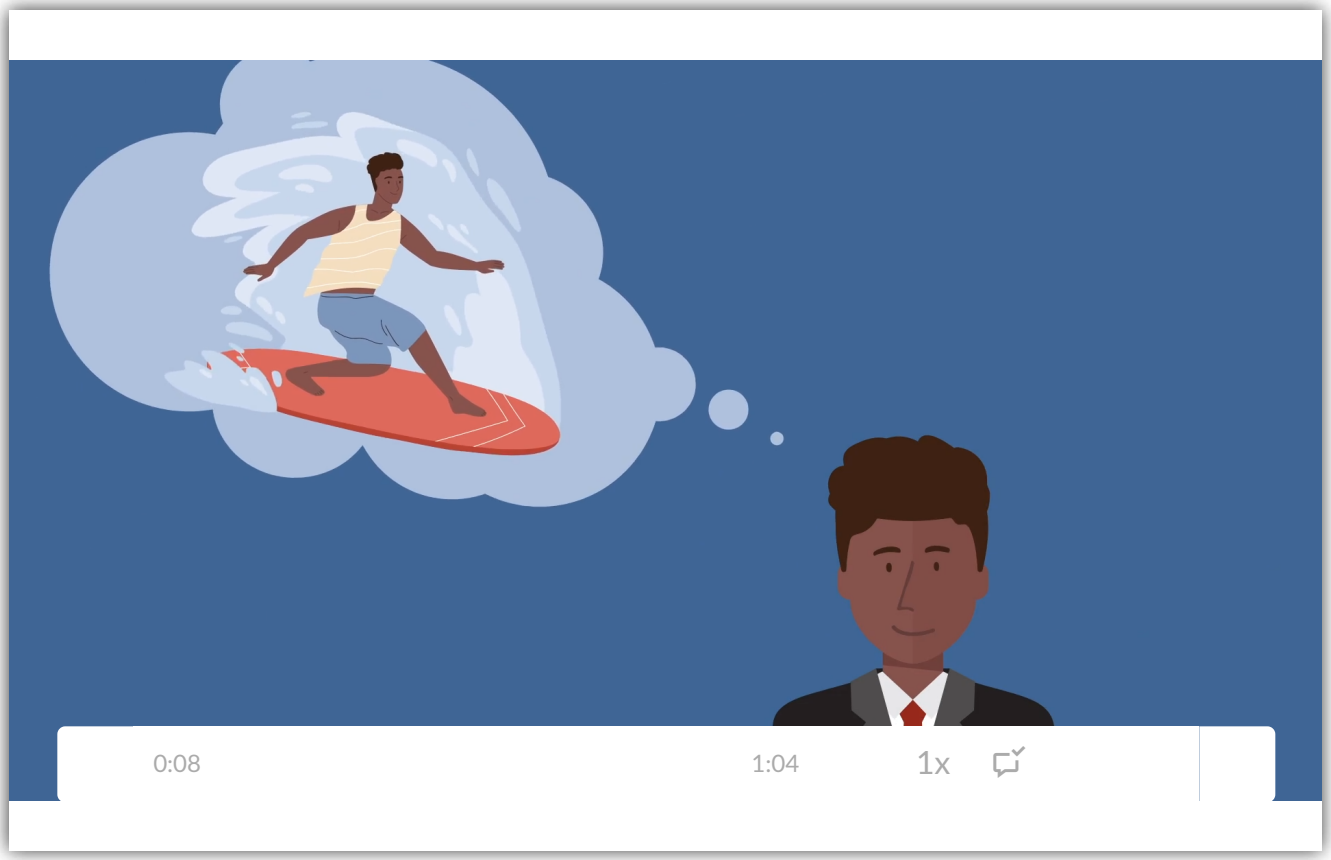
- Explain the structures, interactions, and types of data of a provided dataset.
- Differentiate between SQLite and PostgreSQL databases.
- Use SQLAlchemy to connect to and query a SQLite database.
- Use statistics like minimum, maximum, and average to analyze data.
- Design a Flask application using data.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to SQLite, SQLAlchemy, and Flask (15 minutes)
- SQLite and SQLAlchemy (1 hour)
- Precipitation Analysis (1 hour)
- Weather Station Analysis (2 hours)
- Getting Acquainted with Flask (2 hours)
- Build a Climate App Using Flask (2 hours)
- Application (5 hours)

9.0.3: Investing in Waves and Ice Cream



9.1.1: Download the Weather Data

Your next meeting with W. Avy is coming up soon, so you are eager to start analyzing! Impressing him is your one-way ticket to Oahu. You know that your analysis can only be as good as your data, so the first step is to make sure you have the data downloaded correctly so that you can start exploring it.

Here's what we'll need to get started:

- **Jupyter notebook file:** we will continue to use Jupyter Notebook for our weather analysis. The name of the file is `climate_analysis.ipynb`. This file will have all of the structure to help you get started on your analysis. Let's get started by downloading the notebook.

- **SQLite database:** W. Avy has stored the weather data in a SQLite database. All SQLite databases are **flat files**, which means that they don't have relationships that connect the data to anything else. As a result, flat files can be stored locally, which will help us move more quickly through the analysis.

Now download the SQL database and dataset to your class folder by clicking the links below:

climate_analysis.ipynb
(https://courses.bootcampspot.com/courses/138/files/14670/download?wrap=1)

hawaii.sqlite (https://courses.bootcampspot.com/courses/138/files/14669/download?wrap=1)

Before getting started, take a moment to stretch and practice those surfing moves!

9.1.2: Prepare Your Tools

Once your database is downloaded, it's a good idea to check and make sure you have the right tools in place. Without the tools to analyze it, the data won't do us much good.

We'll kick our analysis off by checking on three tools: Jupyter Notebook, VS Code, and GitHub.

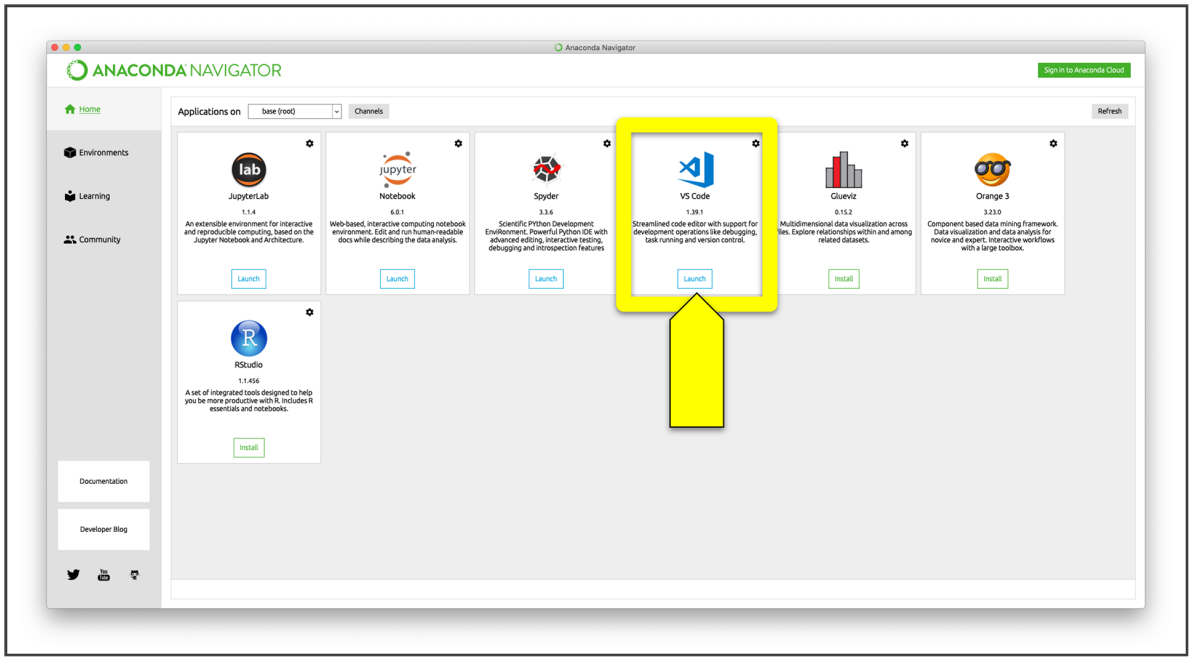
Jupyter Notebook

The first tool we'll check on is Jupyter Notebook. As you already know, Jupyter notebook files are great for sharing code. Since Jupyter Notebook is run in a browser, you will be able to easily share your analysis with W. Avy. All investors will need to do is download the file and run the code you provide.

Check that you have Jupyter Notebook installed by viewing your Applications folder. Open Terminal (on Mac) or Command Prompt (on Windows) and run the command `jupyter --version`. If a version number is returned (any version number is fine) you're good to go. If a version number is not returned, you'll need to reinstall the program.

VS Code

We'll also be using VS Code in this module to create our Flask application. To make sure you have it installed, go to the Anaconda Navigator application and check the list of applications within the Navigator. If you do not see it in the list, go ahead and install it.



GitHub

We'll be using GitHub to store our code, so you should create a new repository for this project. Let's name it "surfs_up." Remember to commit your code early and often!

> **Rewind**
>
> To create a new repository on GitHub:
>
> 1. Give the repository the name "surfs_up."
> 2. (Optional) Add a brief description of the repository and what type of programming software you are using for this project.
> 3. Make the repository public.
> 4. Click "Initialize this repository with a README." Each GitHub repo has a README file that serves as a kind of homepage for the repository. This

> is where you can add a description of your project. We will add a description at the end of this module.
>
> 5. Click "Add .gitignore" and type “Python.”
> 6. Click the green “Create repository” box.
>
> After clicking “Create repository," you’ll be on the homepage of your repository. Once you are back, you will want to copy the HTTP link to the repo from the “Clone or download” green button.

To clone the “surfs_up” repository to your computer, run the following command. Make sure to add your HTTP github repo link. **Hint:** before running this command, make sure you’re in the local directory that you want your code to go in.

```
git clone < github link >
```

Now make sure that the clone was successful. To do this, navigate through your local file system and check that the surfs_up folder is in the correct place.

Nice work! Now you’re ready to import the dependencies.

9.1.3: Import Dependencies

As soon as you finish downloading the SQLite database and checking that your tools are ready to go, your phone buzzes—it's a text from W. Avy! It reads:

"Aloha! Thinking about the surf shop—excited to see what you come up with. Also, thinking longer term, if the shop on Oahu does well, we could expand to other islands."

You fire back a quick response: "Yes! I'll keep the code documented so we won't have to duplicate efforts in the future."

Turning back to your computer, you're ready to take the next step: import the dependencies. You make a mental note to comment out your code.

Now that we have the data downloaded and our tools set up, we can import all the dependencies we'll need. Some will be new, others you might have used before. You will be writing your code in the provided Jupyter notebook file, `climate_analysis.ipynb`. Go ahead and open this Jupyter notebook using your command line.

> **Rewind**
>
> **Dependencies** are previously written snippets of code that we can then use in our code. Dependencies save us tons of time because we don't have to write every line of code ourselves. Instead, we just import the dependency.
>
> Dependencies can be provided by companies or other programmers or analysts, or they can be from code you wrote previously.

Matplotlib Dependencies

The first dependency we will need to import is Matplotlib, as we'll need to graph the results of our analysis to show investors.

Matplotlib's dependency contains code that allows you to plot data. There are many different kinds of plots you can create; for this project, we'll use the "fivethirtyeight" style. This style essentially tries to replicate the style of the graphs from FiveThirtyEight.com. (There are other style types too—if a different style catches your eye, feel free to use it!)

> **Note**
>
> For more information about fivethirtyeight style, see this **FiveThirtyEight style sheet (https://matplotlib.org/3.1.1/gallery/style_sheets/fivethirtyeight.html)** .

Start by running the following code. This will import style from Matplotlib.

```
from matplotlib import style
```

Next, we'll add the specific style we want, fivethirtyeight. Add this line to your code:

```
style.use('fivethirtyeight')
```

Now we need to add the pyplot module, a dependency that provides us with a MATLAB-like plotting framework. Go ahead and add this to your code.

```
import matplotlib.pyplot as plt
```

Next, we'll add NumPy and Pandas dependencies.

NumPy and Pandas Dependencies

We will need to use a few standard dependencies for our code. Go ahead and import NumPy and Pandas dependencies with the following code:

```
import numpy as np
import pandas as pd
```

Next, we'll import datetime.

Datetime Dependencies

We'll use datetime in this module because we'll need to calculate some data points that have to do with dates. To import datetime, run the following code:

```
import datetime as dt
```

Good work! Finally, we'll import a few dependencies from SQLAlchemy.

Import SQLAlchemy Dependencies

We know we want to query a SQLite database, and SQLAlchemy is the best tool to do that. So we'll need to import dependencies from SQLAlchemy.

We can start by adding the SQLAlchemy dependency, but then we will also add the dependencies for `automap`, `session`, `create_engine`, and `func`. These dependencies will help us set up a simple database that we'll use later on.

Add the following to your code:

```
import sqlalchemy
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import session
from sqlalchemy import create_engine, func
```

Great work on getting your dependencies imported into your code! Give yourself a pat on the back: You have just saved yourself a ton of time by importing dependencies instead of coding everything by hand. You are that much closer to days in the waves!

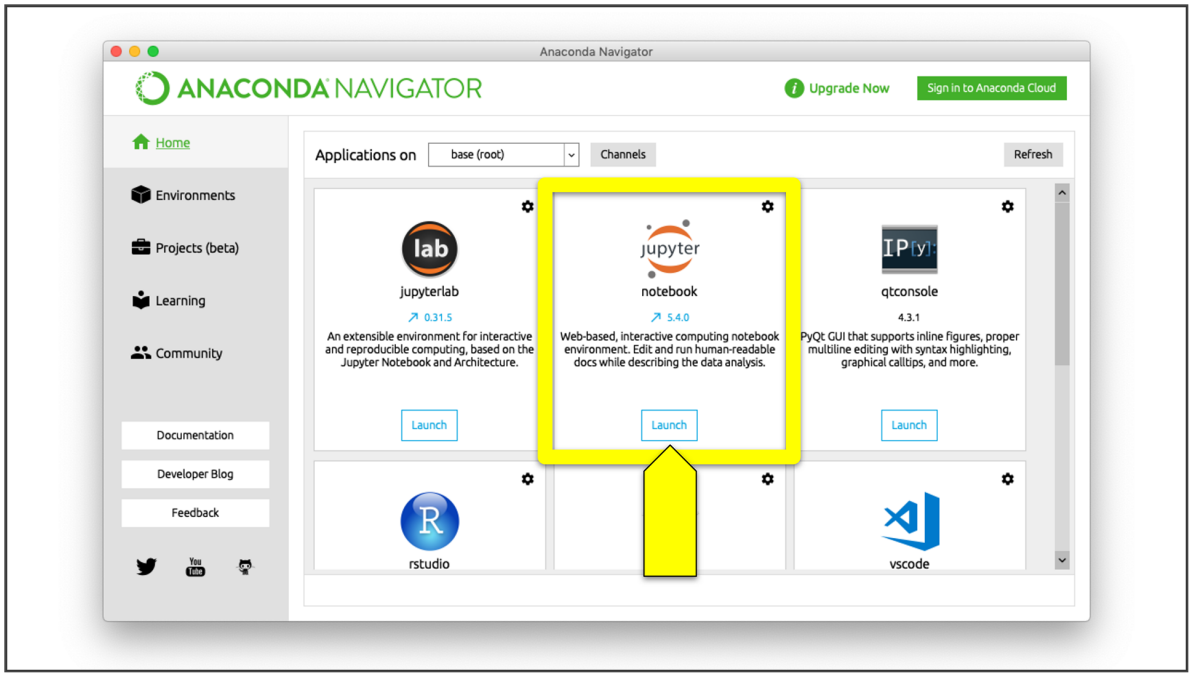
Now let's open the Jupyter notebook file so that we can begin exploring SQLite and SQLAlchemy.

