# Lab: SQLAlchemy

This document defines the problems for the in-class lab for the [Python ORM course @ Software University](https://softuni.bg/trainings/4253/python-orm-october-2023).

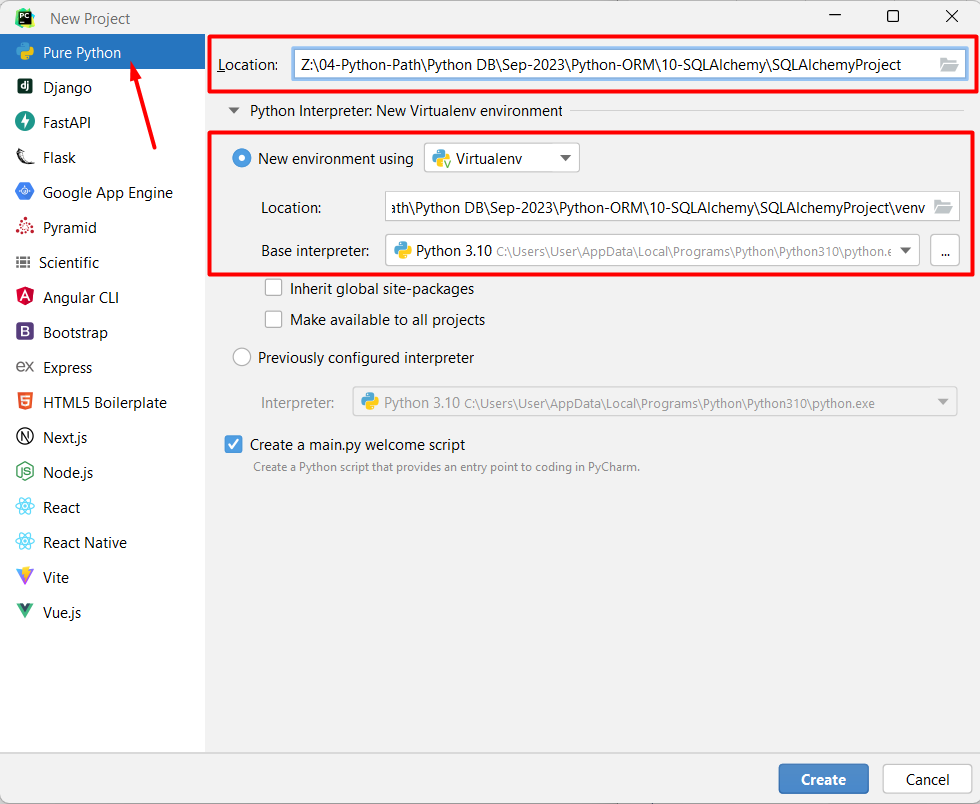
## Installation and Configuration

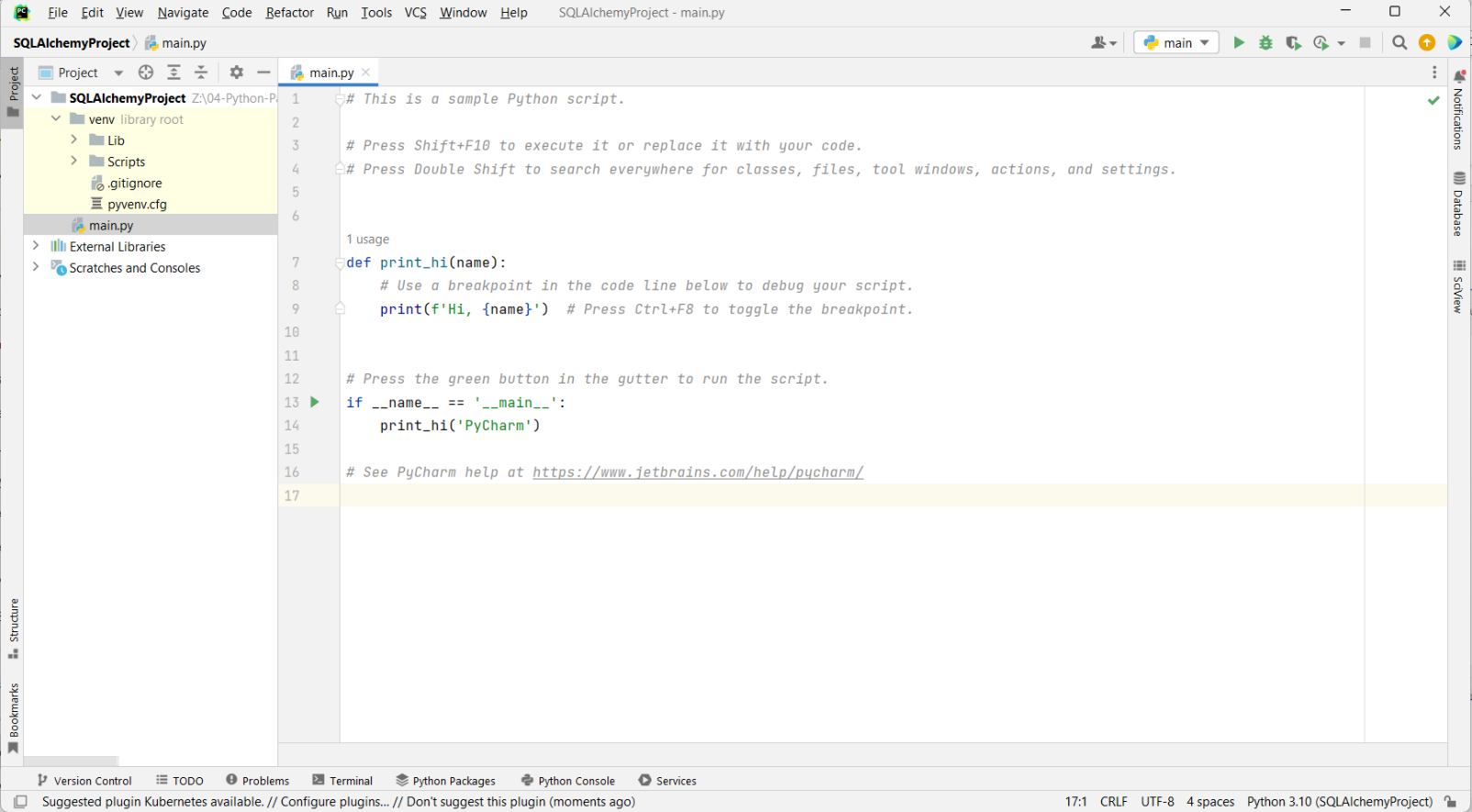
We are going to **create our first SQLAlchemy project**. SQLAlchemy **simplifies the process of working with databases** in Python by providing a set of tools and a **more Pythonic way to manage and query our data**. It is a powerful library for anyone who needs to work with databases in their Python applications.

To create the project, we will be using **PyCharm Professional** (you are free to **use** **PyCharm Community** as well) as our integrated development environment. Let us **launch the tool** and **navigate to File -> New Project**:

A screenshot of a computer

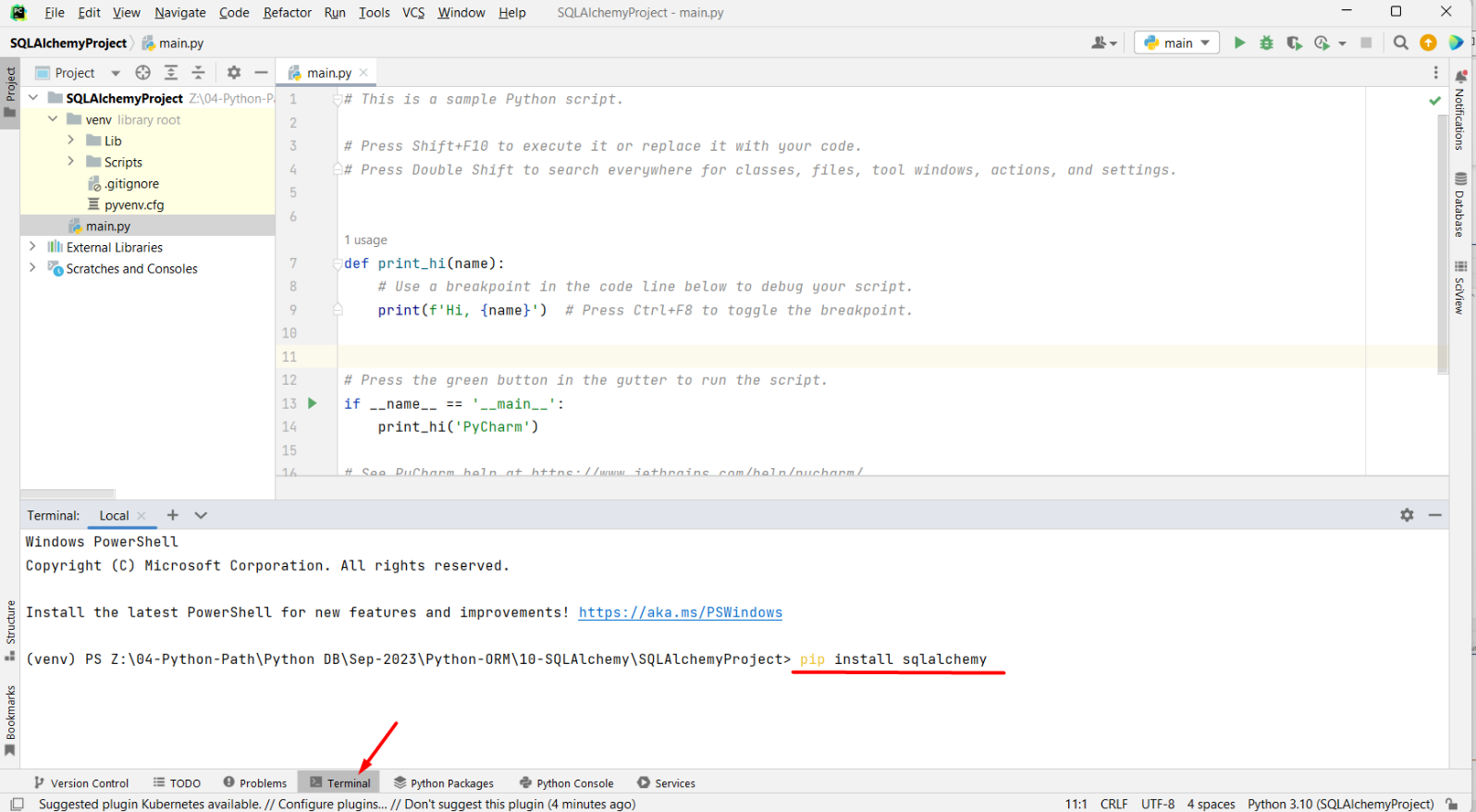
Description automatically generated

For this lab we will be using a **simple Python project**. **Configure the project's** **location** and **name**, ensuring the **virtual environment is set up** as well: 

Click the **"Create"** **button** to initiate the project creation process: 

Next, let us **make the needed installations:**

* **Install** **SQLAlchemy** using the command **"pip intsall sqlalchemy"**



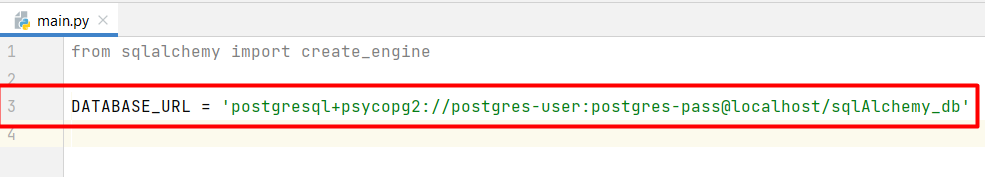
* Install the **PostgreSQL** **driver** (as we will be using PostgreSQL as our project database system) using the **command** **"pip install psycopg2"**.

