Activities 5: Working with Python classes mapped to database tables

Table of contents

Introduction

Theme: Using Python to work with data

Note: This topic relates to Coursework 2 and is not needed for Coursework 1.

This week introduces concepts related to classes and how they can be used when working with data from databases. You should work through all the activities, as the concepts build on each other and are designed to be learned progressively.

Each type of class discussed in Activities 1, 3, 4, and 8 has different syntax, which can seem confusing at first. However, in COMP0034 and COMP0035, you will primarily use SQLModel with an SQLite database, so focus on that syntax.

By the end of this week, you should understand:

- What a class is, including its attributes and methods
- What it means when one class inherits from another
- That ORM (Object-Relational Mapping) is a pattern for mapping Python classes to database tables

You should also be able to:

- Define a class using SQLModel that maps to a database table
- Create an SQLite database from classes defined using SQLModel

1. Python classes (5-01-class.md)
2. Class relationships - inheritance, composition (5-02-inheritance-composition.md)
3. Pydantic (5-03-pydantic.md)
4. ORM and SQLModel (5-04-orm-sqlmodel.md)
5. Using SQLModel to create SQLite database (5-05-sqlmodel-create-db.md)
6. Using SQLModel to add data (5-06-sqlmodel-add-data.md)
7. Summary (5-07-summary.md)
8. Optional Using SQLAlchemy instead of SQLModel (5-08-sqlalchemy.md)

1. Introduction to Python classes

A Python class is way of bundling data (attributes) and behaviour (methods) together. Classes are often described as a "blueprint" for creating objects. Objects specific instances of things, with actual data values for the attributes.

Consider the Paralympics scenario. Each Paralympic Games such as Paris 2024, includes multiple sports, such as Boccia. Each sport has multiple events, for example Boccia has "Men's individual BC1" or "Mixed Pairs BC4".

An event might have attributes such as:

- name
- sport
- category
- competing athletes

It may also have behaviours such as:

- describe the event
- register athletes

This can be represented as a Python class.

For example, *Men's 100m T54*, an athletics event for classification T54 (wheelchair racing), is a specific instance (object), of an *event* class.

A Python class to represent an event can be written as follows:

```
class ParalympicEvent:
    """ Represents a Paralympic event

     Attributes:
         name: A string representing the name of the event
         sport: An integer representing the sport that the event
        belongs to
         classification: An integer representing the event
        classification
         athletes: A list of strings representing the athletes that
        compete in the event

     Methods:
         describe() Prints a description of the event
         register_athlete() Adds an athlete to the list of athletes
     """

    def __init__(self, name, sport, classification):
        self.name = name
        self.sport = sport
        self.classification = classification
        self.athletes = []  # Empty list to hold athlete names

    def describe(self):
        """ Describes the event """
        print(f"{self.name} is a {self.sport} event for classification
        {self.classification}.")
        print("Athletes competing:", ", ".join(self.athletes))

    def register_athlete(self, athlete_name):
        """ Register the athlete with the event

        Args:
            athlete_name: A string representing the name of the athlete
        """
        self.athletes.append(athlete_name)
```

Key concepts in the class definition are:

- `class ParalympicEvent`: the keyword `class` defines this as a Python class. Class names typically start with a capital letter and use CamelCase.
- `docstring`: describes the class and its methods. This example uses the Google style of class docstring (https://google.github.io/styleguide/pyguide.html#384-classes).
- `def __init__(self, name, sport, classification, athletes)`: - `__init__` is a special method, the constructor method, used to initialize new objects from a class. When you create a new object for the class this lets you set the value of its attributes. The parameter `self` is how the current object refers to itself. It lets you access and modify its attributes (variables).
- `self.name, self.sport` are *attributes* of the class
- `describe()` and `register_athlets()` are methods of the class

This is a basic introduction to the core features of a class.