Open Starter Jupyter Notebook

Previously, we've opened Jupyter notebook files via the command line. In this module, we'll use a different method: Anaconda Navigator. There is no right or wrong way to open the file; this is just another option at your disposal.

The Jupyter Notebook file, `climate_analysis.ipynb`, is already downloaded in the surfs_up folder. Navigate to this folder.

Next, open the Anaconda Navigator application, which you should find in the Applications folder. Once you've opened Anaconda-Navigator, find the Jupyter Notebook application icon and click it. This icon should be the top center item, as shown below:



When you click the Anaconda Navigator icon, a new command line window will open, followed by a webpage showing the files on your computer. Navigate through this file structure to find the surfs_up folder where you saved your `climate_analysis.ipynb` file. Click the Jupyter Notebook file to open it.

Nice work! Now we're ready to explore SQLite.

9.1.4: Getting Started with SQLite

We'll be using SQLite to store the weather data that W. Avy shared with us and that we'll need for our analysis.

You know there is a slight possibility that W. Avy will ask you to duplicate this process in the future (to open other shops, or if Oahu doesn't turn out to be a good fit). So, as you work through the process of getting started with SQLite, you make a note to make sure to take your time and really understand what you are doing so that you can do it again in the future.

SQLite provides a quick way to setup a database engine without requiring a server. It's essentially a flat file, but with most of the major capabilities of an SQL database—just like how a "lite" version of ice cream is basically ice cream, but with less fat.

You can compare SQLite databases to a CSV or Excel file: each SQLite database can have one or more tables with columns and rows, and it is stored as a file on your computer. The key difference between SQLite databases and a CSV or Excel file is that we can write queries for it.

SQLite Advantages

While there are a few specific use cases for SQLite, we'll be focusing on how it can be beneficial to you and where you might get the most value from it. The main advantages are:

- **It's local.** One of the core advantages of SQLite is that it allows you to create databases locally on your computer to support testing and easy prototyping. This is beneficial, because if you want to test something out and you need a database, it's not always the most convenient to set up a SQL database server just to try something out.
- **There's an app for that.** Another advantage of SQLite databases are that they can be used on a mobile phone app. Most mobile phone games will use an SQLite database to store certain information about you or your players statistics. While we won't be creating a mobile app in this module, it's still helpful to understand the full context.

SQLite Disadvantages

SQLite also has a couple of disadvantages, however. They are:

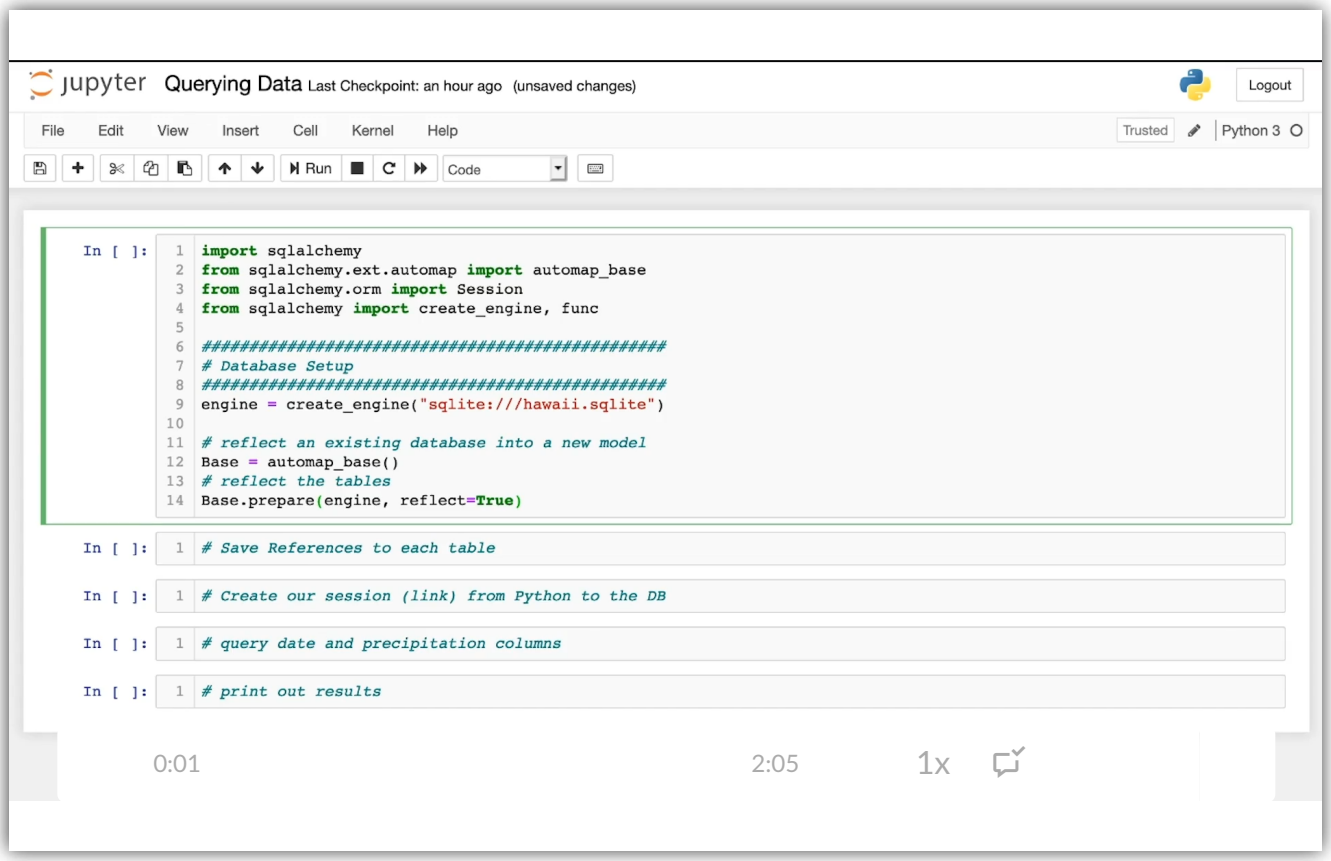
- **It's local.** If you've used a MYSQL database before, you might have noticed that you can have multiple users access the database. With SQLite, there are no users. SQL is local: stored on one computer or phone. So, only that computer or phone will have access.
- **There are fewer security features:** one other disadvantage to be aware of is that SQLite doesn't have as many security features as a traditional SQL database. While it's not something specifically to be concerned with for this module, just keep that in mind as you create other databases later on.

Good work with SQLite! Now let's move on to SQLAlchemy.

9.1.5: Getting Started with SQLAlchemy

In order to connect to the SQLite database, we'll use SQLAlchemy. SQLAlchemy will help us easily connect to our database where we'll store the weather data.

SQLAlchemy is one of the primary technologies we will be looking at in this module. It's extremely helpful for querying databases. If you are familiar with PostgreSQL, you'll see there are many similarities to the process. Watch the following video to learn more about SQLAlchemy.



Let's get started by looking at the SQLAlchemy Object Relational Mapper (ORM).

SQLAlchemy ORM

One of the primary features of SQLAlchemy is the **Object Relational Mapper,** which is commonly referred to as ORM. ORM allows you to create classes in your code that can be mapped to specific tables in a given database. This allows us to create a special type of system called a **decoupled system**.

To understand ORMs and decoupled systems, consider the following scenario. Suppose you are cleaning out the garage, and you find a bunch of wires or ropes that are all knotted together. We would call this a **tightly coupled system:** all of the different ropes are connected to each other, so if we go to grab just one, the whole mess comes along with it. What the ORM does for us is untangle—or decouple—all of those ropes, so we can use just one of them at a time. When we pick one up, we won't pick up the whole knot; or, if one element breaks, it doesn't affect any of the other cords.

Generally speaking, the less coupling in our code, the better. If there are a bunch of relationships between all of your coding components and one of them breaks, everything breaks.

The ORM helps us keep our systems decoupled. We'll get into more specific details about how we can keep our code decoupled, but for now, just remember that your references will be to classes in your code instead of specific tables in the database, and that we'll be able to influence each class independently.

SQLAlchemy Create Engine

Another really great feature of SQLAlchemy is the `create engine` function. This function's primary purpose is to set up the ability to query a SQLite database. After all, data just sitting in a database that we can't access does us no good.

In order to connect to our SQLite database, we need to use the `create_engine()` function. This function doesn't actually connect to our database; it just prepares the database file to be connected to later on.

This function will typically have one parameter, which is the location of the SQLite database file. Try this function by adding the following line to your code.

```
engine = create_engine("sqlite:///hawaii.sqlite")
```

We've got our engine created—good work! Next we're going to reflect our existing database into a new model with the `automap_base() function`. Reflecting a database into a new model essentially means to transfer the contents of the database into a different structure of data.

SQLAlchemy Automap Base

Automap Base creates a base class for an automap schema in SQLAlchemy. Basically, it sets up a foundation for us to build on in SQLAlchemy, and by adding it to our code, it will help the rest of our code to function properly.

In order for your code to function properly, you will need to add this line to your code:

```
Base = automap_base()
```

Good work! You don't need to run your code quite yet—we'll do that in the next section. Let's move onto reflecting our database tables into our code.

SQLAlchemy Reflect Tables

Now that we've gotten our environment set up for SQLAlchemy, we can reflect our tables with the `prepare()` function. By adding this code, we'll reflect the schema of our SQLite tables into our code and create mappings.

> **Important**
> Remember when we talked about keeping our code decoupled? When we **reflect** tables, we create classes that help keep our code separate. This ensures that our code is separated such that if other classes or systems want to interact with it, they can interact with only specific subsets of data instead of the whole dataset.

Add the following code to reflect the schema from the tables to our code:

```
Base.prepare(engine, reflect=True)
```

Now that we've reflected our database tables, we can check out the classes we'll be creating with Automap.

View Classes Found by Automap

Once we have added the `base.prepare() function`, we should confirm that the Automap was able to find all of the data in the SQLite database. We will double-check this by using `Base.classes.keys()`. This code references the classes that were mapped in each table.

- `Base.classes` gives us access to all the classes.
- `keys()` references all the names of the classes.

> **Important**
> Previously, we talked about decoupled systems in the SQLAlchemy ORM. This directly relates to the classes we have created here. These classes

help keep our data separate, or decoupled. Keep in mind that our data is no longer stored in tables, but rather in classes. The code we will run below enables us to essentially copy, or reflect, our data into different classes instead of database tables.

Run the following code:

```
Base.classes.keys()
```

What is the outcome of the code after you run `Base.classes.keys()` ?

- `['precipitation','station]`
- `['station']`
- `['measurement']`
- `['measurement','station']`

Check Answer

Finish ▸

Now that we've viewed all of our classes, we can create references to each table.

Save References to Each Table

In order to reference a specific class, we use `Base.classes.<class name>`. For example, if we wanted to reference the station class, we would use `Base.classes.station`.

Since it can be rather cumbersome to type `Base.classes` every time we want to reference the measurement or station classes, we can give the classes new

variable names. In this case, we will create new references for our `Measurement` class and `Station` class. Add these new variables to your code:

```
Measurement = Base.classes.measurement
Station = Base.classes.station
```

Now that we have our references saved to some new variables, let's work on creating a session link our database.

Create Session Link to the Database

Let's create a session link to our database with our code. First, we'll use an SQLAlchemy Session to query our database. Our session essentially allows us to query for data.

```
session = Session(engine)
```

> ### Add, Commit, Push
>
> Now that we have most of the setup complete, it's time to add, commit, and push your code to GitHub. Remember to follow these steps:
>
> 1. `git add < FILE NAME>`
> 2. `git commit -m "< ADD COMMIT MESSAGE HERE>"`
> 3. `git push origin master`

9.2.1: Retrieve the Precipitation Data

Your dependencies are added, you've connected the SQLite database, and your code is appropriately documented. It’s time to start the analysis.

Now, let’s think about precipitation. W. Avy is concerned about the amount of precipitation on Oahu. There needs to be enough rain to keep everything green, but not so much that you lose out on that ideal surfing and ice cream weather.

You know that you can set W. Avy’s mind at ease by analyzing precipitation levels and showing him the cold, hard, data that backs up Oahu as the perfect place to surf. You have the last 12 months of precipitation data already loaded into your SQLite database, so you are ready to go.

W. Avy supplied you with the data he wants us to use and has asked you to look at a full year of data. When deciding how to parse the data, you remember that his favorite day is August 23, 2017 because it’s the anniversary of the first time he ever went surfing and had ice cream on the same day. So, you decide to start the analysis there.

In the weather database, let's calculate the date one year from August 23, 2017. We'll be creating a variable called `prev_year` and using the datetime dependency that we imported previously.

The datetime dependency has a function called `dt.date()`, which specifies the date in the following format: year, month, day.

Find the Date One Year Ago

Add the most recent date, August 23, 2017, with the following code:

```
prev_year = dt.date(2017, 8, 23)
```

This code specifies the most recent date, but we want to calculate the date one year back. To do this, add the `dt.timedelta()` function to the previous line of code. This function allows us to trace back a certain number of days. In this case, we want to go back 365 days. Go ahead and add the `dt.timedelta()` function to your code.

```
prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)
```

Now that we've got our date from the previous year (August 23, 2016), let's retrieve the amount of precipitation that was recorded, or the **precipitation score**.

Retrieve the Precipitation Scores

We'll begin by creating a variable to store the results of the query we'll write. This variable will be called `results`:

```
results = []
```

This code defines our new variable. Next, let's add our session that we created earlier so that we can query our database. For this we'll use the `session.query()` function, which is how we'll begin all of our queries in SQLAlchemy. From a bird's-eye view, this is how we query a SQLite database using Python.

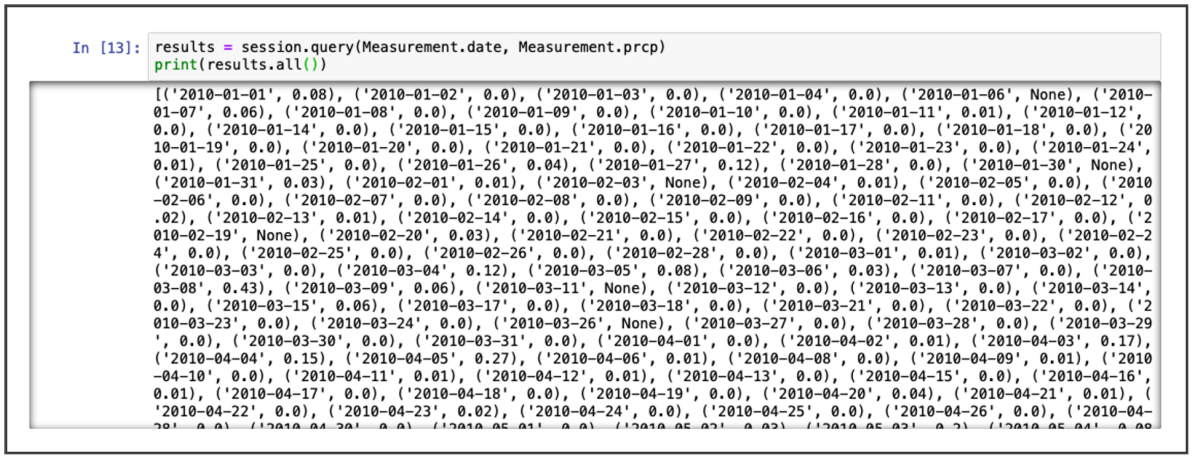
The `session.query()` function for this query will take two parameters. We will reference the Measurement table using `Measurement.date` and `Measurement.prcp`. Add the following to your code:

```
results = session.query(Measurement.date, Measurement.prcp)
```

Let's give this a shot and run the code. You might notice that there isn't anything returned. Let's go ahead and add a new line here. This will print everything that is returned in the query.

```
print(results.all())
```

Here's what you can expect to see.



