FastAPI and SQL (Relational) Databases

Introduction to SQLite

SQLAlchemy

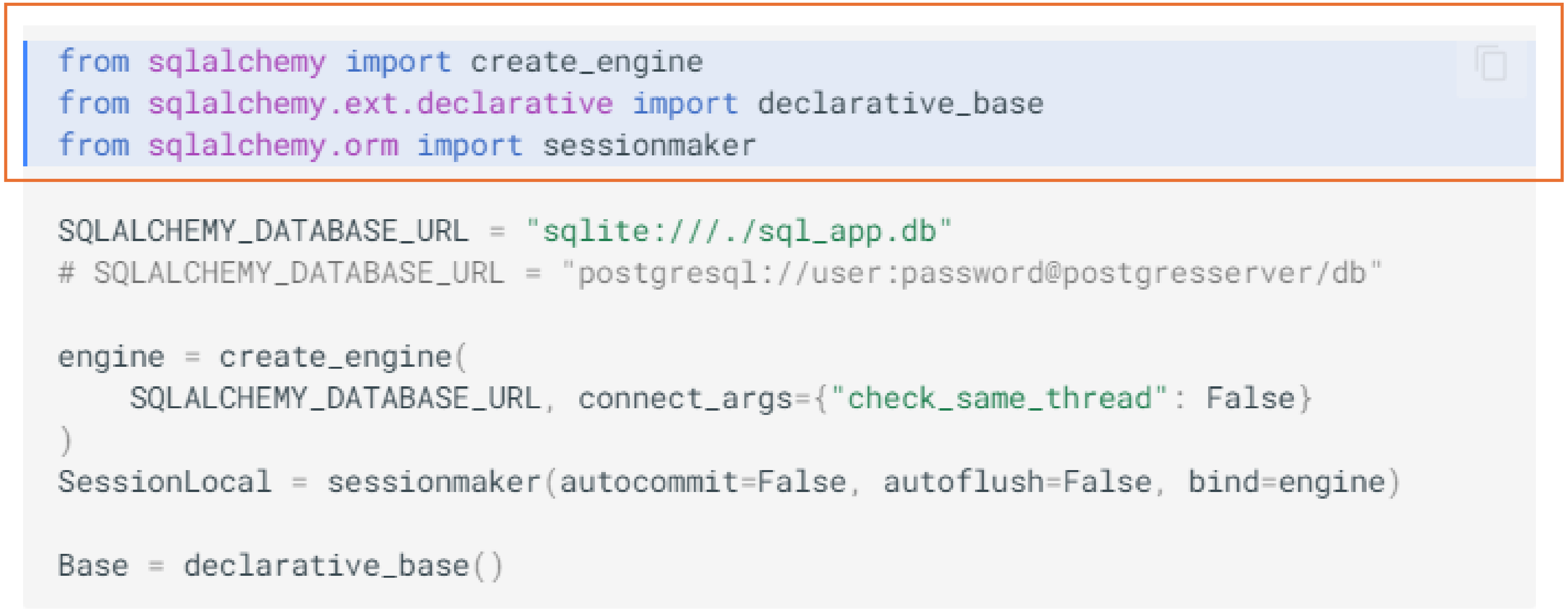
Databases Supported by SQLAlchemy

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server, etc.

Setup

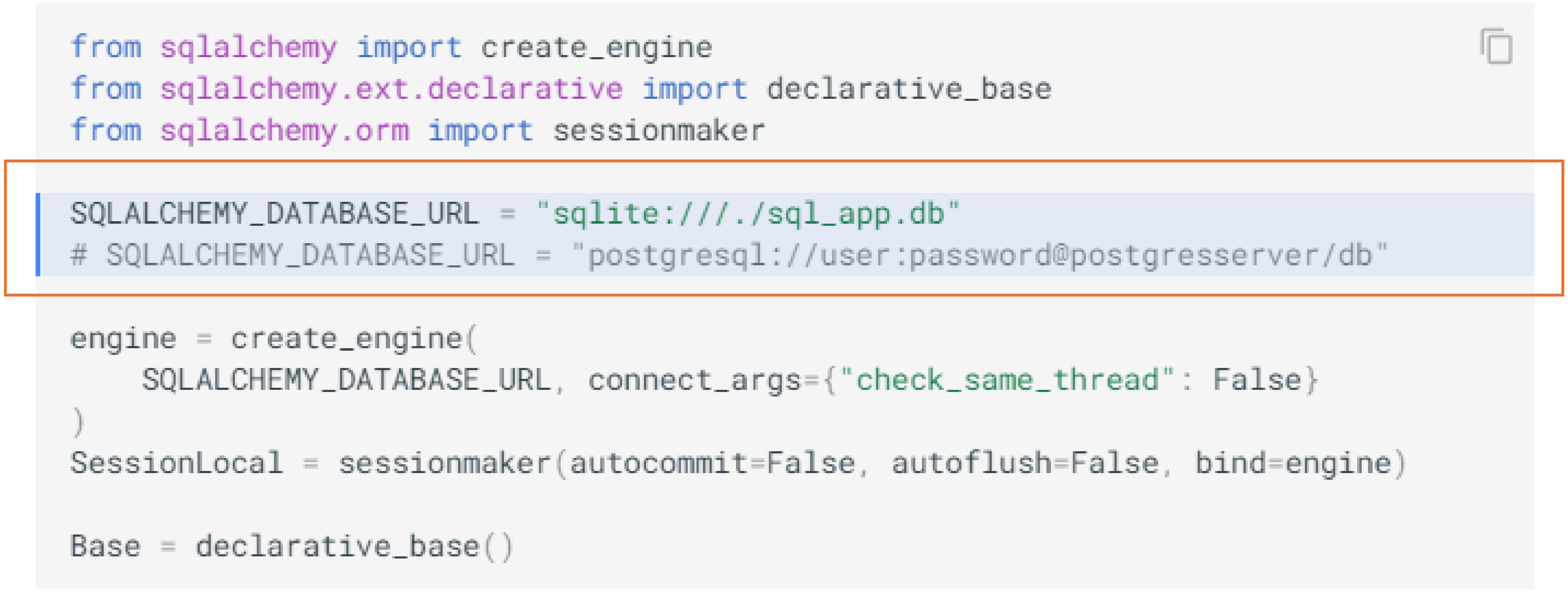
```
pip install sqlalchemy
```

Import the SQLAlchemy parts



```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Create a database URL for SQLAlchemy



```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

The file will be located at the same directory in the file `sql_app.db`.