Creating objects from a class

To create an object to represent a specific event, pass values to the constructor.

```
event = ParalympicEvent(
    name="Men's individual BC1",
    sport="Boccia",
    classification="BC1",
)
```

Use dot notation to call methods:

```
event.describe()  # Should print the event description, "Athletes
        competing" will be empty
event.register_athlete("Sungjoon Jung")  # should register the athlete
event.describe()  # Should print the event again, "Athletes competing"
        should include Sungjoon Jung
```

Activity: Create an instance of a class and use its methods

1. The ParalympicEvent class is in starter_class.py (../../src/activities/starter/starter_class.py).
2. Use the class to create an instance for a given event, and register an athlete.

Activity: Create an Athlete class

1. An athlete likely has more detail than just their name.
2. Create a class *Athlete* with attributes such as name, the team they represent and their disability classification.
3. Add a method that prints the athlete's details. Instead of `describe()`, create a string representation of the class. To do this you can overwrite the default method by defining a method called `__str__(self)`. See example here (https://www.codecademy.com/resources/docs/python/dunder-methods/str).
4. Create an instance of the class and print it.

Next activity (5-02-inheritance-composition.md)

2. Class inheritance and composition

Class inheritance

Inheritance is a feature in object-oriented programming that allows one class (called a **child** or **subclass**) to inherit attributes and methods from another class (called a **parent** or **superclass**).

Benefits of inheritance include:

- Reuse code across related classes
- Organize code more logically
- Extend functionality without rewriting everything

For example, you have an Athlete class. Now you want to add a more specific class Runner that inherits from it and adds specific data and methods that are only relevant for runner:

```
# Base class
class Athlete:
    def __init__(self, first_name, last_name, team_code,
        disability_class):
        self.first_name = first_name
        self.last_name = last_name
        self.team_code = team_code
        self.disability_class = disability_class

    def introduce(self):
        print(f"{self.first_name} {self.last_name} represents
        {self.team_code} in class {self.disability_class}.")


# Subclass
class Runner(Athlete):
    def __init__(self, first_name, last_name, team_code,
        disability_class, distance):
        super().__init__(first_name, last_name, team_code,
        disability_class)
        self.distance = distance  # e.g., 100m, 400m

    def race_info(self):
        print(f"{self.first_name} is running the {self.distance}
        race.")


# Example usage
runner1 = Runner("Li", "Na", "CHN", "T12", "100m")
runner1.introduce()  # Inherited method
runner1.race_info()  # Subclass-specific method
```

Key concepts in the code:

- `class Runner(Athlete)`: the Runner class inheirts from Athlete. A class can inherit from more than one class e.g. `class Runner(Athlete, TeamMember)`.
- `super().__init__()`: calls the constructor of the parent class to initialise inherited attributes.

- The Runner subclass now has access to the args and methods of the Athlete class and can also add its own specific attributes (e.g. `distance`), and methods (e.g. `race_info()`).

Class composition

Composition is a design principle where a class is made up of one or more objects from other classes, rather than inheriting from them.

For example, an Athlete can have a list of Medal objects.

```
from dataclasses import dataclass
from datetime import date
from typing import List


@dataclass
class Medal:
    type: str
    design: str
    date_designed: date


class Athlete:
    def __init__(self, first_name: str, last_name: str, team_code: str,
        disability_class: str, medals: List[Medal]):
        self.first_name = first_name
        self.last_name = last_name
        self.team_code = team_code
        self.disability_class = disability_class
        self.medals = medals  # Composition: Athlete has Medals
```

Note: The code above include type annotations (https://typing.python.org/en/latest/spec/annotations.html). Type annotations (type hints) are an optional notation in Python that specifies te tyoe of a parameter or function result. It tells the programme using the function or class what kind of data to pass and what kind of data to expect when a value is returned.

To create an athlete with medals:

```
# Create medals
medal1 = Medal("gold", "Paris 2024 design", date(2023, 7, 1))
medal2 = Medal("silver", "Tokyo 2020 design", date(2019, 8, 25))

# Create an athlete with medals
athlete = Athlete(
    first_name="Wei",
    last_name="Wang",
    team_code="CHN",
    disability_class="T54",
    medals=[medal1, medal2]
)

print(athlete)
```

Activity

1. Modify your `Athlete` class to include a list of `Medal` objects as an attribute.

Next activity (5-03-pydantic.md)

3. Pydantic classes

Pydantic (https://docs.pydantic.dev/latest/) is a Python library used for data validation and settings management using Python type annotations (https://typing.python.org/en/latest/spec/annotations.html). It's widely used in modern Python projects, particularly with frameworks like FastAPI.