We still have a few aspects to add to our query, but we'll get to that shortly.

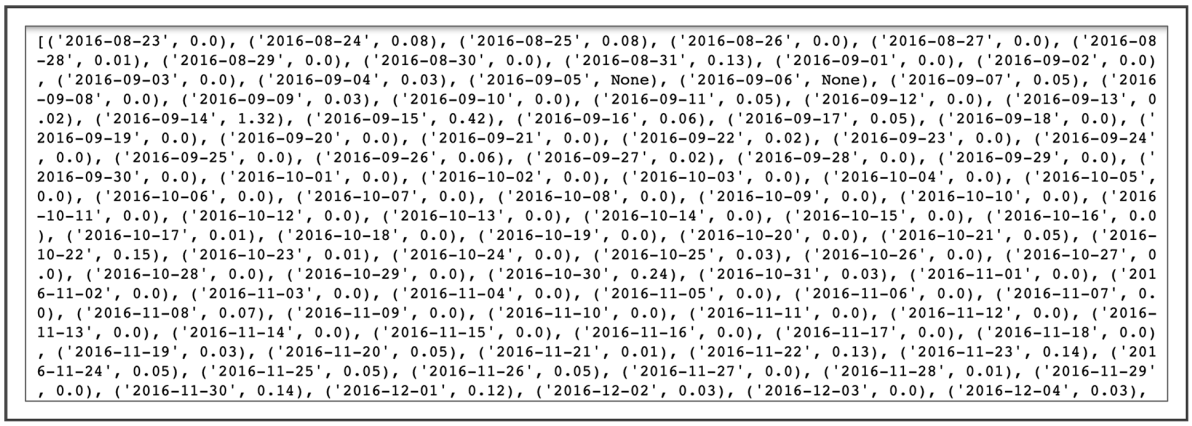
Since we only want to see the most recent data, we need to filter out all of the data that is older than a year from the last record date. We'll use the `filter()` function to filter out the data we don't need. Add the `filter()` function to the existing query.

```
results = session.query(Measurement.date, Measurement.prcp).filter(Measur
```

One last thing: we'll add a function that extracts all of the results from our query and put them in a list. To do this, add `.all()` to the end of our existing query. All said and done, your query should look something like this:

```
results = session.query(Measurement.date, Measurement.prcp).filter(Measur
```

Let's run this code. We'll print the results in order to ensure that we're getting output. Add `print(results)` after the last line of code. Here's what your results should look like:



```
[('2016-08-23', 0.0), ('2016-08-24', 0.08), ('2016-08-25', 0.08), ('2016-08-26', 0.0), ('2016-08-27', 0.0), ('2016-08
-28', 0.01), ('2016-08-29', 0.0), ('2016-08-30', 0.0), ('2016-08-31', 0.13), ('2016-09-01', 0.0), ('2016-09-02', 0.0)
, ('2016-09-03', 0.0), ('2016-09-04', 0.03), ('2016-09-05', None), ('2016-09-06', None), ('2016-09-07', 0.05), ('2016
-09-08', 0.0), ('2016-09-09', 0.03), ('2016-09-10', 0.0), ('2016-09-11', 0.05), ('2016-09-12', 0.0), ('2016-09-13', 0
.02), ('2016-09-14', 1.32), ('2016-09-15', 0.42), ('2016-09-16', 0.06), ('2016-09-17', 0.05), ('2016-09-18', 0.0), ('
2016-09-19', 0.0), ('2016-09-20', 0.0), ('2016-09-21', 0.0), ('2016-09-22', 0.02), ('2016-09-23', 0.0), ('2016-09-24'
, 0.0), ('2016-09-25', 0.0), ('2016-09-26', 0.06), ('2016-09-27', 0.02), ('2016-09-28', 0.0), ('2016-09-29', 0.0), ('
2016-09-30', 0.0), ('2016-10-01', 0.0), ('2016-10-02', 0.0), ('2016-10-03', 0.0), ('2016-10-04', 0.0), ('2016-10-05',
0.0), ('2016-10-06', 0.0), ('2016-10-07', 0.0), ('2016-10-08', 0.0), ('2016-10-09', 0.0), ('2016-10-10', 0.0), ('2016
-10-11', 0.0), ('2016-10-12', 0.0), ('2016-10-13', 0.0), ('2016-10-14', 0.0), ('2016-10-15', 0.0), ('2016-10-16', 0.0
), ('2016-10-17', 0.01), ('2016-10-18', 0.0), ('2016-10-19', 0.0), ('2016-10-20', 0.0), ('2016-10-21', 0.05), ('2016-
10-22', 0.15), ('2016-10-23', 0.01), ('2016-10-24', 0.0), ('2016-10-25', 0.03), ('2016-10-26', 0.0), ('2016-10-27', 0
.0), ('2016-10-28', 0.0), ('2016-10-29', 0.0), ('2016-10-30', 0.24), ('2016-10-31', 0.03), ('2016-11-01', 0.0), ('201
6-11-02', 0.0), ('2016-11-03', 0.0), ('2016-11-04', 0.0), ('2016-11-05', 0.0), ('2016-11-06', 0.0), ('2016-11-07', 0.
0), ('2016-11-08', 0.07), ('2016-11-09', 0.0), ('2016-11-10', 0.0), ('2016-11-11', 0.0), ('2016-11-12', 0.0), ('2016-
11-13', 0.0), ('2016-11-14', 0.0), ('2016-11-15', 0.0), ('2016-11-16', 0.0), ('2016-11-17', 0.0), ('2016-11-18', 0.0)
, ('2016-11-19', 0.03), ('2016-11-20', 0.05), ('2016-11-21', 0.01), ('2016-11-22', 0.13), ('2016-11-23', 0.14), ('201
6-11-24', 0.05), ('2016-11-25', 0.05), ('2016-11-26', 0.05), ('2016-11-27', 0.0), ('2016-11-28', 0.01), ('2016-11-29'
, 0.0), ('2016-11-30', 0.14), ('2016-12-01', 0.12), ('2016-12-02', 0.03), ('2016-12-03', 0.0), ('2016-12-04', 0.03),
```

> **Important**
>
> When you're handling a data analysis problem, printing your results is one of the most important tasks you can do. This can help you debug your code and ensure that you're getting all the data that you are expecting.
>
> You should print your results frequently so that you can make sure you're getting the data that you expect. Otherwise, you might spend hours working on code only to discover you're way off track.

Good work! Now that we've created the query, let's save it so that we can easily access it later, when we dive into Flask. Let's walk through how to do that now.

9.2.2: Save Query Results

In order to have easy access to the results, we need to put them in a DataFrame. This will help solidify a repeatable process and make it easier to run this analysis again—for example, when we need to figure out where to open our second surf and ice cream shop.

We have our weather results saved in a variable. In order to access it in the future, we'll save it to a Python Pandas DataFrame. We'll start by creating a DataFrame variable, `df`, which we can use to save our query results.

In order to save our results as a DataFrame, we need to provide our results variable as one parameter and specify the column names as our second parameter. To do this, we'll add the following line to our code:

```
df = pd.DataFrame(results, columns=['date','precipitation'])
```

This saves our query results in two columns, `date` and `precipitation`. Now we can manipulate the results however we would like. There are many functions you can use to manipulate how DataFrames look, but we'll start with using the `set_index()` function.

Use the set_index() Function

The `set_index()` function can be a little tricky, but let's jump in. For example, let's say our DataFrame looks something like below. Note that our data will not look like this exactly—this is just an example.

```
    date        precipitation
0   08/23/2017  1
1   08/22/2017  2
2   08/21/2017  1
3   08/20/2017  1
```

The first column is auto-generated and contains the row number. However, we want the index column to be the date column, so we'll need to get rid of those row numbers.

To do this, set the index to the date column. This will make the date column the first column.

For this project, we're going to experiment and write over our original DataFrame. By doing this, we can reduce the complexity of our code and use fewer variables.

> **Rewind**
> We can use the variable `inplace` to specify whether or not we want to create a new DataFrame.

Let's go ahead and use the same DataFrame. By setting `inplace=True`, we're saying that we do not want to create a new DataFrame with the modified specifications. If we set it to "False," then we would create a new DataFrame. Add the following to your code:

```
df.set_index(df['date'], inplace=True)
```

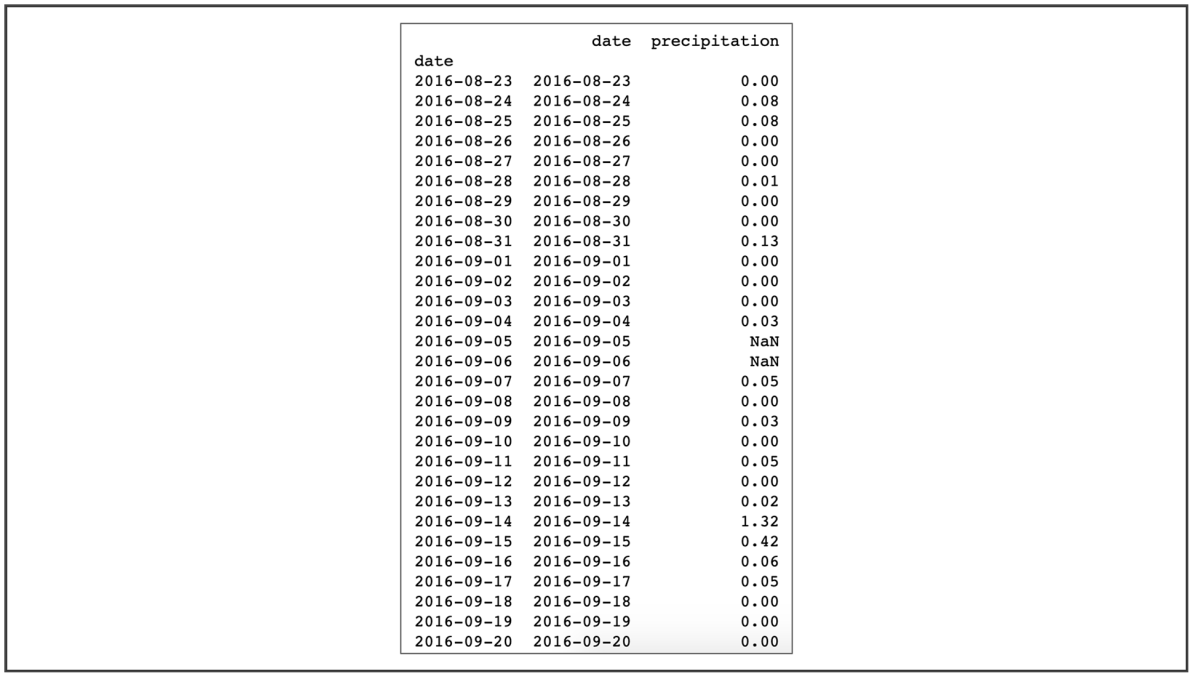
Run the code. Then print the DataFrame with and without the index so that you can see the difference.

Print the DataFrame With and Without the Index

To print a DataFrame with the index, use the following code:

```
print(df)
```

Here's what the printed DataFrame with the index should look like:



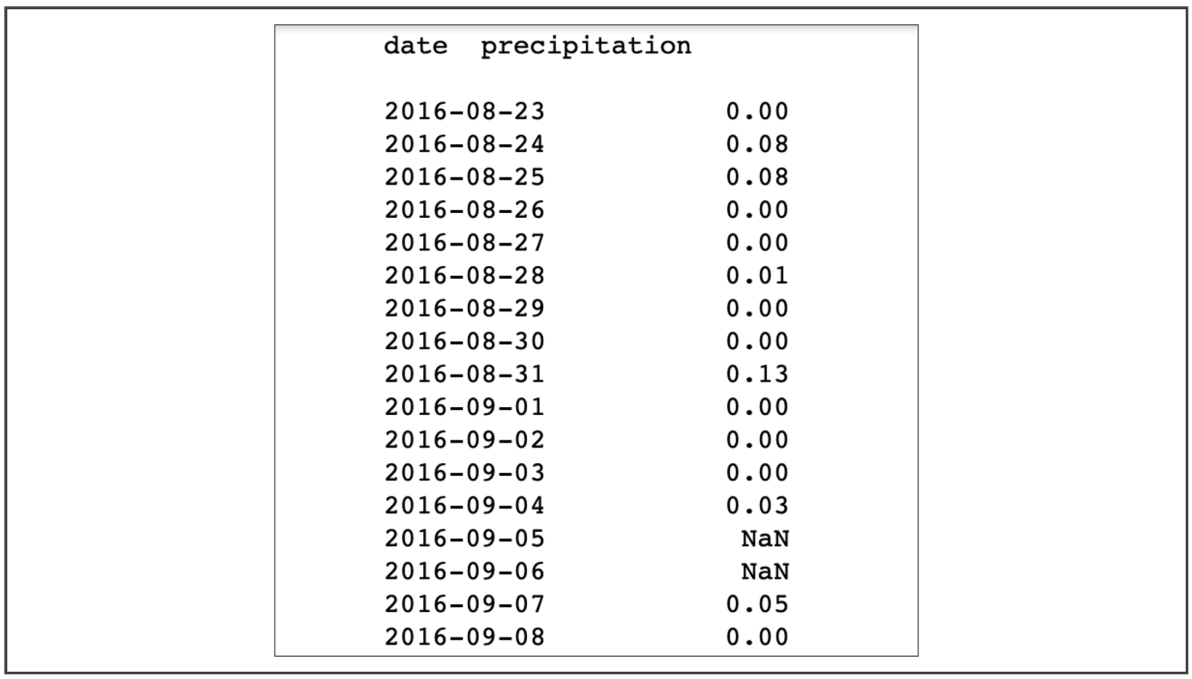
	date	precipitation
date		
2016-08-23	2016-08-23	0.00
2016-08-24	2016-08-24	0.08
2016-08-25	2016-08-25	0.08
2016-08-26	2016-08-26	0.00
2016-08-27	2016-08-27	0.00
2016-08-28	2016-08-28	0.01
2016-08-29	2016-08-29	0.00
2016-08-30	2016-08-30	0.00
2016-08-31	2016-08-31	0.13
2016-09-01	2016-09-01	0.00
2016-09-02	2016-09-02	0.00
2016-09-03	2016-09-03	0.00
2016-09-04	2016-09-04	0.03
2016-09-05	2016-09-05	NaN
2016-09-06	2016-09-06	NaN
2016-09-07	2016-09-07	0.05
2016-09-08	2016-09-08	0.00
2016-09-09	2016-09-09	0.03
2016-09-10	2016-09-10	0.00
2016-09-11	2016-09-11	0.05
2016-09-12	2016-09-12	0.00
2016-09-13	2016-09-13	0.02
2016-09-14	2016-09-14	1.32
2016-09-15	2016-09-15	0.42
2016-09-16	2016-09-16	0.06
2016-09-17	2016-09-17	0.05
2016-09-18	2016-09-18	0.00
2016-09-19	2016-09-19	0.00
2016-09-20	2016-09-20	0.00

Great work! Our DataFrame looks good. However, because we are using the date as the index, the DataFrame has two date columns, which is confusing. So we'll print the DataFrame without the index so we can see just the date and precipitation.