That's why the last part is `./sql_app.db`.

If you were using a **PostgreSQL** database instead, you would just have to uncomment the line:

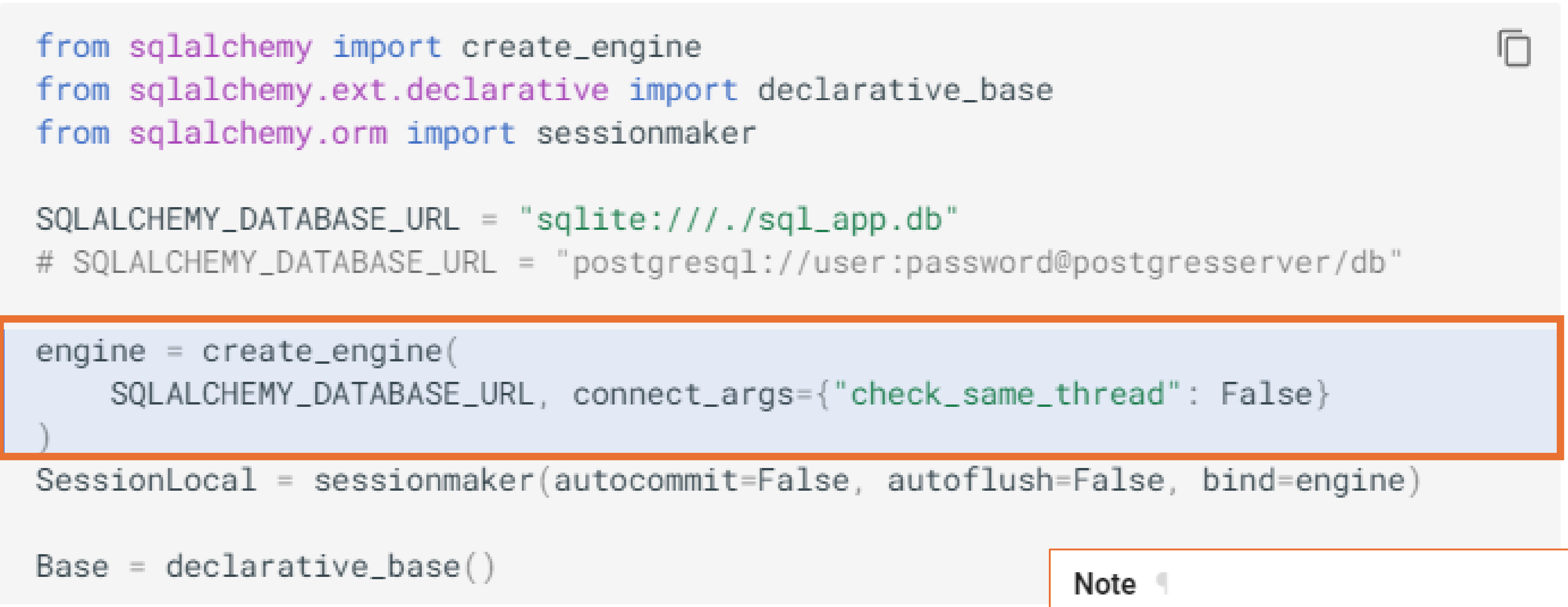
```
SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"
```

...and adapt it with your database data and credentials (equivalently for MySQL, MariaDB or any other).

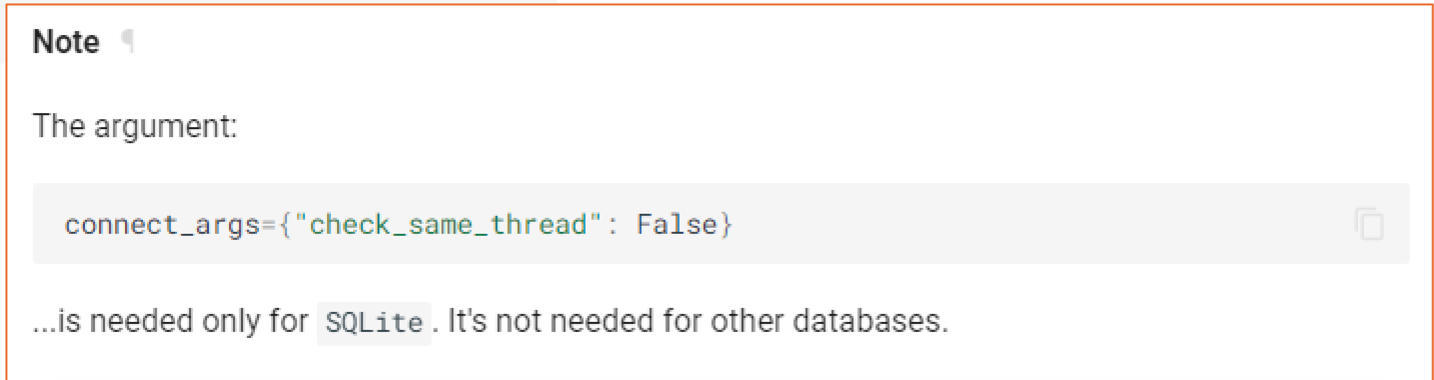
Create the SQLAlchemy engine

The first step is to create a SQLAlchemy "engine".