You will be using pydantic and FastAPI in COMP0034.

Key features of Pydantic:

- Validates data: Ensures that the data you receive from users, APIs, files, etc. matches the types and constraints you specify.
- Parses data: Converts input data into Python objects with the correct types.
- Automatic error messages: Provides clear feedback when data is invalid.

To use Pydantic, your class should inherit it's `BaseModel` class. This inheritance gives your class all the attributes and methods (https://docs.pydantic.dev/latest/concepts/models/#model-methods-and-properties) of the Pydantic BaseModel.

Classes defined using Pydantic are typically referred to as **models**. A Pydantic model is simply a Python class that:

- Inherits from BaseModel
- Uses type annotations to define fields

Note that pydantic syntax varies for different versions of pydantic and different versions of Python. Use the latest pydantic documentation (https://docs.pydantic.dev/latest/) rather than other sources of information.

The `Athlete` class is rewritten as a Pydantic model class:

```
from pydantic import BaseModel


class Athlete(BaseModel):
    first_name: str
    last_name: str
    team_code: str
    disability_class: str
    medals: list[Medal]
```

Pydantic classes can contain methods:

```
from pydantic import BaseModel


class Athlete(BaseModel):
    first_name: str
    last_name: str
    team_code: str
    disability_class: str
    medals: list[Medal]

    def introduce(self) -> str:
        return f"{self.first_name} {self.last_name} represents
        {self.team_code} in class {self.disability_class}."
```

The function signature `def introduce(self) -> str:` uses Python's type hinting to specify that the function returns a string. This is optional but improves readability and clarity.

You can find a version of the classes with Pydantic in starter_pydantic.py (../../src/activities/starter/starter_pydantic.py)

Pydantic validation

You can specify optional fields and default values using type hints and assignment:

```
class Athlete(BaseModel):
    first_name: str  # Must be provided
    last_name: str  # Must be provided
    team_code: str | None  # Optional, can be None
    disability_class: str | None  # Optional, can be None
    medals: list[Medal] = []  # Set to empty as default
```

Activity: Add validation to the Athlete

1. Make a copy of starter_pydantic.py (../../src/activities/starter/starter_pydantic.py)
2. Add validation to the `Athlete` and optionally to the `Medal` class.
3. Create an instance with valid data e.g.
    - create an `athlete` with no medals
    - create a new medal
    - add the new medal to the athlete's medals list
      e.g. `athlete.medals.append(new_medal)`

    Examples of real para athletes from Paris 2024:

    - Yuyan Jiang from team CHN People's Republic of China won 7 gold medals
    - Catherine Debrunner from ITA Italy won 5 gold and 1 bronze
    - Bianka Pap from HUN Hungary won 1 gold, 1 silver and 1 bronze
4. Create an instance with invalid data. Try with and without the use of `try/except` with `ValidationError:` to catch validation errors.

```
from pydantic import ValidationError

 try:
    bp = Athlete(first_name="Bianka", medals=1)
except ValidationError as e:
    print(e.errors())
```

Next activity (5-04-orm-sqlmodel.md)

4. Object relational mapping (ORM)

So far you have learned how to:

- access and manipulate data in a pandas DataFrame
- create a database with sqlite3 and SQL statements, and how to add data (if you completed optional activities in week 3)
- create a Python class

You can use the values from a DataFrame or results from a sqlite3 query to create instances of Python classes. However, this often involves a lot of manual code and requires knowledge of both SQL and Python.

Object-Relational Mapping (ORM) is a technique that lets you interact with a database using Python objects, so you don't have to write raw SQL queries.