For this task, we'll need to use a slightly different print statement. First we'll convert the DataFrame to strings, and then we'll set our index to "False." This will allow us to print the DataFrame without the index. Add the following to your code:

```
print(df.to_string(index=False))
```

When you run the code, your results should look like the image below. Note that this image shows only the first several lines—your results will likely be longer.



date	precipitation
2016-08-23	0.00
2016-08-24	0.08
2016-08-25	0.08
2016-08-26	0.00
2016-08-27	0.00
2016-08-28	0.01
2016-08-29	0.00
2016-08-30	0.00
2016-08-31	0.13
2016-09-01	0.00
2016-09-02	0.00
2016-09-03	0.00
2016-09-04	0.03
2016-09-05	NaN
2016-09-06	NaN
2016-09-07	0.05
2016-09-08	0.00

Good work! Now that our DataFrame is set up, we can sort the results within the DataFrame. If you scroll through the data, you may notice that many dates are not in chronological order. Our next order of business is to fix that.

9.2.3: Sort the DataFrame

W. Avy doesn't just want to see a list of data; he wants to understand trends in the data. You know just the thing that will help; you're going to create a plot of precipitation scores in chronological order. Rather than simply showing him whether it rained on a given day, you'll show him how much it rained and if it was raining on the previous or subsequent days as well. Remember, your goal isn't just to crunch the numbers—it's to help W. Avy really feel good about his investment.

We're going to sort the values by date using the `sort_index() function`. Since we set our index to the date column previously, we can use our new index to sort our results. Add the following line to your code:

```
df = df.sort_index()
```

Now we're going to print our sorted list.

What code would you run to print the sorted list without the index?

- `print(df.to_string)`
- `print(df.to_string(index=False)`
- `print(df)`
- `print(df.to_string(index=True)`

Check Answer

Finish ▸

```
print(df.to_string(index=False))
```

Run the code. The results should look like the following:

```
      date  precipitation

2016-08-23           0.00
2016-08-23            NaN
2016-08-23           1.79
2016-08-23           0.05
2016-08-23           0.15
2016-08-23           0.70
2016-08-23           0.02
2016-08-24           0.08
2016-08-24           2.15
2016-08-24           2.28
2016-08-24            NaN
2016-08-24           1.45
2016-08-24           1.22
2016-08-24           2.15
2016-08-25           0.08
2016-08-25           0.00
2016-08-25           0.21
```

All of the dates are now in order, which is exactly what we were hoping to accomplish. Scroll through the results to make sure.

NOTE

> It’s critical to keep your brain checked-in while you are writing code and solving problems. Otherwise, simple errors may slip past you!

Good work sorting the DataFrame. Now you can plot the results to really impress W. Avy!

9.2.4: Plot the Data

Early one morning, you wake up to a text from W. Avy: “Aloha! Just having some ice cream and thinking about the waves! How is the analysis coming along? Hoping to get your results in time for our quarterly board meeting, but haven’t seen anything from you yet.”

You’ve made great progress on your analysis, so you text back confidently, “Aloha! Putting the final touches on plotting the data today, so you should have my initial findings by this evening.” Then you stretch, sigh, and get to work plotting your results. You are one day closer to Oahu!

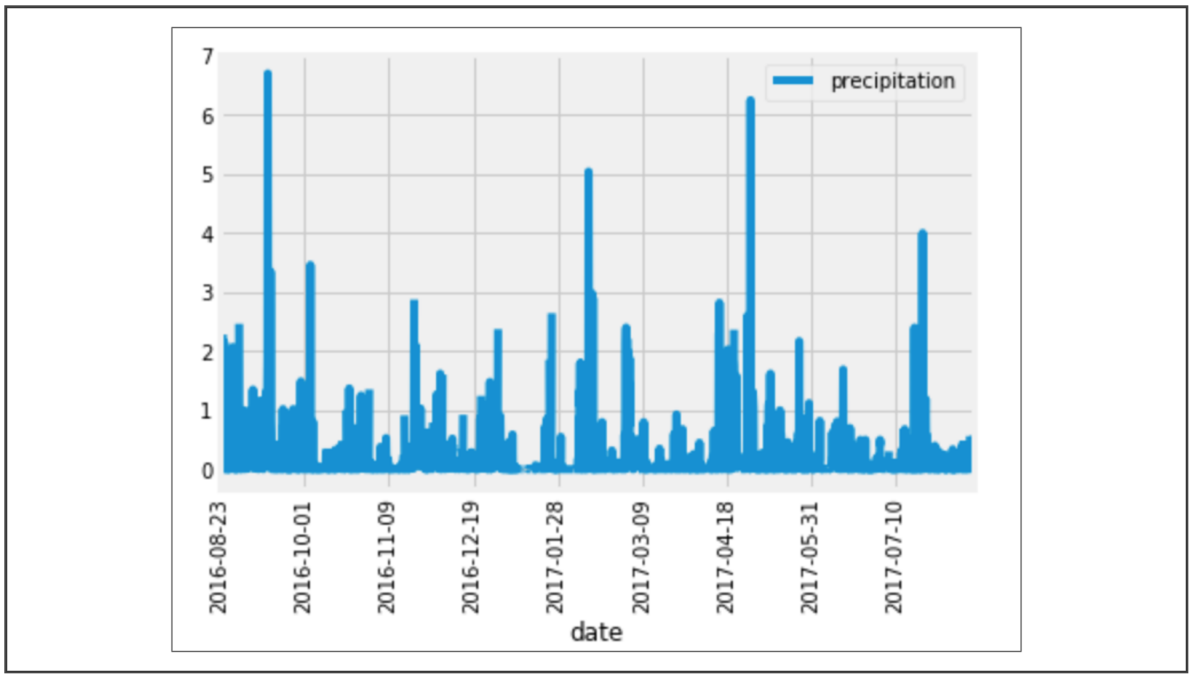
Remember, your goal is to provide W. Avy with insight into the weather patterns of a specific location on Oahu where you would like to build your shop. One way to provide this insight is with a visualization—we’ll plot the results of our precipitation analysis using Matplotlib.

> **Rewind**
>
> We’ve covered plotting before, using Matplotlib. We’ll be using Matplotlib for this project as well. Plotting is essentially displaying your data in a visual way. There are many different types of plots, but we’ll use a select few for this analysis.

Since our DataFrame is represented as the variable `df`, we can use the `df.plot()` function. Type the following code:

```
df.plot()
```

Run this code. Your plot should look similar to the following:



Along the x-axis are the dates from our dataset, and the y-axis is the total amount of precipitation for each day. While this data shows all of the station observations, we are interested in determining weather trends. One trend we can observe based on this plot is that some months have higher amounts of precipitation than others. Awesome—this observation confirms that the plot is useful. W. Avy is going to love it!

Next, we want to create a summary of a few statistics, and then we can send W. Avy an email with our initial findings. Be sure to tell W. Avy that this plot shows the total precipitation per day.

9.2.5: Generate the Summary

In addition to the plot we just made, we want to make sure to provide W. Avy with some solid statistical analysis—such as the mean, standard deviation, minimum, and maximum. He needs hard results if he's going to invest his money.

We're getting close to being able to deliver some of our findings to W. Avy.

> **Rewind**
> Here's a refresher on some key concepts in statistics:
> - **Mean:** the average, which you can find by adding up all the numbers in a dataset and dividing by the number of numbers.
> - **Variance:** how far a set of numbers is from the average.
> - **Standard deviation:** a measure of how spread out numbers in a dataset are; the square root of the variance.
> - **Minimum:** the smallest number in a dataset.
> - **Maximum:** the largest number in a dataset.
> - **Percentiles:** where the number is in relation to the rest of the set of data.
> - **Count:** the total number of numbers or items in a dataset.

Fortunately, Pandas helps us with these calculations. We'll use the `describe()` function to calculate the mean, minimum, maximum, standard deviation, and

percentiles. Add the following to your code:

```
df.describe()
```

Now run everything you've got so far. The summary should look something like this:

	precipitation
count	2021.000000
mean	0.177279
std	0.461190
min	0.000000
25%	0.000000
50%	0.020000
75%	0.130000
max	6.700000

This data gives us a summary of different statistics for the amount of precipitation in a year. The count is the number of times precipitation was observed. The other statistics are the precipitation amounts for each station for each day.

> ### Add, Commit, Push
> You've accomplished a lot so far! Now is a good time to update or add your code to your GitHub repository.

Now that we've completed our precipitation analysis, we can share it with W. Avy and move on to the station analysis.

9.3.1: Find the Number of Stations

When you sent your initial findings to W. Avy yesterday, you were a little nervous to hear his reaction—which is only natural when the stakes are so high! But to your delight and relief, W. Avy is ecstatic. His text reads, “This is great! It’s clear from your analysis that Oahu is a great location for the new surf shop. We’re almost ready to switch out our suit and ties for some sandals! My only question is, how many stations are being used to collect this information? Is it possible that we don’t have enough data collection stations for this information to be valid?”

Thankfully, you know you can run a query on the SQLite database to find this information quickly. You respond, “Glad the analysis is helping you with your decision-making! Great question about the number of stations. Let me do some quick queries and find out for us.” And, with that, you get back to work.

We need to write a query to get the number of stations in our dataset. We’ll use our session that we created earlier to query our database.

Begin by adding the starting point for our query, which is the following line:

```
session.query()
```

Continuing with our query, we'll use `func.count`, which essentially counts a given dataset we are interested in. In this case, we want to count the total number of stations. We can do this by referencing `Station.station`, which will give us the number of stations. Add the query parameters to your code, like this:

```
session.query(func.count(Station.station))
```

Now we need to add the `.all()` function to the end of this query so that our results are returned as a list. Your final query should look like the following:

```
session.query(func.count(Station.station)).all()
```

Run the query.

How many stations did your query return?

- 8
- 21
- 9
- 14

Check Answer

Finish ▸

Now we know there are 9 stations from which precipitation data is being collected. However, in order to truly answer W. Avy's question, we don't just need to know the number of stations; we need to know how active the stations are as

well. That is, we want to figure out which stations tend to have the most precipitation recordings. Let’s figure that out next.

9.3.2: Determine the Most Active Stations

Determining how active the stations are will tell us how much data has been collected from each station. In this case, active essentially means the number of recordings for each station. This will help us figure out how reliable our data is, which, in turn, will boost W. Avy's confidence in his investment.

Now that we've found the total number of stations, we need to run a query to determine the most active stations. This query is a bit more complicated, but with your solid understanding of queries, you'll be able to master it!

Begin with the function we use to start every query in SQLAlchemy:

```
session.query()
```

Next, we need to add a few parameters to our query. We'll list the stations and the counts, like this:

```
session.query(Measurement.station, func.count(Measurement.station))
```

Now that we have our core query figured out, let's add a few filters to narrow down the data to show only what we need.

On what class should you use the `groupby()` function?

- `Measurement.date`
- `Measurement.station`
- `Measurement.prcp`

Check Answer

Finish ▸

We want to group the data by the station name, and then order by the count for each station in descending order. We're going to add `group_by()` first.

> **Rewind**
> You have previously used the `groupby()` function. The `group_by()` function for SQLite follows the same idea and groups data similarly.

Here's what your code should look like:

```
session.query(Measurement.station, func.count(Measurement.station)).\
    group_by(Measurement.station)
```

Now let's add the `order_by` function. This function will order our results in the order that we specify, in this case, descending order. Our query results will be returned as a list.

After the code above, add `order_by(func.count(Measurement.station).desc())`, as shown below.

```
session.query(Measurement.station, func.count(Measurement.station)).\
group_by(Measurement.station).order_by(func.count(Measurement.station).de
```

Now we need to add the `.all()` function here as well. This will return all of the results of our query. This is what your query should look like:

```
session.query(Measurement.station, func.count(Measurement.station)).\
group_by(Measurement.station).order_by(func.count(Measurement.station).de
```

Good work! Run this query. Your results should look something similar to this:

```
[('USC00519281', 2772),
 ('USC00519397', 2724),
 ('USC00513117', 2709),
 ('USC00519523', 2669),
 ('USC00516128', 2612),
 ('USC00514830', 2202),
 ('USC00511918', 1979),
 ('USC00517948', 1372),
 ('USC00518838', 511)]
```

In the left column is the station ID, and on the right are the counts for each station. The counts indicate which stations are most active. We can also see which stations are the least active.

Great work! This was a complicated SQLAlchemy query, so you should feel accomplished.

9.3.3: Find Low, High, and Average Temperatures

W. Avy tells you that he's interested in the most active station; he believes it will provide the most data and help you determine the best location for the surf shop. However, you know that more data doesn't necessarily equate to more accurate results. But W. Avy is passionate about the location—he's convinced that it will provide the best weather for surfing and eating ice cream. So you tell him that you'll investigate this location further.

It occurs to you that he hasn't asked for an analysis of the temperature yet, so you decide to dive into temperature data.

Let's get to work on our temperature analysis! We'll be using the results from our last query, which gave us the most active station, to gather some basic statistics. For our most active station, we'll need to find the minimum, maximum, and average temperatures.

Like our previous queries, we'll begin with this line of code:

```
session.query()
```

Next, we will calculate the minimum, maximum, and average temperatures with the following functions: `func.min`, `func.max`, and `func.avg`. Add these functions

to your query, like this:

```
session.query(func.min(Measurement.tobs), func.max(Measurement.tobs), fur
```

With the minimum, maximum, and average in our query, we now need to add one filter. We'll be filtering out everything but the station W. Avy is interested in. If you look at the outcome of the previous query, you can see that the most active station is USC00519281. Therefore, we will need to add this station ID to our filter below.

```
session.query(func.min(Measurement.tobs), func.max(Measurement.tobs), fur
filter(Measurement.station == 'USC00519281')
```

Finally, add the `.all()` function to return our results as a list. Here's what your final query should look like:

```
session.query(func.min(Measurement.tobs), func.max(Measurement.tobs), fur
filter(Measurement.station == 'USC00519281').all()
```

Go ahead and run the query. Your results should look like the following:

```
[(54.0, 85.0, 71.66378066378067)]
```

The results show that the low (minimum) temperature is 54 degrees, the high (maximum) temperature is 85 degrees, and the average temperature is approximately 71.7 degrees.

We have the minimum, maximum, and average temperatures for our station—great work! W. Avy has asked for us to share the results, so let's go above and beyond and create a visualization for him. Visualizing our data will allow us—and W. Avy—

to notice trends, as well as draw more accurate conclusions from it. Let’s plot our data!

9.3.4: Plot the Highest Number of Observations

W. Avy is thrilled. He's ready to make a decision about the surf shop location. But before he takes his proposal to the board of directors, he could benefit from a visualization of your results. You want to make sure that all stakeholders have all the information they need about the station closest to your proposed surf shop location.

You tell W. Avy that you're plotting the results of the analysis so that he can share a visual presentation with the board if needed, and convince them to invest in your shop.

We need to create a plot that shows all of the temperatures in a given year for the station with the highest number of temperature observations.