We will later use this engine in other places.



```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```



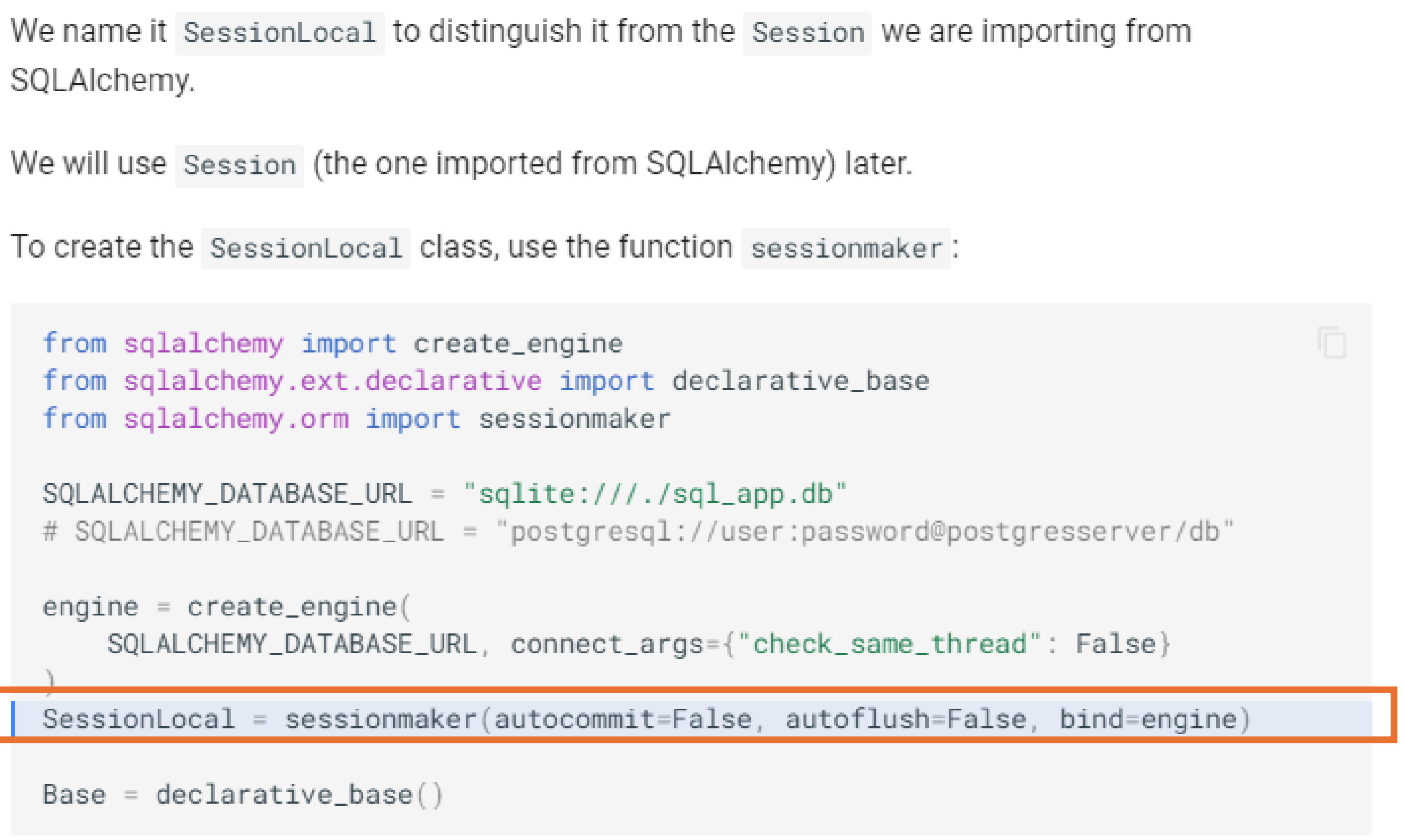
Note

The argument:

```
connect_args={"check_same_thread": False}
```

...is needed only for `SQLite`. It's not needed for other databases.

Create a SessionLocal class



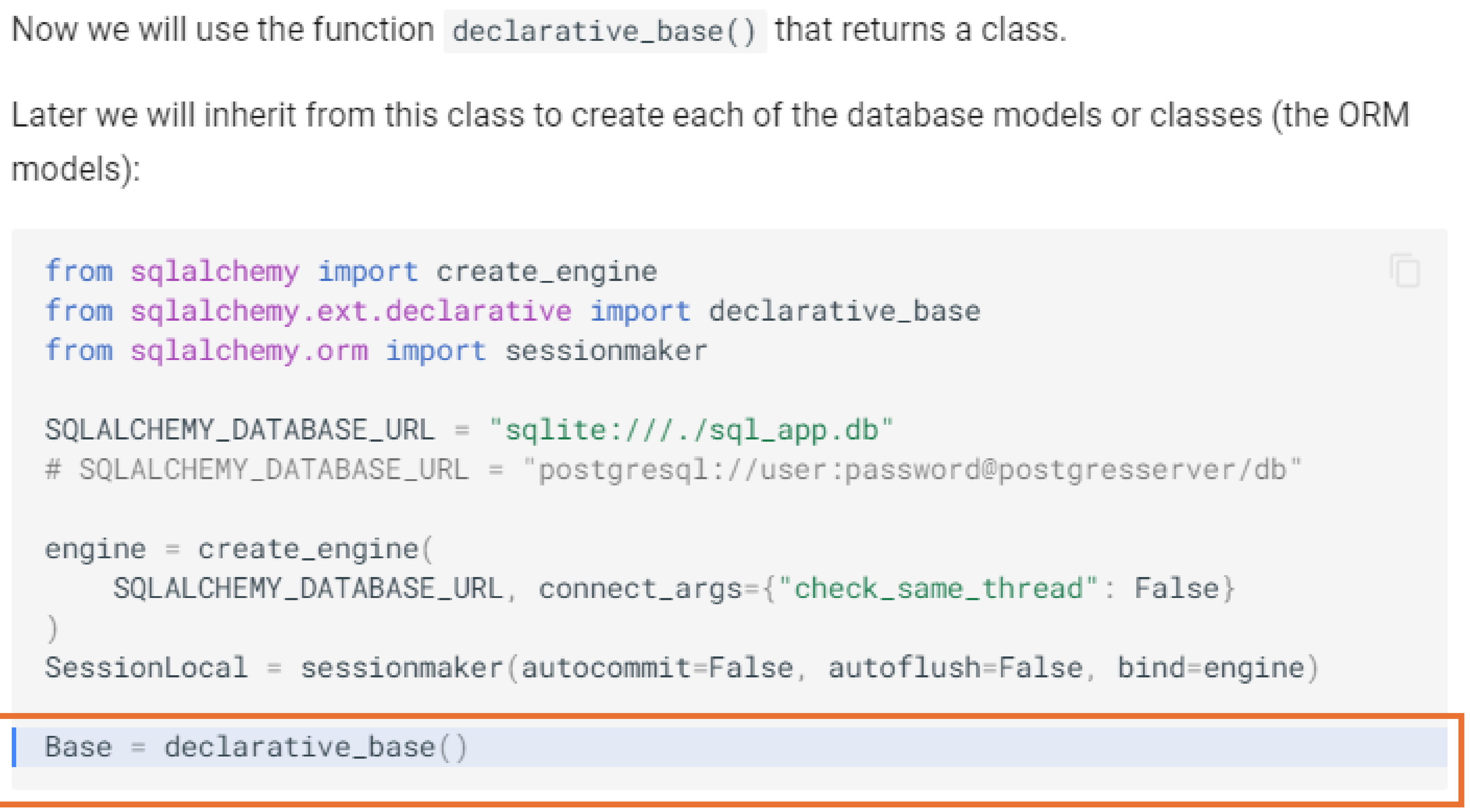
We name it `SessionLocal` to distinguish it from the `Session` we are importing from SQLAlchemy.

We will use `Session` (the one imported from SQLAlchemy) later.

To create the `SessionLocal` class, use the function `sessionmaker`:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Create a Base class

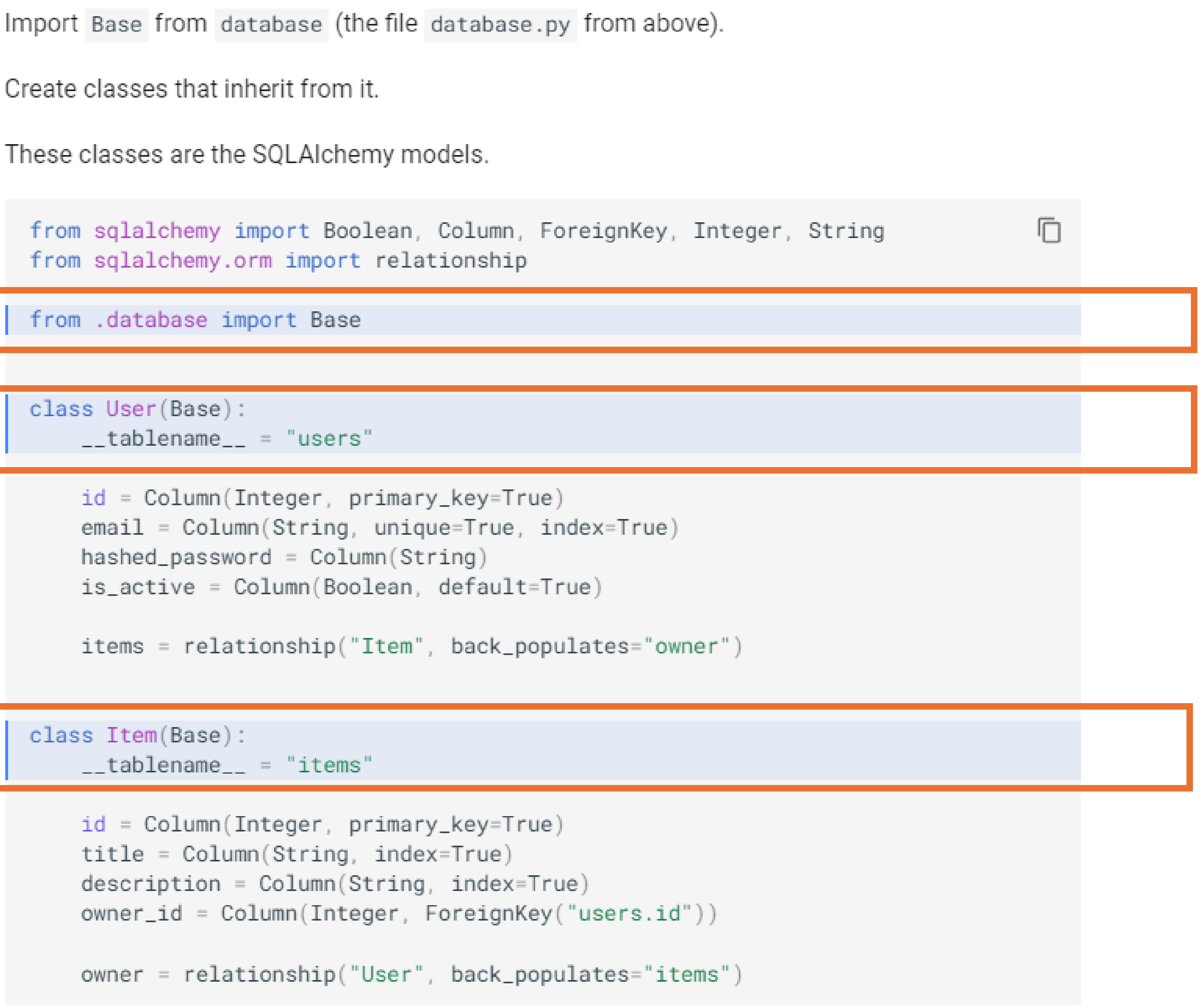


Now we will use the function `declarative_base()` that returns a class.

Later we will inherit from this class to create each of the database models or classes (the ORM models):

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Create SQLAlchemy models from the Base class



Import `Base` from `database` (the file `database.py` from above).

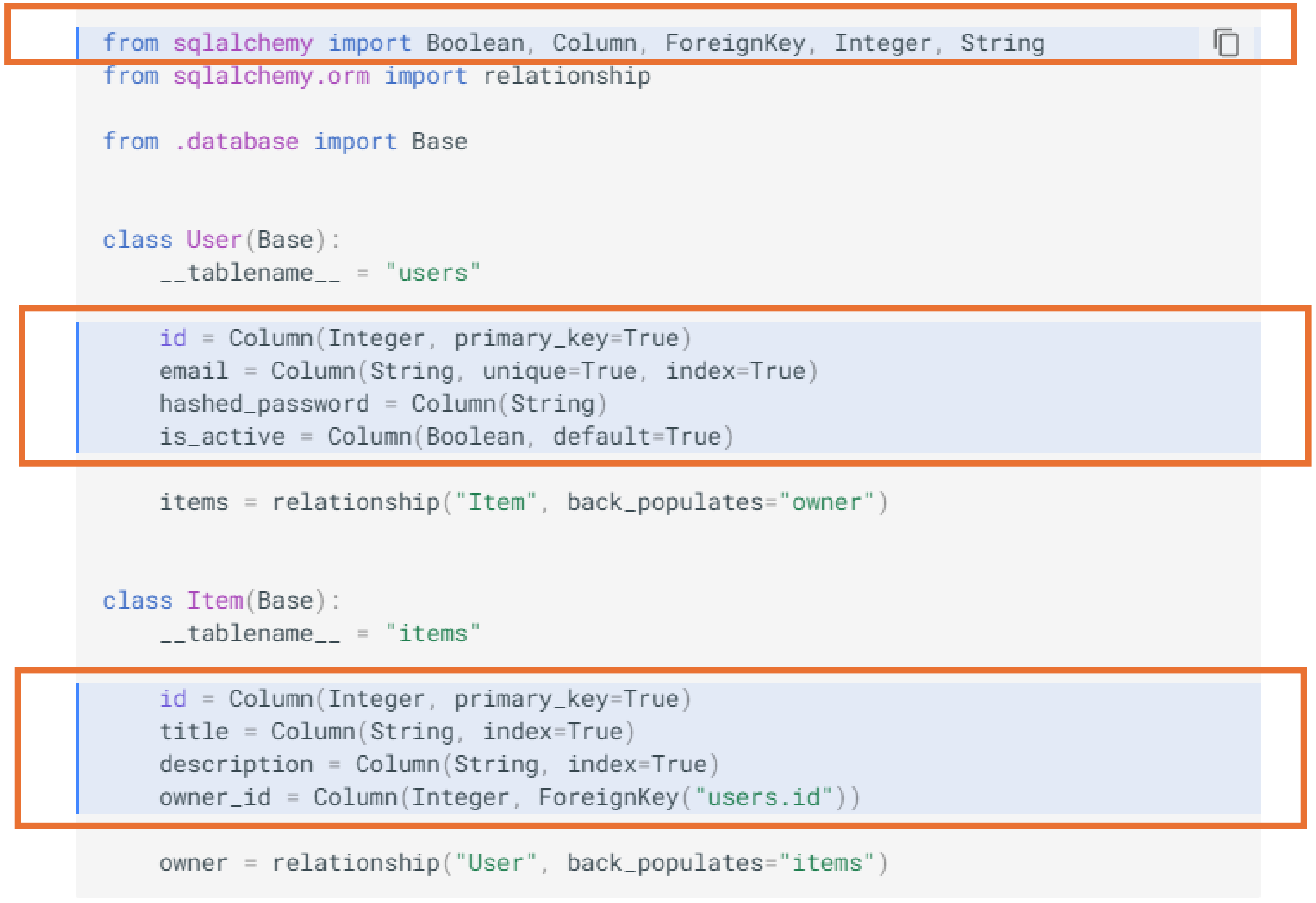
Create classes that inherit from it.

These classes are the SQLAlchemy models.