**The main.py file we will use to import** the required **modules from the SQLAlchemy** library in Python. The **create\_engine** function is used to establish a **connection to the database**: A screenshot of a computer

Description automatically generated

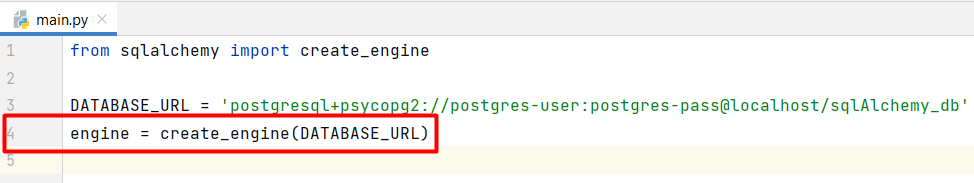
Now, let us use the **create\_engine** function to establish the connection to the PostgreSQL database.

First, we will create a string variable (constant) called **"DATABASE\_URL"** that holds the connection string for our PostgreSQL database:



* **"postgresql+psycopg2"** specifies that we are **connecting to PostgreSQL**. SQLAlchemy uses this information to determine the appropriate driver or connector to use.
* **"postgres-user"** is the **username** used to authenticate with the PostgreSQL database.
* **"postgres-pass"** is the **password** used to access the database.
* **"localhost"** is the **host where the PostgreSQL database is located**. In this case, we want to **use the local database** on our machine.
* **"sqlAlchemy\_db"** is the **name of the specific database** within the PostgreSQL server that we want to connect to. This is the database where our data is stored.

Next, we will call the **create\_engine** function and **pass the created variable** as an argument to **create the database engine**:



## Defining Models

It is time to **define our first model** using SQLAlchemy's ORM. First, let us **create a new Python file** called **models.py** where we will **store our models**. In this file we will **import** the **declarative\_base** function used to create a **base class** for defining the database tables as **Python classes**: A screenshot of a computer

Description automatically generated

We will use the created **"Base"** class as a parent for our database table classes. It **provides the necessary functionality** for mapping our Python class to a database table.

Now, let us define a **"User"** table with three columns: **"id"**, **"username"**, and **"email"**:

A screenshot of a computer program

Description automatically generated

* We **imported the classes** **and the** **types** needed to define the columns of our database table.
* Then, we defined a Python class called **"User"**, which **represents a database table** and inherits from the **"Base"** class.
* We added three columns to our table:
  + A **primary key** column named **"id"** with the data type **"Integer"** and set the **"primary\_key"** argument to **True**.
  + A **username** column of data type **"String"**.
  + An **email** column of data type **"String"**.

Before we create the table, we want to **specify the name of the database table** as **"users"**. We can do that by using the **\_\_tablename\_\_** attribute:

A screenshot of a computer program

Description automatically generated

After defining the table schema as a Python class, we will use the **create\_all** method to **create the corresponding table** in the database. We are passing the **"engine"** object (previously created with the database connection details) to the method. This action actually **generates the SQL statements** needed to **create the** **"users"** **table** in the database and **executes them**:

A screenshot of a computer program

Description automatically generated

Finally, we can **run** the **models.py** file and **see the changes to our database**:  
A screenshot of a computer

Description automatically generated

## Migrations

### 3.1 Install and Configure Alembic

As our application evolves, we may need to make changes to our database schema. These changes can include adding or modifying tables, columns, indexes, and relationships. Alembic provides a structured way to define these changes in Python code and apply them to the database, ensuring consistency across different database instances (e.g., development, staging, production).

Let us **install the Alembic package** in Python using the **command** **"pip install alembic"**:

A screenshot of a computer

Description automatically generated

However, we need to **initialize an Alembic migration environment** for our project. Let us type down the command **"alembic init alembic"**:

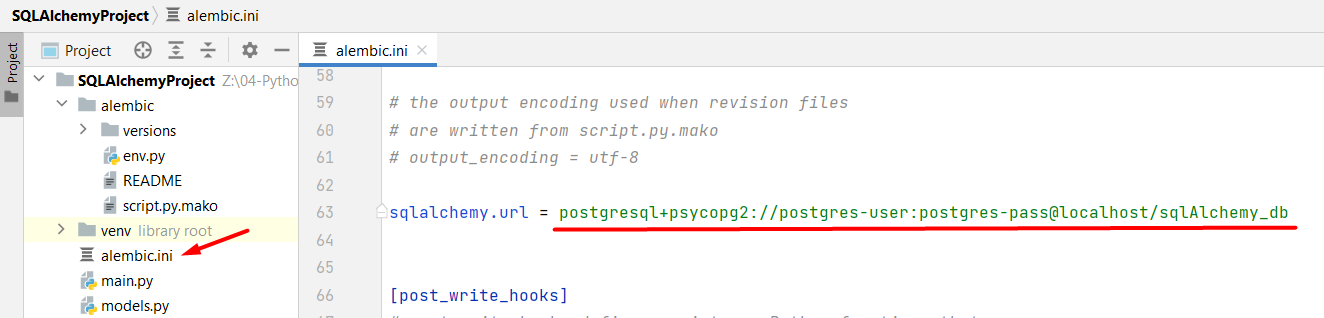
A screenshot of a computer

Description automatically generated

This way Alembic will create a directory named **"alembic"** in our project's root directory. Inside this directory, Alembic will **create several subdirectories** and **files**, including a **versions directory** for storing migration scripts, a **script.py.mako** **template file** for creating new migrations, and an **alembic.ini** **configuration file**:

A screenshot of a computer

Description automatically generated

The **alembic.ini** file is an important configuration file where we will **specify details about our database** **connection**, which will **allow Alembic to** **connect to our database and apply migrations**. Let us open the file and find the **sqlalchemy.url** configuration setting. We will change the connection string to the one we use for this project:  


The **env.py** file serves as the **configuration and entry point for managing database migrations** in a Python project. In this file, we will **specify the database schema** that Alembic **should compare against** when generating database migration scripts. We will set the **target\_metadata** variable to the metadata attribute of the **Base** class:

A screenshot of a computer

Description automatically generated

### 3.2 Using Alembic

Now, it is time to create a migration and apply it to the database. First, let us generate a new migration script in Alembic with the terminal command **"alembic revision --autogenerate -m "Add User Table"**:A screenshot of a computer

Description automatically generated

Here is a breakdown of what this command does:

* **"alembic revision"** - the Alembic command for **creating a new migration revision**.
* **"--autogenerate"** - the flag that tells Alembic to **automatically generate the migration script** by comparing the current state of the database to the state defined in our SQLAlchemy models.
* **"-m "Add User Table"** - the flag is used to **provide a message** for the migration.

When the **migration script is successfully generated**, it will appear in the **versions** directory:  
A screenshot of a computer

Description automatically generated

Next, let us apply the pending migration with the terminal command **"alembic upgrade head"**:A screenshot of a computer

Description automatically generated

Here is a breakdown of what this command does:

* **"alembic upgrade"** - the Alembic command used to **apply database migrations**.
* **"head"** - the keyword represents the **latest available version of the database schema**. This way we are telling Alembic to apply all pending migrations up to the latest version of the database schema.

Now, we can see **two tables** in our database:  
A screenshot of a computer

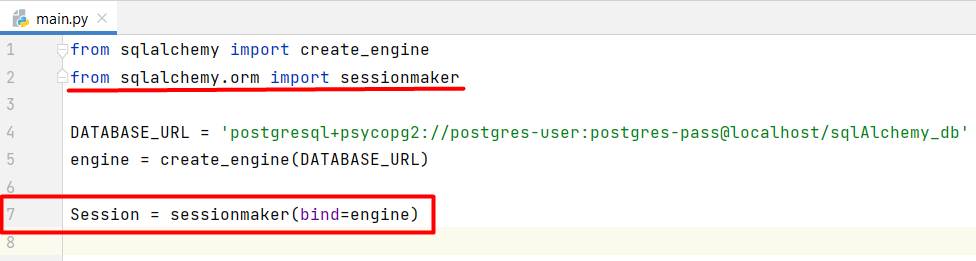
Description automatically generated

The **"alembic\_version"** table is a **system table** that Alembic automatically creates and uses to k**eep track of the current migration version** of our database. It is an essential part of Alembic's functionality for managing migrations. The table typically contains a single row with a version identifier.