Create a Query for the Temperature Observations

To create a query, first select the column we are interested in. We want to pull `Measurement.tobs` in order to get our total observations count. Add this to your code:

```
session.query(Measurement.tobs)
```

Now filter out all the stations except the most active station with `filter(Measurement.station == 'USC00519281')`. Your code should look like this:

```
results = session.query(Measurement.tobs).\
filter(Measurement.station == 'USC00519281')
```

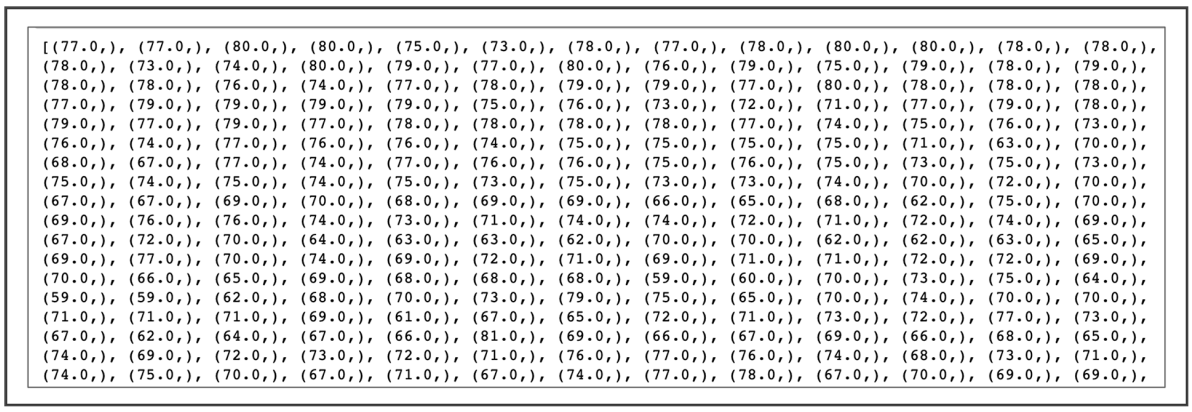
We need to apply another filter to consider only the most recent year. For this we can reuse some of the code we have written previously. Then we'll add the `.all()` function to save our results as a list. Here's what your query should look like:

```
results = session.query(Measurement.tobs).\
filter(Measurement.station == 'USC00519281').\
filter(Measurement.date >= prev_year).all()
```

To run this code, you will need to add a print statement around it.

```
print(results)
```

Your query results should look like the following.



Not too easy to read, right? Let's fix that, as investors will need to read this data.

Which of the following is the best way to make the data easier to read?

- Put it into a DataFrame.
- Create a string.
- Put it into a list.

Check Answer

Finish ▸

To make the results easier to read, understand, and use, we'll put them in a DataFrame.

Convert the Temperature Observation Results to a DataFrame

> **Rewind**
>
> When creating a DataFrame, our first parameter is our list, and the second parameter is the column(s) we want to put our data in. In this case, we want to put our temperature observations result list into a DataFrame.

To convert the results to a DataFrame, add the following to your code:

```
df = pd.DataFrame(results, columns=['tobs'])
```

Add a `print(df)` statement after the last line and run the code. Below is what your data should now look like. Feel free to remove the index column.

	tobs
0	77.0
1	77.0
2	80.0
3	80.0
4	75.0
5	73.0
6	78.0
7	77.0
8	78.0
9	80.0
10	80.0
11	78.0
12	78.0
13	78.0
14	73.0
15	74.0
16	80.0

Awesome! Now we'll use this data to create a plot.

Plot the Temperature Observations

We'll be creating a histogram from the temperature observations. This will allow us to quickly count how many temperature observations we have.

> **Important**
>
> A **histogram** is a graph made up of a range of data that is separated into different bins.
>
> When creating a histogram, you'll need to figure how many bins you need. It's recommended that you stay within a range of 5 to 20 bins. You may need to play around with the data a bit to find a good fit somewhere between 5 and 20. A "good fit" is one that represents the data well and highlights areas where there is a lot of data and areas where there is not a lot of data. It's all about finding the right balance.

We're going to divide our temperature observations into 12 different bins. This is intended to provide enough detail, but not too much. Note that we don't need to specify the ranges in which the data will be separated; we just need to specify the number of bins.

To create the histogram, we need to use the `plot()` function and the `hist()` function and add the number of bins as a parameter. Add the following to your code:

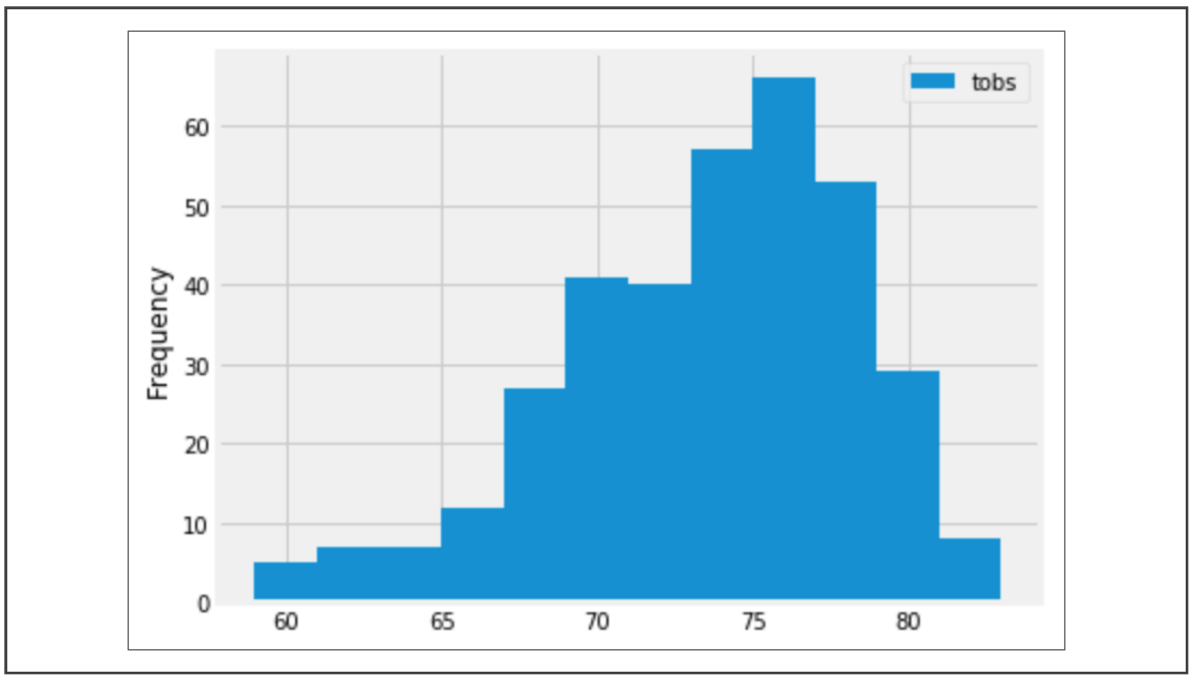
```
df.plot.hist(bins=12)
```

Using `plt.tight_layout()`, we can compress the x-axis labels so that they fit into the box holding our plot.

```
plt.tight_layout()
```

For this particular graph, using this function won't change much, but it can be a lifesaver in situations where the x-axis doesn't fit into the box. It's a cosmetic change, but it makes a big difference when presenting professional work.

When you run the code, your plot should look like the following. Notice how the 12 "bins" are visualized in this plot, just like you specified with your code `df.plot.hist(bins=12)`. "Bin" refers to each rectangular column in the plot, as shown below.



Looking at this plot, we can infer that a vast majority of the observations were over 67 degrees. If you count up the bins to the right of 67 degrees, you will get about 325 days where it was over 67 degrees when the temperature was observed.

> ### Add, Commit, Push
>
> The weather station analysis is complete! Now is a good time to add, commit, and push your updates to GitHub. Remember the steps:
>
> 1. `git add < FILENAME>`
> 2. `git commit -m "< ADD COMMIT MESSAGE HERE>"`
> 3. `git push origin master`

Now test your skills in the following Skill Drill.

> ### Skill Drill
>
> Adjust the number of bins in the plot to 5, and then adjust the number to 20. Take note of any differences in the plot caused by changing the number of bins.

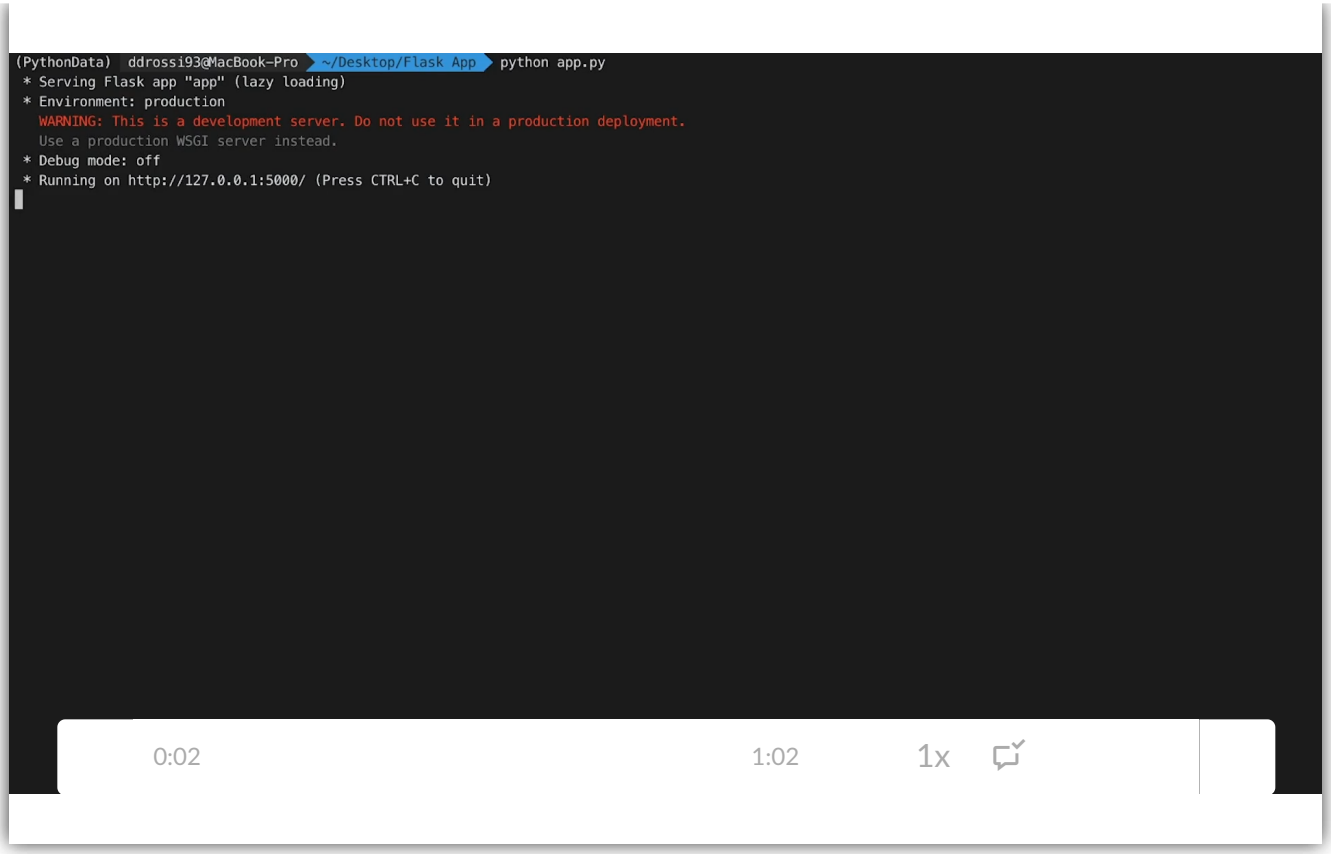
Our work with the precipitation and station analysis is complete, so let’s share our findings!

9.4.1: Incorporate Flask into Data Analysis

You're having breakfast with W. Avy the day before the big presentation. Your analysis is sound, the eggs are fluffy, and there are birds chirping outside. All is well with the world! Suddenly, W. Avy starts to look concerned: “I know this code works, you know it works, but how will we show it to my board of directors? I doubt they even know what GitHub is, much less how to use it.”

You pause to think. W. Avy has a point, the board of directors probably doesn’t really care about the mechanics of the code you’ve written. They just want to be able to access the results. You respond, "I got this, W. Avy. Just leave it to me. I’ll use Flask, which will let me display my results in a webpage. All you have to do is provide your board with the URL.”

You're now going to add a new tool to your data analysis toolbelt: Flask. Watch the following video to learn more about this web framework.



Flask allows you to create Python applications and then share the results of those applications with others via a webpage, making it a powerful tool for data analysis and visualization.

> **Note**
>
> Flask is also helpful when it comes to the job search. You can share your work in a web interface which is simple and effective, rather than viewing code on GitHub. Many employers will want to see your code, but more importantly, they want to see what it can do.

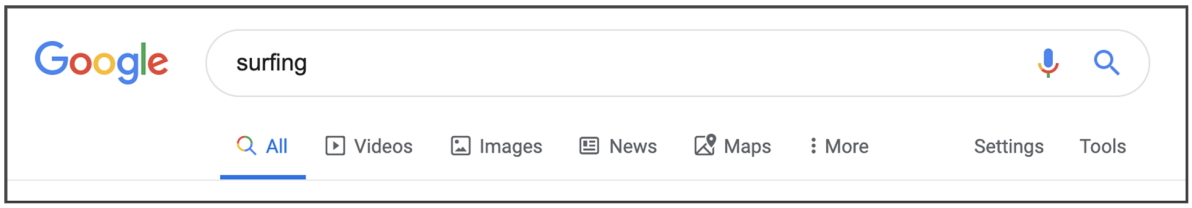
Your audience is a key factor when it comes to data visualization. There may be people in your audience who are not interested in the code itself, but rather what the code can do. Flask makes it possible to summarize the key ideas of your code in a way that allows people who don't have as much coding experience to understand the results of your code. We'll create a Flask application that will allow us to share our findings in an easy-to-interpret way.

Let’s get started by connecting our database so that we can set up our Flask application. We’ll also create a new Python file for the application.

9.4.2: Building Flask Routes

You and W. Avy finish your breakfast quickly and grab an extra coffee to go—you want to get to work building the Flask application so that it's foolproof for the board of directors' meeting tomorrow. You know the first thing you must figure out is how many routes your application needs.

Routes are a core Flask concept. They can be tricky to build (we'll get into that in a moment) but conceptually they are straightforward. In fact, you use routes whenever you google something, for example. If you search for "surfing," you get a number of different categories of pages offered to you: images, maps, news, videos, and more.



These are all different **routes,** or different pathways that a search can take. When we build our webpage for W. Avy, we'll need to make sure we have the correct routes, so that when investors click on the URL they can clearly navigate to the analysis they want to see.

We're going to create five routes for our investors: Welcome, Precipitation, Stations, Monthly Temperature, and Statistics. But first things first: let's install Flask.

9.4.3: Set Up Flask and Create a Route

You need to get to work! You get out your laptop—it's time to install Flask and get acquainted with how to programmatically create a Flask route.

We need to set up Flask and then get started on creating our first Flask route. When creating a Flask application, we'll need to do a few things first. Here's our course of action:

1. Install Flask.
2. Create a new Python file.
3. Import the Flask dependency.
4. Create a new Flask app instance.
5. Create Flask routes.
6. Run a Flask app.

Install Flask

You can install Flask by running the following in the command line:

```
pip install flask
```

If you already have Flask installed, the output of your code will show that you have already installed many of the components, if not all of them. Running this command will ensure that you have the most up-to-date version of Flask.

Create a New Python File and Import the Flask Dependency

Create a new Python file called `flask_example.py`. You should create this file in VS Code.

Once the Python file is created, we can import the dependency we need. This dependency will enable your code to access all that Flask has to offer.

To import the Flask dependency, add the following to your code:

```
from flask import Flask
```

Create a New Flask App Instance

We're now ready to create a new Flask app instance. "Instance" is a general term in programming to refer to a singular version of something. Add the following to your code to create a new Flask instance called `app`:

```
app = Flask(__name__)
```

> **Important**
> You probably noticed the `__name__` variable inside of the `Flask()` function. Let's pause for a second and identify what's going on here.
>
> This `__name__` variable denotes the name of the current function. You can

use the `__name__` variable to determine if your code is being run from the command line or if it has been imported into another piece of code. Variables with underscores before and after them are called **magic methods** in Python.