```python
from sqlalchemy import Boolean, Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship

from .database import Base


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

    items = relationship("Item", back_populates="owner")


class Item(Base):
    __tablename__ = "items"

    id = Column(Integer, primary_key=True)
    title = Column(String, index=True)
    description = Column(String, index=True)
    owner_id = Column(Integer, ForeignKey("users.id"))

    owner = relationship("User", back_populates="items")
```

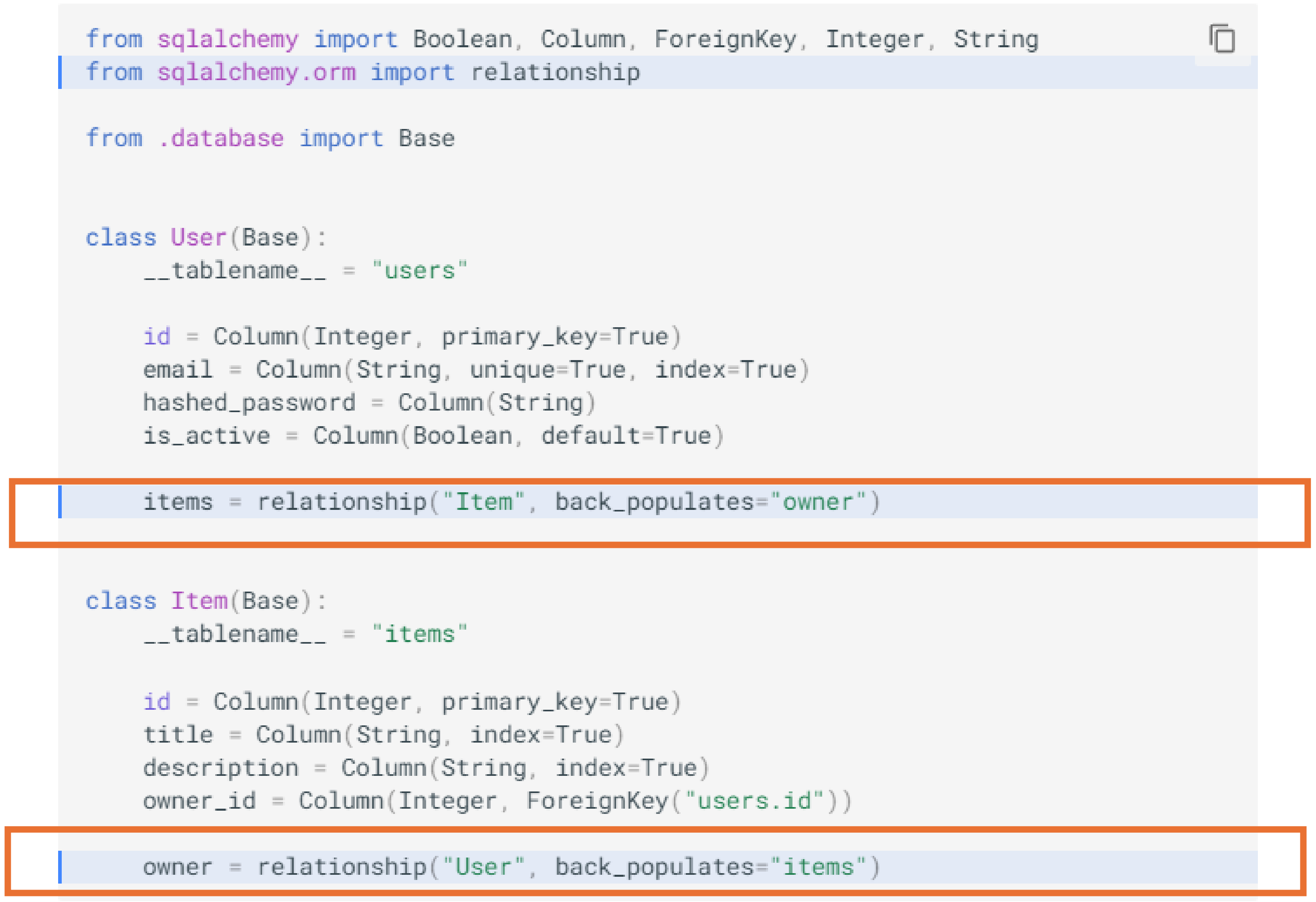
The __tablename__ attribute tells SQLAlchemy the name of the table to use in the database for each of these models.

Create model attributes/columns



