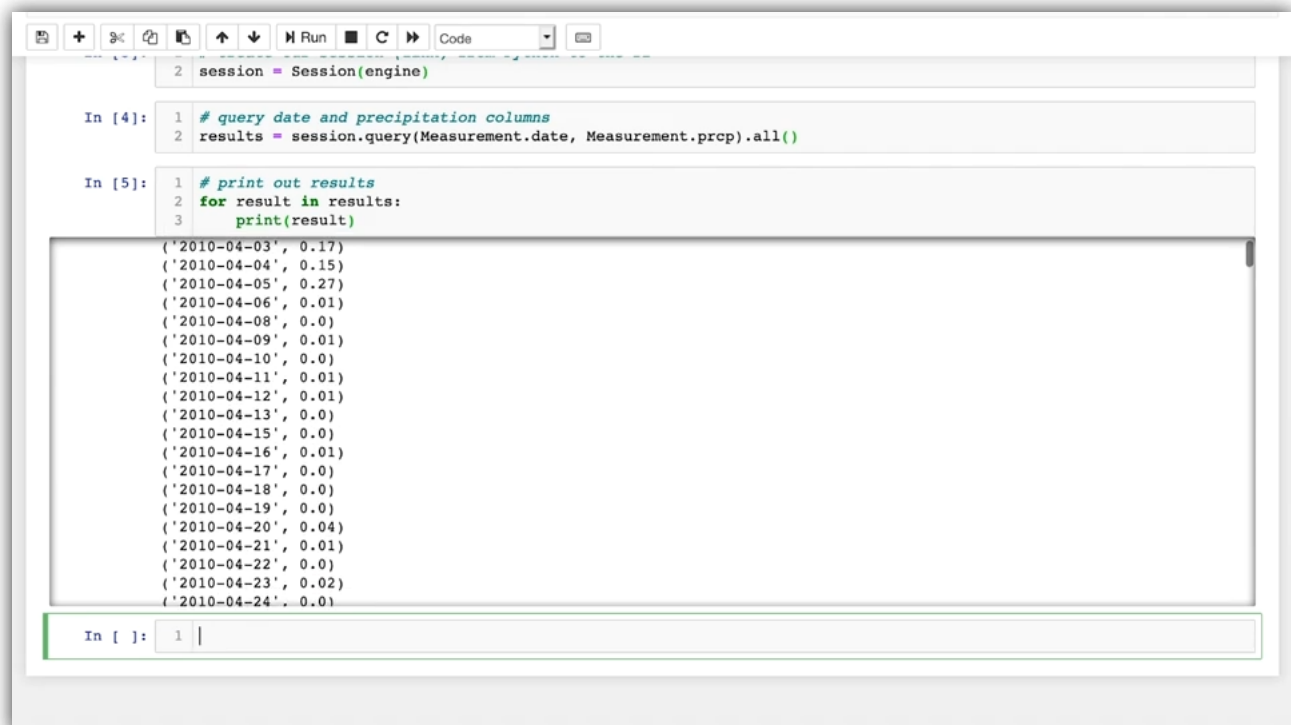


9.1.5

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.



```
2 session = Session(engine)

In [4]: 1 # query date and precipitation columns
        2 results = session.query(Measurement.date, Measurement.prcp).all()

In [5]: 1 # print out results
        2 for result in results:
        3     print(result)

('2010-04-03', 0.17)
('2010-04-04', 0.15)
('2010-04-05', 0.27)
('2010-04-06', 0.01)
('2010-04-08', 0.0)
('2010-04-09', 0.01)
('2010-04-10', 0.0)
('2010-04-11', 0.01)
('2010-04-12', 0.01)
('2010-04-13', 0.0)
('2010-04-15', 0.0)
('2010-04-16', 0.01)
('2010-04-17', 0.0)
('2010-04-18', 0.0)
('2010-04-19', 0.0)
('2010-04-20', 0.04)
('2010-04-21', 0.01)
('2010-04-22', 0.0)
('2010-04-23', 0.02)
('2010-04-24', 0.0)
```

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']`



Feedback

Correct. Nice work! These are the correct keys.

 Retake

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 to 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 main`

© 2020 - 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.