Popular Python ORM libraries include:

- SQLAlchemy (https://docs.sqlalchemy.org/en/20/)
- SQLModel (https://sqlmodel.tiangolo.com/)

The libraries aim to make database operations more intuitive and help keep your code cleaner and easier to maintain.

Previously, COMP0035 used SQLAlchemy. While powerful, it can be complex for beginners. This year, we will use SQLModel instead. SQLModel, which is built on top of Pydantic and SQLAlchemy, uses Python type annotations for simplicity.

For example, in week 3, you designed the Paralympic Games table like this:

Games			
int	id	PK	NOT NULL, UNIQUE
int	type		CHECK (type IN ('winter', 'summer'))
int	year		
date	start		
date	end		
int	countries		
int	events		
int	sports		
int	participants_m		
int	participants_f		
int	participants		
string	highlights		
string	URL		

As with Pydantic, SQLModel classes are referred to as **models**. Using SQLModel, a Python model class that directly maps to the Games table could be written like this:

```
from datetime import date

from sqlmodel import Field, SQLModel


class Games(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    type: str = Field()
    year: int
    start: date
    end: date
    countries: int
    events: int
    sports: int
    participants_m: int
    participants_f: int
    participants: int
    highlights: str
    URL: str
```

Key concepts:

- `class Games(SQLModel, table=True):` - inherit the SQLModel class. SQLModel inherits Pydantic BaseModel and SQLAlchemy so implicitly inherit these when you inherit SQLModel. `table=True` indicates that this class also defines a database table.
- attributes are defined similarly to pydantic. The `Field()` class can optionally be used to define the attributes to indicate constrains such as key fields and other validation rules.
- `int | None`: a PK is always required and can't be NULL so why declare it as optional? The `id` is generated by the database, not by your code. When you create an instance of the class, `id` will be `None` until the object is saved to the database and the database assigns a value.

Activity: Write SQLModel classes

1. Create a copy of starter_sqlmodel.py (../../src/activities/starter/starter_sqlmodel.py) and name it `models.py`. The name is not crucial but is a name you will see used often when you start to create web apps using the SQLModel classes.
2. Write SQLModel classes for the following tables:

Country		
int	id	PK
string	country	

Disability		
int	id	PK
string	description	

Team			
string	code	PK	
string	name		
string	region		CHECK (region IN ('Asia', 'Europe', 'Africa', 'America', 'Oceania'))
string	sub_region		
string	member_type		CHECK (member_type IN ('country', 'team', 'dissolved', 'construct'))
string	notes		
int	country_id	FK	ON UPDATE CASCADE, ON DELETE SET NULL

You may want to refer to the SQLModel tutorial (https://sqlmodel.tiangolo.com/tutorial/create-db-and-table/#create-the-table-model-class) for help.

Next activity (5-05-sqlmodel-create-db.md)

5. Using SQLModel to create an SQLite database

Classes that are defined with the argument `table=True` can be created in a database with a few lines of code.

```
from sqlmodel import SQLModel, create_engine

# 1. Define the classes (code not shown here)

# 2. Create a connection to a SQLlite database, replace
        'database_name.db' with path to the database and its name
engine = create_engine("sqlite:///database_name.db")

# 3. Create all the tables in the database
SQLModel.metadata.create_all(engine)
```

The following example is from the SQLModel documentation:

```
from __future__ import annotations
from sqlmodel import Field, SQLModel, create_engine


class Hero(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str
    secret_name: str
    age: int | None = None


engine = create_engine("sqlite:///hero.db")

SQLModel.metadata.create_all(engine)
```

Code files and structure

Typically, you will see model classes saved in a file called `models.py`.

The code to create the database using these models is likely in another file, e.g. `database.py`.

The code to run the app would call the function to create the database using the models. This is likely to be in an `app.py` or `main.py` module.

The structure of this code is shown in the SQLModel tutorial here (https://sqlmodel.tiangolo.com/tutorial/code-structure/#single-module-for-models).

Activity: Create the database using this code structure

Write code to create the paralympics database separated into `models.py`, `database.py` and `app.py` files (modules).

You can copy the model code from starter_models.py (../../src/activities/starter/starter_models.py) to `models.py`.