```
from sqlalchemy import Boolean, Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship

from .database import Base


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

    items = relationship("Item", back_populates="owner")


class Item(Base):
    __tablename__ = "items"

    id = Column(Integer, primary_key=True)
    title = Column(String, index=True)
    description = Column(String, index=True)
    owner_id = Column(Integer, ForeignKey("users.id"))

    owner = relationship("User", back_populates="items")
```

Create the relationships



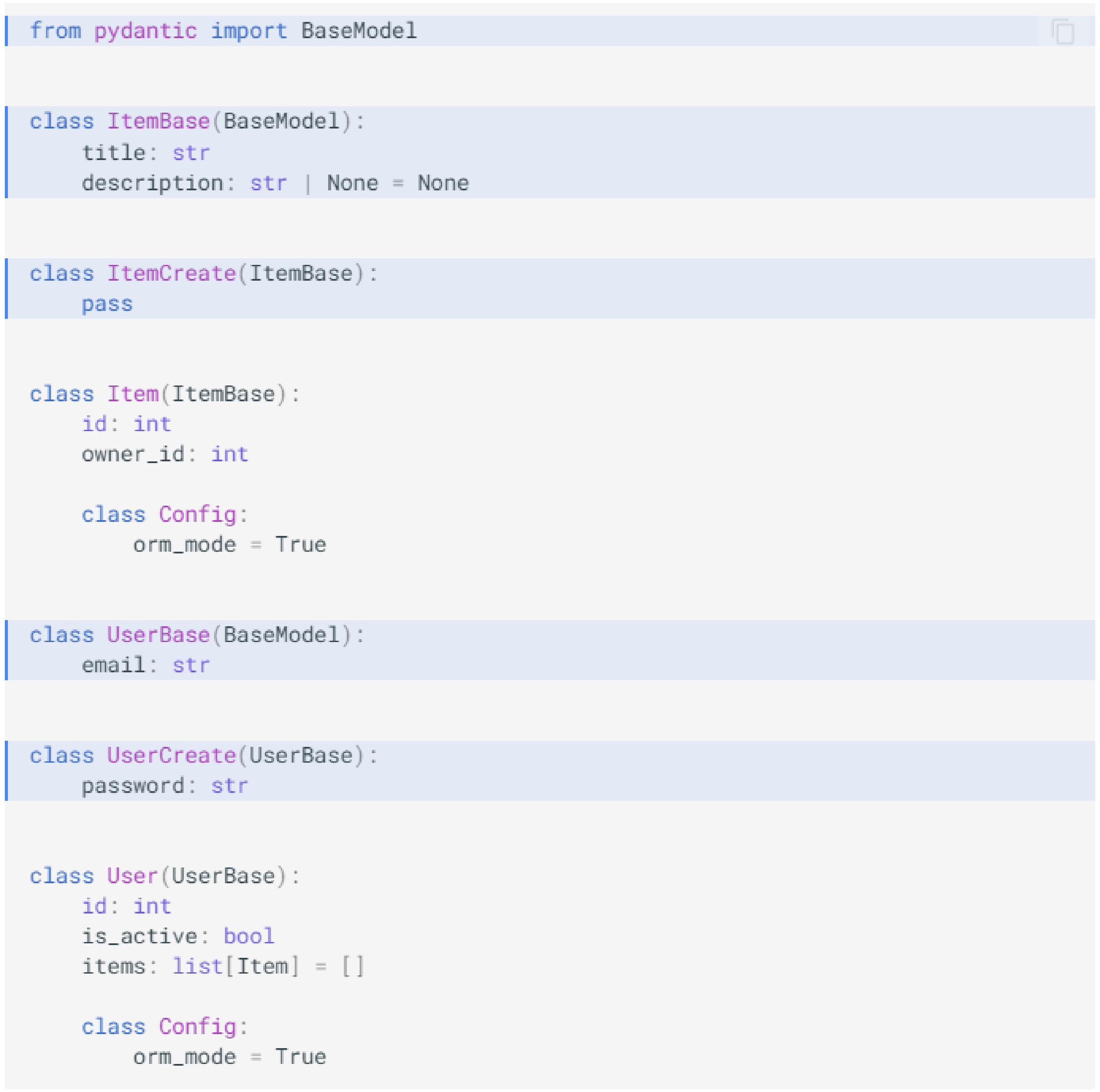
```python
from sqlalchemy import Boolean, Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship

from .database import Base


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

    items = relationship("Item", back_populates="owner")


class Item(Base):
    __tablename__ = "items"

    id = Column(Integer, primary_key=True)
    title = Column(String, index=True)
    description = Column(String, index=True)
    owner_id = Column(Integer, ForeignKey("users.id"))

    owner = relationship("User", back_populates="items")
```

When accessing the attribute `items` in a `User`, as in `my_user.items`, it will have a list of `Item` SQLAlchemy models (from the `items` table) that have a foreign key pointing to this record in the `users` table.

When you access `my_user.items`, SQLAlchemy will actually go and fetch the items from the database in the `items` table and populate them here.

And when accessing the attribute `owner` in an `Item`, it will contain a `User` SQLAlchemy model from the `users` table. It will use the `owner_id` attribute/column with its foreign key to know which record to get from the `users` table.

Create the Pydantic models



```python
from pydantic import BaseModel


class ItemBase(BaseModel):
    title: str
    description: str | None = None


class ItemCreate(ItemBase):
    pass


class Item(ItemBase):
    id: int
    owner_id: int

    class Config:
        orm_mode = True


class UserBase(BaseModel):
    email: str


class UserCreate(UserBase):
    password: str


class User(UserBase):
    id: int
    is_active: bool
    items: list[Item] = []

    class Config:
        orm_mode = True
```

SQLAlchemy style and Pydantic style

Notice that SQLAlchemy *models* define attributes using `=`, and pass the type as a parameter to `Column`, like in:

```
name = Column(String)
```

while Pydantic *models* declare the types using `:`, the new type annotation syntax/type hints:

```
name: str
```

Keep these in mind, so you don't get confused when using `=` and `:` with them.

Create initial Pydantic *models* / schemas

Create an `ItemBase` and `UserBase` Pydantic *models* (or let's say "schemas") to have common attributes while creating or reading data.

And create an `ItemCreate` and `UserCreate` that inherit from them (so they will have the same attributes), plus any additional data (attributes) needed for creation.

So, the user will also have a `password` when creating it.

But for security, the `password` won't be in other Pydantic *models*, for example, it won't be sent from the API when reading a user.

CRUD utils

CRUD comes from: **C**reate, **R**ead, **U**pdate, and **D**elete.

Read data

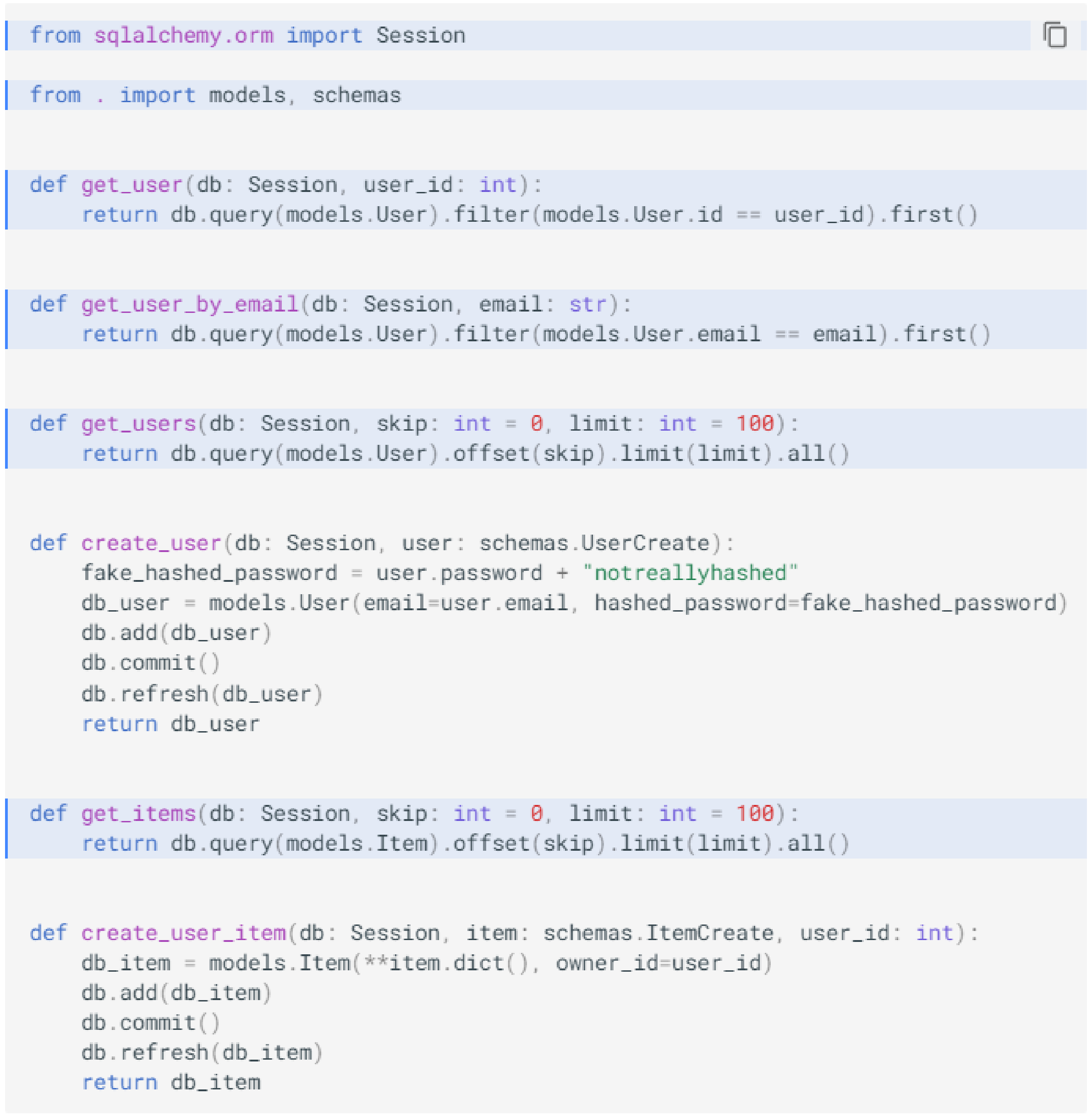
Import **Session** from sqlalchemy.orm, this will allow you to declare the type of the db parameters and have better type checks and completion in your functions.

Import **models** (the SQLAlchemy models) and **schemas** (the Pydantic models / schemas).

Create utility functions to:

- Read a single user by ID and by email.
- Read multiple users.
- Read multiple items.

Read data