## Queries and CRUD Operations

### 4.1. Creating a Session

When working with SQLAlchemy to create queries, it is crucial to **establish a session**. A session in SQLAlchemy is a **critical component for managing interactions with the database**. It helps to ensure that your **queries are executed** **consistently** and efficiently while providing a structured and controlled way to work with the database. Let us create a **Session** class in SQLAlchemy:



* The **sessionmaker** class is a factory for creating SQLAlchemy sessions.
* By binding the **sessionmaker** to the **engine**, we **associate the session with the** **sqlAlchemy\_db** database, so any operations performed through this session will be executed on that database.
* The **Session** class we created is essentially a **session factory**. You will use it to create individual sessions whenever we need to work with the database.

### 4.2. Perform Database Operations

Let us start by **creating a new Python file called** **"services"** where we will put queries and interactions with the database. Here we will write functions to perform various database operations. First, we will **import the** **"User"** **model** and the **"Session"** **factory class**:A screenshot of a computer

Description automatically generated

Now, we will **create a new session**, **create a new user object**, **add it** to that **session**, and then **commit the changes** to the database:

A screenshot of a computer program

Description automatically generated

Run the **services.py** **file to execute the code**. If we open the database table, we can **see the changes**: A screenshot of a computer

Description automatically generated

To **retrieve all data from a table**, we simply need to **use the** **all()** **method on a query** that is **constructed** **using the** **query()** method. The **query()** method is used to create a query that specifies which records we want to retrieve, and when we apply the **all()** method to that query, it **fetches all the records** matching our query criteria from the database:

A screenshot of a computer

Description automatically generated

To update a record from the database, first, we will need to get the object that we want to update. To get a user object in SQLAlchemy we can use the **filter\_by()** **method** to filter the recordsbased on the **username** **"john\_doe"** and then use **first()** method to **retrieve the first matching record**. It **retrieves the first matching** **user**, or it returns **None**:

A screenshot of a computer

Description automatically generated

If the **user object is found**, we will **assign the new email address** to the **email attribute** of the **user\_to\_update** object:

A screenshot of a computer program

Description automatically generated

We can **check the updated data** in the database table: A computer screen shot of a computer

Description automatically generated

Similar, to the process of updating, we will execute a **deletion using SQLAlchemy**. First, we need to **get the** **User** **object** we want to delete. Then, if the object is found, we will **use the** **delete()** method on the found object. It **does not immediately remove** the record from the database but **flags it for deletion** within the current session. The **actual deletion** of the record from the database **occurs when we** **commit the changes** using the **session.commit()** command:

A screenshot of a computer program

Description automatically generated

We can **check the changes made to the** database table: A screenshot of a computer

Description automatically generated

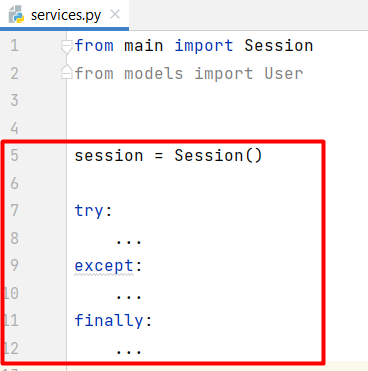
## Transactions

For this example, we will need to **populate the** **"users"** **table** with some data. **Hint**: **use the** **add\_all()** method:

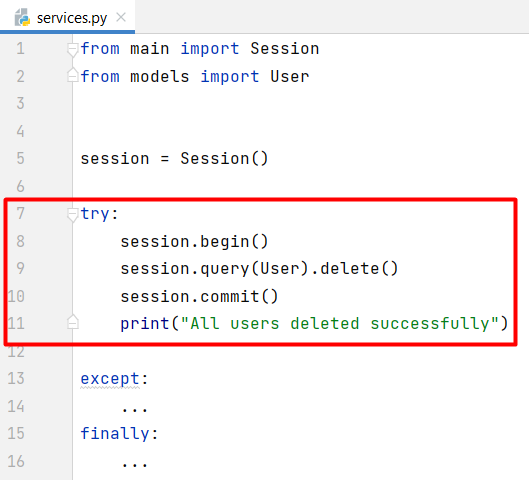
|  |  |
| --- | --- |
| **username** | **email** |
| john\_doe | john.doe@example.com |
| sarah\_smith | sarah.smith@gmail.com |
| mike\_jones | mike.jones@company.com |
| emma\_wilson | emma.wilson@domain.net |
| david\_brown | david.brown@email.org |

Now, let us write **our first transaction**. Transactions are often used to group **multiple database operations** into a single unit of work, ensuring that **all changes are committed together** or **rolled back in case of an error**.