For more information, see this **article on magic methods (https://www.geeksforgeeks.org/dunder-magic-methods-python/)** .

Create Flask Routes

Our Flask app has been created—let's create our first route!

First, we need to define the starting point, also known as the **root.** To do this, we'll use the function `@app.route('/')`. Add this to your code now.

```
@app.route('/')
```

> **Note**
> Notice the forward slash inside of the `app.route`? This denotes that we want to put our data at the root of our routes. The forward slash is commonly known as the highest level of hierarchy in any computer system.

Next, create a function called `hello_world()`. Whenever you make a route in Flask, you put the code you want in that specific route below `@app.route()`. Here's what it will look like:

```
@app.route('/')
def hello_world():
    return 'Hello world'
```

Great job! You have just created your first Flask route! Now that we have some code, let's keep running.

Run a Flask App

The process of running a Flask app is a bit different from how we've run Python files. To run the app, we're first going to need to use the command line to navigate to the folder where we've saved our code. You should save this code in the same folder you've used in the rest of this module.

Once you've ensured that your code is saved in the proper directory, then run the following command if you are on a Mac. This command sets the `FLASK_APP` environment variable to the name of our Flask file, `flask_example.py`.

> **Note**
> Environment variables are essentially dynamic variables in your computer. They are used to modify the way a certain aspect of the computer operates. For our FLASK_APP environment variable, we want to modify the path that will run our `flask_example.py` file so that we can run our file.

```
export FLASK_APP=flask_example.py
```

There won't be any output when you run this command, so don't worry if you don't see anything.

If you are on a Windows computer, you will need to do the same thing, but in a slightly different way. Start by opening up Anaconda Shell. Once you've done that, enter this command.

```
set FLASK_APP=flask_example.py
```

Now let's run our Flask app. To do this, type the following command in your command line and press Enter:

```
flask run
```

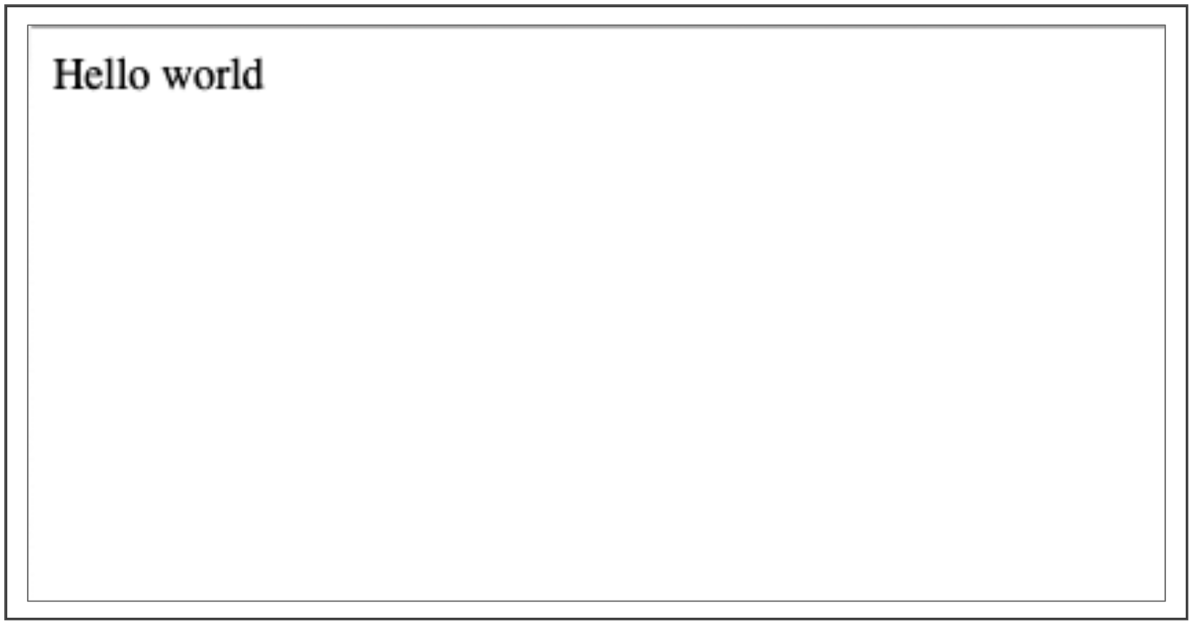
When you run this command, you'll notice a line that says "Running on" followed by an address. This should be your localhost address and a port number.

> **Important**
> A port number is essentially the endpoint of a given program or service. Any Flask application you create can have whatever port number you would like, but the most common is 5000.

Copy and paste your localhost address into your web browser. Generally, a localhost will look something like this, for context.

```
localhost:5000
```

This is what you should see:



You ran your first Flask app! You should feel ready to create more routes that incorporate our analysis.

For some extra practice, let's create another route in the following Skill Drill.

Skill Drill

Think of some simple code from which you could create a route. Then try to create a new route implementing that logic.

Add, Commit, Push

Great work on the Flask app. Now is a good time to add, commit, and push your updates to GitHub. Remember these steps:

1. `git add < FILE NAME>`
2. `git commit -m "< ADD COMMIT MESSAGE HERE>"`
3. `git push origin master`

9.5.1: Set Up the Database and Flask

With your Flask application set up, you know there is only one more step separating you and long days filled with surfing and ice cream on Oahu: creating the appropriate routes so that W. Avy's board of directors will be able to easily access the analysis. You know you'll want to put together a route for each segment of your analysis: Precipitation, Stations, Monthly Temperature, and Statistics, as well as a welcome route that will orient W. Avy and his associates to the webpage.

You know this task might be tough, but motivated by the fact that this is the final step, you refill your coffee and get back to making your dreams come true.

We've learned how to set up and create a Flask application. Now it's time to create our routes so that W. Avy's board of directors can easily access our analysis. We're so close, so stay focused because this will be good stuff! Let's begin by creating a new Python file and importing dependencies our app requires.

Set Up the Flask Weather App

We need to create a new Python file and import our dependencies to our code environment. Begin by creating a new Python file named `app.py`. This will be the file we use to create our Flask application.

Once the Python file is created, let's get our dependencies imported. The first thing we'll need to import is datetime, NumPy, and Pandas. We assign each of these an alias so we can easily reference them later. Add these dependencies to the top of your `app.py` file.

```
import datetime as dt
import numpy as np
import pandas as pd
```

Now let's get the dependencies we need for SQLAlchemy, which will help us access our data in the SQLite database. Add the SQLAlchemy dependencies after the other dependencies you already imported in `app.py`.

```
import sqlalchemy
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine, func
```

Finally, add the code to import the dependencies that we need for Flask. You'll import these right after your SQLAlchemy dependencies.

```
from flask import Flask, jsonify
```

Good work! Now that we've created the Python file and imported dependencies, we're ready to set up our database engine.

Set Up the Database

We'll set up our database engine for the Flask application in much the same way we did for `climate_analysis.ipynb`, so most of this setup process will be familiar. Add the following code to your file:

```
engine = create_engine("sqlite:///hawaii.sqlite")
```

What does the following line of code allow you to do?

```
engine = create_engine("sqlite:///hawaii.sqlite")
```

- Access the SQLite database.
- Change data in the SQLite database.
- Create a new Flask application.

Check Answer

Finish ▸

The `create_engine()` function allows us to access and query our SQLite database file. Now let's reflect the database into our classes.

```
Base = automap_base()
```

Just as we did previously, we're going to reflect our tables.

What Python Flask function will you use to reflect the tables?

- `Base.prepare()`
- `Base.reflect()`

Check Answer

Finish ▸

Add the following code to reflect the database:

```
Base.prepare(engine, reflect=True)
```

With the database reflected, we can save our references to each table. Again, they'll be the same references as the ones we wrote earlier in this module. We'll create a variable for each of the classes so that we can reference them later, as shown below.

```
Measurement = Base.classes.measurement
Station = Base.classes.station
```

Finally, create a session link from Python to our database with the following code:

```
session = Session(engine)
```

Next, we need to define our app for our Flask application.

Set Up Flask

To define our Flask app, add the following line of code. This will create a Flask application called “app.”

```
app = Flask(__name__)
```

Notice the `__name__` variable in this code. This is a special type of variable in Python. Its value depends on where and how the code is run. For example, if we wanted to import our `app.py` file into another Python file named `example.py`, the variable `__name__` would be set to `example`. Here’s an example of what that might look like:

```
import app

print("example __name__ = %s", __name__)

if __name__ == "__main__":
    print("example is being run directly.")
else:
    print("example is being imported")
```

However, when we run the script with `python app.py`, the `__name__` variable will be set to `__main__`. This indicates that we are not using any other file to run this code.

Now we’re ready to build our Flask routes!

9.5.2: Create the Welcome Route

Our first route will be one of the most important. We need to ensure our investors can easily access all of our analysis, so our welcome route will essentially be the entryway to the rest of our analysis.

Our first task when creating a Flask route is to define what our route will be. We want our welcome route to be the **root,** which in our case is essentially the homepage.

> **Rewind**
> To understand routes, remember the Google example we used earlier. If you google "surfer," for example, you'll see search options for images, videos, news, maps, and more. These are all the different "routes" you can take, and the Google homepage is essentially the root.

> **Important**
> All of your routes should go after the `app = Flask(__name__)` line of code. Otherwise, your code may not run properly.

We can define the welcome route using the code below:

```
@app.route("/")
```

Now our root, or welcome route, is set up. The next step is to add the routing information for each of the other routes. For this we'll create a function, and our return statement will have f-strings as a reference to all of the other routes.This will ensure our investors know where to go to view the results of our data.

First, create a function `welcome()` with a return statement. Add this line to your code:

```
def welcome():
    return()
```

Next, add the precipitation, stations, tobs, and temp routes that we'll need for this module into our return statement. We'll use f-strings to display them for our investors.

```
def welcome():
    return(
    '''
    Welcome to the Climate Analysis API!
    Available Routes:
    /api/v1.0/precipitation
    /api/v1.0/stations
    /api/v1.0/tobs
    /api/v1.0/temp/start/end
    ''')
```

> **Note**
> When creating routes, we follow the naming convention `/api/v1.0/` followed by the name of the route. This convention signifies that this is version 1 of our application. This line can be updated to support future versions of the app as well.

The welcome route is now defined, so let's try to run our code. You can run Flask applications like any other Python file, but you'll need a web browser to view the results.

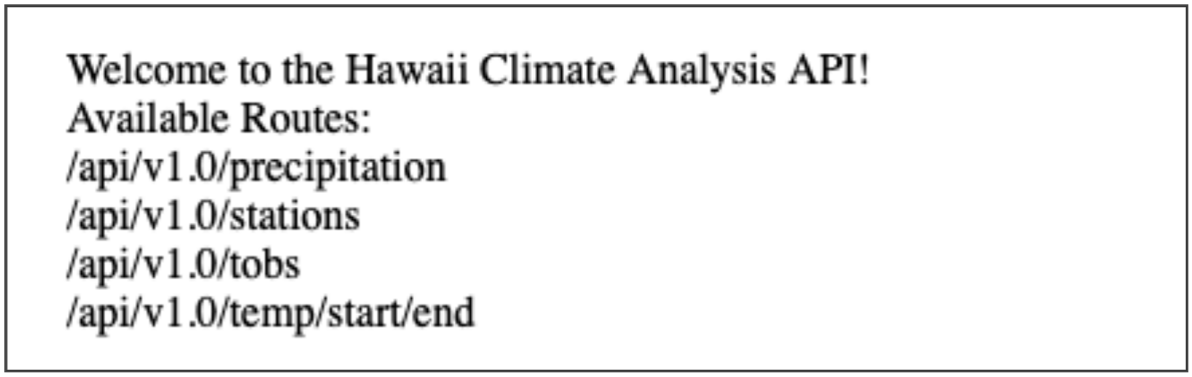
Let's start by using the command line to navigate to your project folder. Then run your code.

```
python app.py
```

After running the code, you'll likely see more text output than you have so far. This is exactly what should happen. The output will probably look something like the following, with a web address where you can view your results:

```
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Now, all you need to do is copy and paste that web address into your web browser and you'll be able to see your second Flask route!



Great work on successfully setting up your welcome route. You're learning how to build and run a more complex Flask application. Next, we'll split up the code we wrote for the temperature analysis, precipitation analysis, and station analysis, and apply it to the respective routes. Let's start with the precipitation route.

9.5.3: Precipitation Route

The next route will return the precipitation data for the last year. We've kept W. Avy in the loop while we've been coding, of course, but by building this route he'll be able to access this analysis in real time with just a URL.

The next route we'll build is for the precipitation analysis. This route will occur separately from the welcome route.

> **Caution**
> Every time you create a new route, your code should be aligned to the left in order to avoid errors.

To create the route, add the following code. Make sure that it's aligned all the way to the left.

```
@app.route("/api/v1.0/precipitation")
```

Next, we will create the `precipitation()` function.

```
def precipitation():
    return()
```

Now we can add code to the function. This code may look almost identical to code you've written previously, but now we'll dive deeper into our precipitation analysis and figure out how to best integrate it into our application.

First, we want to add the line of code that calculates the date one year ago from the most recent date in the database. Do this now so that your code looks like the following:

```
def precipitation():
    prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)
    return()
```

Next, write a query to get the date and precipitation for the previous year. Add this query to your existing code.

```
def precipitation():
    prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)
    precipitation = session.query(Measurement.date, Measurement.prcp).\
        filter(Measurement.date >= prev_year).all()
    return()
```

> **Hint**
>
> Notice the `.\` in the first line of our query? This is used to signify that we want our query to continue on the next line. You can use the combination of `.\` to shorten the length of your query line so that it extends to the next line.

Finally, we'll create a dictionary with the date as the key and the precipitation as the value. To do this, we will "jsonify" our dictionary. `Jsonify()` is a function that converts the dictionary to a JSON file.

> **Rewind**

> JSON files are structured text files with attribute-value pairs and array data types. They have a variety of purposes, especially when downloading information from the internet through API calls. We can also use JSON files for cleaning, filtering, sorting, and visualizing data, among many other tasks. When we are done modifying that data, we can push the data back to a web interface, like Flask.

We'll use `jsonify()` to format our results into a JSON structured file. When we run this code, we'll see what the JSON file structure looks like. Here's an example of what a JSON file might look like:

```
{
"city" : {
"name" : "des moines",
        "region" : "midwest"
}
```

```
def precipitation():
    prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)
    precipitation = session.query(Measurement.date, Measurement.prcp).\
     filter(Measurement.date >= prev_year).all()
    precip = {date: prcp for date, prcp in precipitation}
    return jsonify(precip)
```

Our second route is defined! Now let's run our code. If your application is still running from the previous route, be sure to either close the window or quit the Flask application. Once you run your code, your output will look like the below. As a reminder, you can copy `http://127.0.0.1:5000/` into your web browser to see the results of your code. If you see this window, that means that you've properly setup your Flask application.

```
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

At this point, you might be wondering where to view the output of your code. Just like we did previously, copy the web address (`http://127.0.0.1:5000/`) into our web browser. You will need to navigate to the precipitation route in order to see the output of your code. You can do this by adding `api/v1.0/precipitation` to the end of the web address. This is what your output should look like:

Good work! You have your first route with code logic and are now prepared to build the other routes. The next route we'll turn to is the stations route.

9.5.4: Stations Route

Remember all of the work you did on the stations analysis? Now you'll build a route for your app that will allow this analysis to come to life.

You completed two routes, so now it's time to move on to the third: the stations route. For this route we'll simply return a list of all the stations.