```python
from sqlalchemy.orm import Session

from . import models, schemas


def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()


def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()


def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.User).offset(skip).limit(limit).all()


def create_user(db: Session, user: schemas.UserCreate):
    fake_hashed_password = user.password + "notreallyhashed"
    db_user = models.User(email=user.email, hashed_password=fake_hashed_password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user


def get_items(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Item).offset(skip).limit(limit).all()


def create_user_item(db: Session, item: schemas.ItemCreate, user_id: int):
    db_item = models.Item(**item.dict(), owner_id=user_id)
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

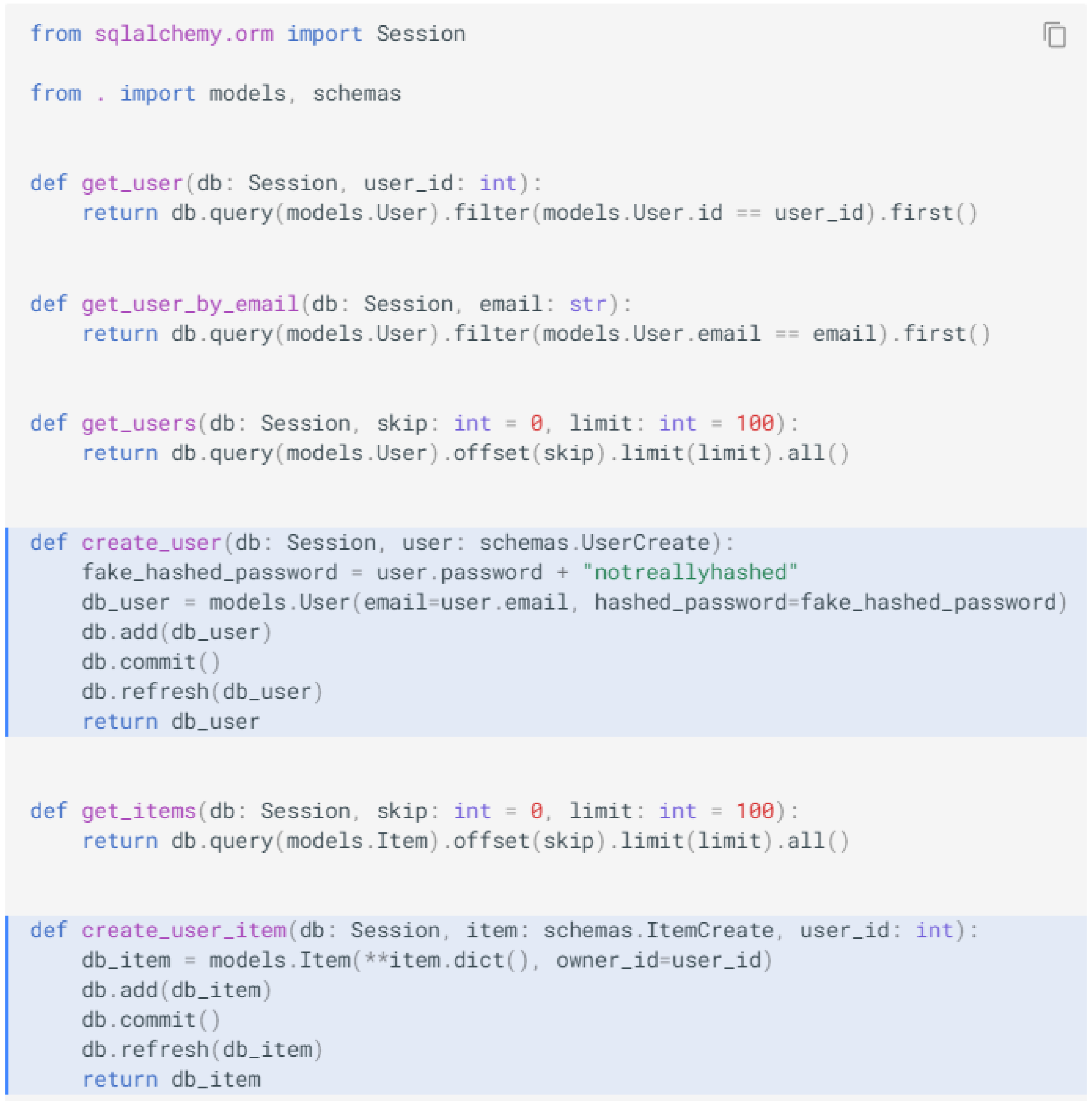
Create data

Now create utility functions to create data.

The steps are:

- Create a SQLAlchemy model instance with your data.
- add that instance object to your database session.
- **commit** the changes to the database (so that they are saved).
- refresh your instance (so that it contains any new data from the database, like the generated ID).

Create data



```python
from sqlalchemy.orm import Session

from . import models, schemas


def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()


def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()


def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.User).offset(skip).limit(limit).all()


def create_user(db: Session, user: schemas.UserCreate):
    fake_hashed_password = user.password + "notreallyhashed"
    db_user = models.User(email=user.email, hashed_password=fake_hashed_password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user


def get_items(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Item).offset(skip).limit(limit).all()


def create_user_item(db: Session, item: schemas.ItemCreate, user_id: int):
    db_item = models.Item(**item.dict(), owner_id=user_id)
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

RUN

```
uvicorn sql_app.main:app --reload
```