First, we will **create a new session**. Then, we will create **try-except-finally blocks** to frame the transaction:



In the **try block**, we will **start a transaction** within the session. All the subsequent operations will be part of this transaction. Then, we will **construct a query to delete all records** from the **"User"** **table**. It is executed within the transaction:



In the **except block**, if an **Exception** **occurs** **during the execution** of the transaction, the **rollback()** method is called to **roll back the transaction** (any changes made within the transaction are undone) and the **error message** along with the details will be **printed**:

A screenshot of a computer program

Description automatically generated

The **finally block is** executed whether the try block completes successfully or an exception occurs. In it, the **session is closed** to release any resources and ensure proper cleanup:

A screenshot of a computer program

Description automatically generated

## Simple Relations

### 6.1. Defining a Relation

Let us **create a new model called** **"Order"** that establishes a **many-to-one relationship** with the **"User"** **model**. First, let us **create the model** with the main fields: **"id"** (Primary key field) and **"is\_completed"** (Boolean field with a default value set to False):

A screenshot of a computer program

Description automatically generated

Next, we will **establish a many-to-one relationship** between the **"Order"** model and the **"User"** model. We will use a **"user\_id"** **column to** **store the foreign key**, which is an efficient way to link orders to users. Then, we will add a **"user"** **relationship to specify the relation between the orders and the users**. It **does not add a column** to the **"orders"** **table,** but we **use it to access the associated "User" object**:

A screenshot of a computer program

Description automatically generated

Let us **check the resulting table** in the database:

A screenshot of a computer

Description automatically generated

### 6.2. Populate Order Table

Let us **populate the** **"Order"** **table**. First, we will need to **populate the** **"User"** **table** - we will use the same data as at the 5th bullet point above:

|  |  |
| --- | --- |
| **username** | **email** |
| john\_doe | john.doe@example.com |
| sarah\_smith | sarah.smith@gmail.com |
| mike\_jones | mike.jones@company.com |
| emma\_wilson | emma.wilson@domain.net |
| david\_brown | david.brown@email.org |

Since we have **added and deleted data multiple times**, the **user** **IDs may not begin from lower numbers** like 1, 2, or 3; instead, they might start from values such as 7, or 8. We can **check the IDs explicitly**, so we do not hit an error:

A screenshot of a computer

Description automatically generated

This time we will **add multiple data to the "orders" table**. For that, we will **use the add\_all() method**. In the orders table, there are 3 columns - the **"id"** column is **automatically populated** as it is a primary key, and the **"is\_completed"** column has a **default value**; so we **only need to explicitly set the** "**user\_id**" **values**:

A screenshot of a computer program

Description automatically generated

Let us execute the query and check the orders database table:

A screenshot of a computer

Description automatically generated

### 6.3. Queries for Relationships

Let us **retrieve data from the "orders" table** and **visualize it in a user-friendly way**. We want to **retrieve all orders**, **sort them in descending order** **by** **"user\_id"** and **return information** about the order ID, whether is it completed, and the username of the user related to that order in the format:

**"Order number {order.id}, Is completed: {order.is\_completed}, Username: {user.username}"**

A screen shot of a computer code

Description automatically generated

## Database Pooling

A connection pool **improves the efficiency and performance of database interactions** by maintaining a set of **connections that can be shared and reused**, rather than creating new connections for each request or operation. This is particularly valuable in scenarios where **multiple users or threads need to access the database concurrently**, as it helps **manage resources and reduces the overhead** associated with connection management.

Let us **configure a database connection pool** with an **initial size of** **10 connections** and the ability to **create up to 20 additional connections** temporarily if needed:  
A screenshot of a computer program

Description automatically generated