Begin by defining the route and route name. As a reminder, this code should occur outside of the previous route and have no indentation. Add this route to your code:

```
@app.route("/api/v1.0/stations")
```

With our route defined, we'll create a new function called `stations()`. Go ahead and add the following code:

```
def stations():
    return()
```

Now we need to create a query that will allow us to get all of the stations in our database. Let's add that functionality to our code:

```
def stations():
results = session.query(Station.station).all()
    return()
```

We want to start by unraveling our results into a one-dimensional array. To do this, we want to use the `function np.ravel()`, with `results` as our parameter.

Next, we will convert our unraveled results into a list. To convert the results to a list, we will need to use the list function, which is `list()`, and then convert that array into a list. Then we'll jsonify the list and return it. Let's add that functionality to our code:

```
def stations():
results = session.query(Station.station).all()
stations = list(np.ravel(results))
return jsonify(stations)
```

The stations route is ready to be tested! To test it, run the code in the command line and then check if the result is correct in the web browser (`http://localhost:5000/`). Don't forget to add the remainder of the route to see the output of your code. Here's what your results should look like in the web browser:


```
["USC00519397","USC00513117","USC00514830","USC00517948","USC00518838",
"USC00519523","USC00519281","USC00511918","USC00516128"]
```


If your output is not the same as above, make sure to double-check your code to ensure you didn't miss anything.

Once you've got everything looking correct, you are ready to move on to the temperature observations route.

9.5.5: Monthly Temperature Route

You're making progress! You have just two more routes to create and then you'll be ready to share your app with W. Avy and the investors. Now you'll create the temperature observations route.

For this route, the goal is to return the temperature observations for the previous year. As with the previous routes, begin by defining the route with this code:

```
@app.route("/api/v1.0/tobs")
```

Next, create a function called `temp_monthly()` by adding the following code:

```
def temp_monthly():
    return()
```

Now, calculate the date one year ago from the last date in the database. (This is the same date as the one we calculated previously.) Your code should look like this:

```
def temp_monthly():
prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)
    return()
```

The next step is to query the primary station for all the temperature observations from the previous year. Here's what the code should look like with the query

statement added:

```
def temp_monthly():
prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)
    results = session.query(Measurement.tobs).\
filter(Measurement.station == 'USC00519281').\
filter(Measurement.date >= prev_year).all()
    return()
```

Finally, as before, unravel the results into a one-dimensional array and convert that array into a list. Then jsonify the list and return our results, like this:

```
def temp_monthly():
prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)

      results = session.query(Measurement.tobs).\
           filter(Measurement.station == 'USC00519281').\
           filter(Measurement.date >= prev_year).all()

      temps = list(np.ravel(results))
```

As we did earlier, we want to jsonify our `temps` list, and then return it. Add the return statement to the end of your code so that the route looks like this:

```
def temp_monthly():
prev_year = dt.date(2017, 8, 23) - dt.timedelta(days=365)

    results = session.query(Measurement.tobs).\
filter(Measurement.station == 'USC00519281').\
filter(Measurement.date >= prev_year).all()
temps = list(np.ravel(results))
    return jsonify(temps)
```

Before moving on to the next route, let's test our code to make sure it works.

> **Note**

> Testing your code regularly can help prevent any errors or bugs in your code. If errors occur, testing your code often will allow you to pinpoint where and why the error is occurring.

To test your code, navigate to your project folder, surfs_up, for this module in the command line. Then, run the following command:

```
python app.py
```

Use the web address to see the output of your code and the various routes you've created. For this route, use the web address (`http://localhost:5000/ (http://localhost:5000/)`) provided by Flask in the command line, which will hold all of our monthly temperature readings. Here's what your output should look like:


```
[77.0,77.0,80.0,80.0,75.0,73.0,78.0,77.0,78.0,80.0,80.0,78.0,78.0,78.0,73.0,74.0,80.0,79.0,
77.0,80.0,76.0,79.0,75.0,79.0,78.0,79.0,78.0,78.0,76.0,74.0,77.0,78.0,79.0,79.0,77.0,80.0,7
8.0,78.0,78.0,77.0,79.0,79.0,79.0,79.0,75.0,76.0,73.0,72.0,71.0,77.0,79.0,78.0,79.0,77.0,79
.0,77.0,78.0,78.0,78.0,78.0,77.0,74.0,75.0,76.0,73.0,76.0,74.0,77.0,76.0,76.0,74.0,75.0,75.
0,75.0,75.0,71.0,63.0,70.0,68.0,67.0,77.0,74.0,77.0,76.0,76.0,75.0,76.0,75.0,73.0,75.0,73.0
,75.0,74.0,75.0,74.0,75.0,73.0,75.0,73.0,73.0,74.0,70.0,72.0,70.0,67.0,67.0,69.0,70.0,68.0,
69.0,69.0,66.0,65.0,68.0,62.0,75.0,70.0,69.0,76.0,76.0,74.0,73.0,71.0,74.0,74.0,72.0,71.0,7
2.0,74.0,69.0,67.0,72.0,70.0,64.0,63.0,63.0,62.0,70.0,70.0,62.0,62.0,63.0,65.0,69.0,77.0,70
.0,74.0,69.0,72.0,71.0,69.0,71.0,71.0,72.0,72.0,69.0,70.0,66.0,65.0,69.0,68.0,68.0,68.0,59.
0,60.0,70.0,73.0,75.0,64.0,59.0,59.0,62.0,68.0,70.0,73.0,79.0,75.0,65.0,70.0,74.0,70.0,70.0
,71.0,71.0,71.0,69.0,61.0,67.0,65.0,72.0,71.0,73.0,72.0,77.0,73.0,67.0,62.0,64.0,67.0,66.0,
81.0,69.0,66.0,67.0,69.0,66.0,68.0,65.0,74.0,69.0,72.0,73.0,72.0,71.0,76.0,77.0,76.0,74.0,6
8.0,73.0,71.0,74.0,75.0,70.0,67.0,71.0,67.0,74.0,77.0,78.0,67.0,70.0,69.0,69.0,74.0,78.0,71
.0,67.0,68.0,67.0,76.0,69.0,72.0,76.0,68.0,72.0,74.0,70.0,67.0,72.0,60.0,65.0,75.0,70.0,75.
0,70.0,79.0,75.0,70.0,67.0,74.0,70.0,75.0,76.0,77.0,74.0,74.0,74.0,69.0,68.0,76.0,74.0,71.0
,71.0,74.0,74.0,74.0,74.0,80.0,74.0,72.0,75.0,80.0,76.0,76.0,77.0,75.0,75.0,75.0,75.0,72.0,
74.0,74.0,74.0,76.0,74.0,75.0,73.0,79.0,75.0,72.0,72.0,74.0,72.0,72.0,77.0,71.0,73.0,76.0,7
7.0,76.0,76.0,79.0,81.0,76.0,78.0,77.0,74.0,75.0,78.0,78.0,69.0,72.0,74.0,74.0,76.0,80.0,80
.0,76.0,76.0,76.0,77.0,77.0,77.0,82.0,75.0,77.0,75.0,76.0,81.0,82.0,81.0,76.0,77.0,82.0,83.
0,77.0,77.0,77.0,76.0,76.0,79.0]
```


Great work! Next, let's create a route for the statistics analysis.

9.5.6: Statistics Route

The investors will need to see the minimum, maximum, and average temperatures. For this we'll create a route for our summary statistics report.

Just one more route to create! Our last route will be to report on the minimum, average, and maximum temperatures. However, this route is different from the previous ones in that we will have to provide both a starting and ending date. Add the following code to create the routes:

```
@app.route("/api/v1.0/temp/<start>")
@app.route("/api/v1.0/temp/<start>/<end>")
```

Next, create a function called `stats()` to put our code in.

```
def stats():
    return()
```

We need to add parameters to our `stats()` function: a `start` parameter and an `end` parameter. For now, set them both to `None`.

```
def stats(start=None, end=None):
return()
```

With the function declared, we can now create a query to select the minimum, average, and maximum temperatures from our SQLite database. We'll start by just creating a list called `sel`, with the following code:

```
def stats():
sel = [func.min(Measurement.tobs), func.avg(Measurement.tobs), func.max(M
```

Since we need to determine the starting and ending date, add an `if-not` statement to our code. This will help us accomplish a few things. We'll need to query our database using the list that we just made. Then, we'll unravel the results into a one-dimensional array and convert them to a list. Finally, we will jsonify our results and return them.

> **Note**
> In the following code, take note of the asterisk in the query next to the `set` list. Here the asterisk is used to indicate there will be multiple results for our query: minimum, average, and maximum temperatures.

```
def stats():
    sel = [func.min(Measurement.tobs), func.avg(Measurement.tobs),

    if not end:
      results = session.query(*sel).\
       filter(Measurement.date >= start).\
       filter(Measurement.date <= end).all()


    temps = list(np.ravel(results))
    return jsonify(temps)
```

Now we need to calculate the temperature minimum, average, and maximum with the start and end dates. We'll use the `sel` list, which is simply the data points we need to collect. Let's create our next query, which will get our statistics data.

```
def stats(start=None, end=None):
        sel = [func.min(Measurement.tobs), func.avg(Measurement.tobs), fu

    if not end:
      results = session.query(*sel).\
       filter(Measurement.date <= start).all()
          temps = list(np.ravel(results))
          return jsonify(temps)

    results = session.query(*sel).\
          filter(Measurement.date >= start).\
        filter(Measurement.date <= end).all()
    temps = list(np.ravel(results))

    return jsonify(temps)
```

Finally, we need to create an `if` statement to run our code. Without this statement, the code will not create the Flask application. To do this, navigate to the “surfs_up” folder in the command line, and then enter the following command to run your code:

```
python app.py
```

After running this code, you’ll be able to copy and paste the web address provided by Flask into a web browser. Open `/api/v1.0/temp/start/end route` and check to make sure you get the correct result, which is:

```
[null,null,null]
```

This code tells us that we have not specified a start and end date for our range. Fix this by entering any date in the dataset as a start and end date. The code will output the minimum, maximum, and average temperatures. For example, let’s say

we want to find the minimum, maximum, and average temperatures for June 2017. You would add the following path to the address in your web browser:

```
/api/v1.0/temp/2017-06-01/2017-06-30
```

When you run the code, it should return the following result:

```
[71.0,77.21989528795811,83.0]
```

You've just completed your second Flask application. Great work! Flask and Python are incredibly useful tools to have in your toolbelt. But this is challenging stuff, so give yourself a pat on the back for getting through it—and successfully!

> ### Add, Commit, Push
>
> The Flask app is complete, which is a good time to add, commit, and push our updates to GitHub. Remember these steps:
>
> 1. `git add < FILE NAME>`
> 2. `git commit -m "< ADD COMMIT MESSAGE HERE>"`
> 3. `git push origin master`

Module 9 Challenge

Submit Assignment

Due Mar 15 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> W. Avy likes your analysis, but he feels there is one piece of information missing: seasonal data. He's asked you to gather data on the seasons of Oahu and determine whether the seasons could affect the surf and ice cream shop business. Specifically, are there certain times of the year when business might be slower, or the type of customer could be different?

In this challenge, you will be finding a few key aspects of Oahu's seasonal weather data. The investors want to ensure you've hit all of the key points before opening the surf shop.

Background

Coming from the mainland, you know how variable the weather can be in the summer and winter. Your investors want to ensure that there are enough customers between seasons to sustain the business throughout the year.

Objectives

The goals of this challenge are for you to:

- Determine key statistical data about the month of June.
- Determine key statistical data about the month of December.
- Compare your findings between the month of June and December.

- Make 2 or 3 recommendations for further analysis.
- Share your findings in the Jupyter Notebook.

Instructions

Complete the following steps.

1. Identify key statistical data in June across all of the stations and years using the describe() function.
2. Identify key statistical data in December across all stations and years using the describe() function.
3. Share your findings in the Jupyter Notebook with a few sentences describing the key differences in weather between June and December and 2-3 recommendations for further analysis.

Submission

1. Save all of your code in your repo folder.
2. Add, commit, and push all of the files to your repository.
3. Submit the link to your repository through Canvas.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 9 Rubric

Criteria	Ratings					Pts
Written Analysis	**40.0 pts Mastery** Presents a cohesive written analysis that describes the key differences between the June and December data and provides 2-3 recommendations for further analysis	**30.0 pts Approaching Mastery** Presents a cohesive written analysis that describes the key differences between the June and December data, with minor errors, and provides 1-2 recommendations for further analysis	**20.0 pts Progressing** Presents a developing written analysis that describes the key differences between the June and December data, with minor errors, and provides 1 recommendation for further analysis	**10.0 pts Emerging** Presents a limited written analysis that describes the key differences between the June and December data, with significant errors.	**0.0 pts Incomplete** No submission was received, submission was empty or blank, or submission contains evidence of academic dishonesty	40.0 pts
Average June temperature across all stations and years	**30.0 pts Mastery** Identify count, max, min, mean, standard deviation, and percentiles in June across all of the stations and years using the describe() function, with no errors.	**23.0 pts Approaching Mastery** Identify 5 of the 6 metrics (count, max, min, mean, standard deviation, and percentiles) in June across all of the stations and years using the describe() function, with one or two minor errors.	**17.0 pts Progressing** Identify 4 of the 6 metrics (count, max, min, mean, standard deviation, and percentiles) in June across all of the stations and years using the describe() function, with two or three minor errors.	**10.0 pts Emerging** Identify 3 or fewer of the 6 metrics (count, max, min, mean, standard deviation, and percentiles) in June across all of the stations and years using the describe() function, with significant errors.	**0.0 pts Incomplete** No submission was received, submission was empty or blank, or submission contains evidence of academic dishonesty	30.0 pts
Average December temperature across all stations and years	**30.0 pts Mastery** Identify count, max, min, mean, standard deviation, and percentiles in December across all of the stations and years using the describe()	**23.0 pts Approaching Mastery** Identify 5 of the 6 metrics (count, max, min, mean, standard deviation, and percentiles) in December across all of the stations and years using the describe() function, with	**17.0 pts Progressing** Identify 4 of the 6 metrics (count, max, min, mean, standard deviation, and percentiles) in December across all of the stations and years using the describe() function, with two or three minor errors.	**10.0 pts Emerging** Identify 3 or fewer of the 6 metrics (count, max, min, mean, standard deviation, and percentiles) in December across all of the stations and years using the describe() function, with	**0.0 pts Incomplete** No submission was received, submission was empty or blank, or submission contains evidence of academic dishonesty	30.0 pts
						Total Points: 100.0

function, with no errors.	one or two minor errors.	significant errors.

Module 9 Career Connection

Where to Find More Networking Tips and Practice

Introduction:

We know networking can be intimidating and full of unknowns. How do you approach people with confidence? What should you share when you talk about yourself? How do you build authentic relationships?

Get answers to questions like these and all the support you need to network like a pro in one of our Networking workshops. **Click here to view and register for upcoming sessions. (https://careerservicesonlineevents.splashthat.com/)**

Unit Assessment: Databases

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Some of the questions on this assessment require specific resources. Download the following resources before you get started. There are also several websites used as question resources listed below.

players.csv

matches.csv

schema.txt

dracula.txt

CTA - System Information - List of 'L' Stops (https://data.cityofchicago.org/Transportation/CTA-System-Information-List-of-L-Stops/8pix-ypme)

CTA - Ridership - 'L' Station Entries - Daily Totals (https://data.cityofchicago.org/Transportation/CTA-Ridership-L-Station-Entries-Daily-Totals/5neh-572f)

Census_Data.sqlite

data-Unit-Assessment-Databases

Please click **Start** when you are ready to begin the activity.

Start

data-Unit-Assessment-Databases

Using *players.csv*

Your department is participating as a team in a fantasy tennis competition at work, and your boss wants to drive the decisions with data.

In pgAdmin, create a database called “tennis_db” and make a new table called “players.” Create columns to match the headers in the *players.csv* file. Import the CSV into the new table.

What data type is most appropriate for the “player_id” column?

- `STRING`
- `VARCHAR`
- `INT`
- `FLOAT`

◄ Next ►

data-Unit-Assessment-Databases

Using *matches.csv*

Your boss's favorite player is Serena Williams. They've asked for a list of every match she's won.

Create a new table in tennis_db called "matches." Create columns to match the headers in the *matches.csv* file. Import the CSV into the new table.

Write a query to display all of the matches where Serena Williams won.

How many matches in the dataset has Serena Williams won?

- 41
- 8
- 45
- 37

◄ Next ►

data-Unit-Assessment-Databases

Using the *tennis_db* from questions 1 and 2, write a query that returns the count of players for each dominant hand group (i.e. right, left).

What is the distribution of the players' dominant hands?

- 5023 are left handed, 487 are right handed.
- 93 are left handed, 898 are right handed.
- 487 are left handed, 5023 are right handed.
- 898 are left handed, 1456 are right handed.

◄ Next ►

(continued from last question)

Still using the *tennis_db*, write a query that returns the number of WINS for each dominant hand.

What is the distribution of wins for each dominant hand?

HINT: This will require you to join two tables together.

- 80 wins for left handed players, 2126 wins for right handed players.
- 241 wins for left handed players, 2126 wins for right handed players.
- 2126 wins for left handed players, 241 wins for right handed players.
- 487 wins for left handed players, 5023 wins for right handed players.

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

data-Unit-Assessment-Databases

◄ Next ►

You've been hired to create a SQL database for a local gym owner. Based on your meetings with the owner, you've determined that the initial database design should follow *schema.txt*.
The gym owner contacts you to let you know that the gym is going to start offering drop-in classes. The classes are paid for separately, and trainers are paid a percentage commission for the class (commission is determined for each class).

Using https://app.quickdatabasediagrams.com/#/, add two new tables with the following columns to the ERD:

```
Classes
-
Class_ID PK
Trainer_ID
Gym_ID
Class_Name
Commission_Percentage

Class_Attendance
-
Member_ID
Class_ID
```

Add the appropriate data types and foreign key constraints to your ERD.

How many foreign key relationships needed to be added?

- 2
- 6
- 4

(continued from last question)

Using the ERD from the last question, export the ERD to a PostgreSQL query and open the query.

What is the keyword used when adding the foreign key constraints?

- `ALTER`
- `CREATE`
- `UPDATE`
- `INSERT`

◄ Next ►

(continued from last question)

Create a “gym_db” database in pgAdmin. Open the query tool and run the exported ERD query to create all the tables.

The following SQL code is attempting to insert a class into the database, but it returns an error.

```
insert into gym
(gym_id, gym_name, address, city, zipcode)
values (1, 'Average Joe''s Gymnasium', '123 Main St.',

insert into classes
(class_id, trainer_id, gym_id, class_name, commission_p
values (1,1,1,'Wrench Dodging',0.1);

insert into trainers
(trainer_id, gym_id, first_name, last_name)
values (1, 1, 'Patches', 'O''Houlihan');
```

What needs to be changed for the code to run correctly?

- The insert into “trainers” needs to happen before the insert into “classes”
- The ‘commission_percentage’ value should be a percent, not a decimal
- Foreign key restraints need to be temporarily relaxed

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

data-Unit-Assessment-Databases

◄ Next ►

Using:

CTA - System Information - List of 'L' Stops

CTA - Ridership - 'L' Station Entries - Daily Totals

A journalist has contacted you to perform data analysis for an article they're writing about CTA ridership. They want to investigate how the CTA serves the North and South sides of Chicago. They've provided you two datasets with ridership information and station information, but they need to merge them together to connect ridership data with location data.

The list of L stops has multiple stops per station, to account for the direction the trains are heading in. You need to reduce the information down to one line per station.

1. Drop *STOP_ID*, *DIRECTION_ID*, and *STOP_NAME*
2. Compress the boolean columns (*ADA, RED, BLUE, G, BRN, P, Pexp, Y, Pnk, O*) using pandas.groupby().any()
3. Remove duplicate rows
4. Don't worry about parsing Location, we'll do that in the next question

How many rows are in the transformed dataset?

- 204
- 147

(Continued from the previous question)

The following code will perform the task from the previous question:

```
station_bools =
l_stops_df[['MAP_ID','ADA','RED','BLUE','G','BRN','P',
].groupby('MAP_ID').any()
l_stops_df = l_stops_df.drop(['STOP_ID', 'DIRECTION_ID
'ADA','RED','BLUE','G','BRN','P','Pexp','Y','Pnk','O'],
    .merge(station_bools, how='left', left_on='MAP_ID',
right_index=True).drop_duplicates()
```

A journalist has contacted you to perform data analysis for an article they're writing about CTA ridership. They want to investigate how the CTA serves the North and South sides of Chicago. They've provided you two datasets with ridership information and station information, but they need to merge them together to connect ridership data with location data.

The *Location* column is currently stored as a string. Parse the *Location* column into a *Latitude* and *Longitude* column using a regular expression to replace the parentheses and the `pandas.Series().str.split()` method. Convert the now split numbers to numeric data types.

What character needs to be placed before a parenthesis in a regular expression to escape the parenthesis?

- “
- /

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

(Continued from the previous question)

The following code will perform the task from the previous question:

```
l_stops_df[['latitude','longitude']] =
l_stops_df['Location'].str.replace('\(|\)','',
regex=True).str.split(',',expand=True).apply(pd.to_nume
```

A journalist has contacted you to perform data analysis for an article they're writing about CTA ridership. They want to investigate how the CTA serves the North and South sides of Chicago. They've provided you two datasets with ridership information and station information, but they need to merge them together to connect ridership data with location data.

Merge the ridership dataset and station dataset by *station_id* and *MAP_ID*, respectively.

What type of join is the most appropriate to use for this merge?

- inner
- left
- full outer

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄

Next ►

(Continued from the previous question)

The following code will perform the task from the previous question:

```
df = pd.merge(ridership_df, l_stops_df, how='left', le
right_on='MAP_ID')
```

A journalist has contacted you to perform data analysis for an article they're writing about CTA ridership. They want to investigate how the CTA serves the North and South sides of Chicago. They've provided you two datasets with ridership information and station information, but they need to merge them together to connect ridership data with location data.

Create a boolean column in the merged dataset to determine if a station is in the South side or not, using the latitude 41.881 as a cutoff. Calculate the mean rides per station for the North side and South side.

What is the mean rides per station for the South side (rounded to the nearest whole number)?

- 3015
- 3021
- 2887

◄ Next ►

(Continued from the previous question)

The following code will perform the task from the previous question:

```
df['south_side'] = df['latitude'] < 41.881
df[['south_side','rides']].groupby('south_side').mean()
```

A journalist has contacted you to perform data analysis for an article they're writing about CTA ridership. They want to investigate how the CTA serves the North and South sides of Chicago. They've provided you two datasets with ridership information and station information, but they need to merge them together to connect ridership data with location data.

Convert the *date* column to a datetime data type and create a new column for just the year. Calculate the mean rides per station, grouping by year and side.

In which year was the absolute difference between North side ridership and South side ridership the smallest?

- 2005
- 2009
- 2010

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄ Next ►

(Continued from the previous question)

The following code will perform the task from the previous question:

```
rides_by_sides = df[['year','south_side','rides']] \
    .groupby(['year','south_side']).mean() \
    .reset_index() \
    .pivot(index='year',columns='south_side',values='ri
rides_by_sides.columns = ['north','south']
(rides_by_sides['north'] - rides_by_sides['south']).plo
```

A journalist has contacted you to perform data analysis for an article they're writing about CTA ridership. They want to investigate how the CTA serves the North and South sides of Chicago. They've provided you two datasets with ridership information and station information, but they need to merge them together to connect ridership data with location data.

Clean up any duplicated columns (e.g. *station_id* & *MAP_ID*, *Location & Latitude/Longitude*) in the merged dataset. Open up pgAdmin and create a database called "CTA_DB". Create a schema to match the merged dataset. Load the merged dataset into CTA_DB using `pandas.to_sql()` .

Which column, if any, should be the primary key?

- `north`
- `rides`
- `year`

◄ Next ►

data-Unit-Assessment-Databases

Using *Census_Data.sqlite*

The marketing department wants to focus on areas around the country that satisfy certain demographics.

Download *Census_Data.sqlite* and use SQLAlchemy to connect to the database. Use Pandas to import the Census_Data table into a DataFrame.

Out of the cities with populations over 100,000 people, which one has the youngest median age?

- College Station, TX
- Athens, GA
- Provo, UT
- Delray Beach, FL

◄ Next ►

Consider this code snippet from a Flask app:

```
@app.route("/api/v1.0/users/<id>")
def user(***):
```

What needs to replace `***` in order to get the user's id?

- `id`
- `“/api/v1.0/users/<id>”`
- `<id>`
- We can replace the `***` with any word.

◄ Next ►

The following function returns the index page for a Flask route, but it's missing the Flask decorator:

```
# MISSING CODE HERE
def index():
    """List all available api routes."""
    return (
        f"Available Routes:<br/>"
        f"/api/v1.0/names<br/>"
        f"/api/v1.0/passengers"
    )
```

What line of code needs to be added above the function to route to the index?

- `@app.route('/')`
- `index('/')`
- `app.route('/')`
- `route('/')`

Unit Assessment: Databases

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Some of the questions on this assessment require specific resources. Download the following resources before you get started. There are also several websites used as question resources listed below.

players.csv

matches.csv

schema.txt

dracula.txt

CTA - System Information - List of 'L' Stops (https://data.cityofchicago.org/Transportation/CTA-System-Information-List-of-L-Stops/8pix-ypme)

CTA - Ridership - 'L' Station Entries - Daily Totals (https://data.cityofchicago.org/Transportation/CTA-Ridership-L-Station-Entries-Daily-Totals/5neh-572f)

Census_Data.sqlite

data-Unit-Assessment-Databases

You've been given data on passengers of the Titanic in a SQLite file, and you want to create an API that will serve the data in a JSON format. You've created a bare-bones Flask app that connects to the SQLite database and serves a welcome page. Eventually, you will add two endpoints: one that returns a list of all passenger names, and a more detailed endpoint that returns a list of all passengers, including their name, age, and sex. But first, you need a reference to the *passenger* table in the SQLite database.

```
from sqlalchemy.ext.automap import automap_bas
e
from sqlalchemy.orm import Session
from sqlalchemy import create_engine
```

```
from flask import Flask, jsonify

engine = create_engine("sqlite:///titanic.sqli
te")
Base = automap_base()
Base.prepare(engine, reflect=True)

Passenger = # MISSING CODE HERE

app = Flask(__name__)

@app.route("/")
def welcome():
   """List all available api routes."""
   return (
       f"Available Routes:<br/>"
       f"/api/v1.0/names<br/>"
       f"/api/v1.0/passengers"
   )


if __name__ == '__main__':
   app.run(debug=True)
```

On line 11, what does Passenger need to have assigned to it so that it will refer to the *passenger* table in the SQLite database?

- `sqlite:///titanic.sqlite/passengers`
- `passenger`
- `Base.classes.passenger`

Unit Assessment: Databases

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Some of the questions on this assessment require specific resources. Download the following resources before you get started. There are also several websites used as question resources listed below.

players.csv

matches.csv

schema.txt

dracula.txt

CTA - System Information - List of 'L' Stops (https://data.cityofchicago.org/Transportation/CTA-System-Information-List-of-L-Stops/8pix-ypme)

CTA - Ridership - 'L' Station Entries - Daily Totals (https://data.cityofchicago.org/Transportation/CTA-Ridership-L-Station-Entries-Daily-Totals/5neh-572f)

Census_Data.sqlite

data-Unit-Assessment-Databases

(Continued from the previous question)

Next, create an endpoint to list all of the names of the passengers. Use the route */api/v1.0/names*.

```
from sqlalchemy.ext.automap import automap_bas
e
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

from flask import Flask, jsonify

engine = create_engine("sqlite:///titanic.sqli
te")
Base = automap_base()
Base.prepare(engine, reflect=True)
```

```
Passenger = Base.classes.passenger

app = Flask(__name__)

@app.route("/")
def welcome():
   """List all available api routes."""
   return (
       f"Available Routes:<br/>"
       f"/api/v1.0/names<br/>"
       f"/api/v1.0/passengers"
   )

### Create an endpoint for names ###

if __name__ == '__main__':
   app.run(debug=True)
```

On line 24, what is the first line you will have to write to make a new route for passenger names?

- `route('/api/v1.0/names')`
- `def names():`
- `f"/api/v1.0/names"`
- `@app.route('/api/v1.0/names')`

Item 17

◄ Next ►

Unit Assessment: Databases

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Some of the questions on this assessment require specific resources. Download the following resources before you get started. There are also several websites used as question resources listed below.

players.csv

matches.csv

schema.txt

dracula.txt

CTA - System Information - List of 'L' Stops (https://data.cityofchicago.org/Transportation/CTA-System-Information-List-of-L-Stops/8pix-ypme)

CTA - Ridership - 'L' Station Entries - Daily Totals (https://data.cityofchicago.org/Transportation/CTA-Ridership-L-Station-Entries-Daily-Totals/5neh-572f)

Census_Data.sqlite

data-Unit-Assessment-Databases

(Continued from the previous question)

An endpoint has been created for all passenger data, and it connects to the database and converts all of the data into a list of dictionaries. However, the last line of the function is missing.

```
from sqlalchemy.ext.automap import automap_bas
e
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

from flask import Flask, jsonify

engine = create_engine("sqlite:///titanic.sqli
te")
```

```
Base = automap_base()
Base.prepare(engine, reflect=True)

Passenger = Base.classes.passenger

app = Flask(__name__)

@app.route("/")
def welcome():
   """List all available api routes."""
   return (
       f"Available Routes:<br/>"
       f"/api/v1.0/names<br/>"
       f"/api/v1.0/passengers"
   )

@app.route("/api/v1.0/names")
def names():
   """Return a list of all passenger names"""
   # Query all passengers
   session = Session(engine)
   results = session.query(Passenger.name).all
()

   # Convert list of tuples into normal list
   all_names = list(np.ravel(results))

   return jsonify(results)

@app.route("/api/v1.0/passengers")
def passengers():
   """Return a list of passenger data includin
g the name, age, and sex of each passenger"""
   # Query all passengers
   session = Session(engine)
   results = session.query(Passenger.name, Pas
senger.age, Passenger.sex).all()

   # Create a dictionary from the row data and
append to a list of all_passengers
   all_passengers = []
   for name, age, sex in results:
       passenger_dict = {}
       passenger_dict["name"] = name
       passenger_dict["age"] = age
       passenger_dict["sex"] = sex
       all_passengers.append(passenger_dict)

   ### Return list of all passenger data in JS
```

◄ Next ►

```
ON format ###

if __name__ == '__main__':
   app.run(debug=True)
```

On line 52, what line of code needs to be inserted so that the list of all passenger data will be sent to the end-point in JSON format?

- `return all_passengers`
- `return all_passengers.json()`

Using *dracula.txt*

Use the following code and replace `p` with a regular expression to find the most common word that follows “vampire” in the text:

```
import pandas as pd
import re

dracula_df = pd.read_csv('dracula.txt', sep='\n', heade
dracula_df.columns = ['text']

p = 'YOUR REGULAR EXPRESSION HERE'
dracula_df['text'].str.extractall(p, flags=re.I)[0].val
```

What is the most common word that follows “vampire” in the text?

- rest
- drink
- sleep
- live