Write code in `database.py` and `app.py` to create the database. Use a new database name `paralympics_sqlmodel.db` as you probably already have one called `paralympics.db` from an earlier activity.

Next activity (5-06-sqlmodel-add-data.md)

The section below is optional, you can skip if you wish.

Using copilot to generate the SQLModels from a sql schema

This is a reflection on my experience of using copilot to generate SQLModels from a SQL schema.

I generated the initial version of the SQLModel classes using copilot in Pycharm with paralympics_schema.sql attached and the prompt: "Based on the .sql schema, write SQLModel classes"

The generated code was then checked against current SQLModel syntax:

- https://sqlmodel.tiangolo.com/tutorial/create-db-and-table/ to define a class (for a table)
- https://sqlmodel.tiangolo.com/tutorial/relationship-attributes/ for relationship attributes

Limitations of the initial copilot-generated solution:

- The attributes for date were treated as string. I decided to leave this since SQLite store a date as a string.
- Foreign key constraints were not recognised. The SQLModel documentation (https://sqlmodel.tiangolo.com/tutorial/relationship-attributes/cascade-delete-relationships/#ondelete-options) explains how to include these if you need to use them.
- Check constraints were not recognised. These are not directly supported in SQLModel in the current version, SQLAlchemy syntax was needed. I did not find this in the SQLModel documentation. The SQLAlchemy documentation can be complex to follow so this article has a clear and shorter summary: https://plainenglish.io/blog/creating-table-constraints-with-sqlmodel

The following shows one solution for the check constraint:

```
from typing import Optional

from sqlalchemy import CheckConstraint
from sqlmodel import Field, SQLModel


class Team(SQLModel, table=True):
    code: str = Field(primary_key=True)
    name: str
    region: Optional[str]
    sub_region: Optional[str]
    member_type: Optional[str]
    notes: Optional[str]
    country_id: Optional[int] = Field(default=None,
        foreign_key="country.id")

    __table_args__ = (
        CheckConstraint("member_type IN ('country', 'team',
        'dissolved', 'construct')"),
        CheckConstraint("region IN ('Asia', 'Europe', 'Africa',
        'America', 'Oceania')")
    )
```

6. Using SQLModel to add data to an SQLite database

To add a single record to a database table is conceptually similar to the code used with sqlite:

- create the instance
- add it
- commit it

```
hero_1 = Hero(name="Deadpond", secret_name="Dive Wilson")
hero_2 = Hero(name="Spider-Boy", secret_name="Pedro Parqueador")

engine = create_engine("sqlite:///database.db")

with Session(engine) as session:
    session.add(hero_1)
    session.add(hero_2)
    session.commit()
```

To add data to the database, you can read the values from the data file and create object instances. One way to do this is to read the data into a pandas DataFrame first, for example:

```
import pandas as pd
from sqlmodel import Session, SQLModel, create_engine
from mymodels import MyData  # Replace with your actual model

# 1. Read Excel file
df = pd.read_excel("your_data.xlsx")

# 2. Convert DataFrame rows of values to SQLModel instances
records = []
for _, row in df.iterrows():
    record = MyData(**row.to_dict())
    records.append(record)

# 3. Insert into database
engine = create_engine("sqlite:///mydatabase.db")
with Session(engine) as session:
    session.add_all(records)
    session.commit()
```

Adding data to the paralympics database is more complex than the example above due to the number of tables and the relationships. This is skipped for now and covered in week 8.

Next activity (5-07-summary.md)

7. Summary

Many concepts have been introduced this week. These are key foundations that will support your design work over the next five weeks and your application development work next term.

If you are unsure of the concepts, please ask questions during office hours or in a tutorial; or carry out your own research, there are many helpful tutorials online.

Key concepts to understand:

- A Python **class** models a real world object. It has:
    - **attributes** to store data
    - **methods** to define behaviour.
    - A class can have just attributes. An **object** is one instance of a class that has been created.
- **Inheritance** allows one class to inherit the attributes and methods from another, whilst also implementing its own.
- **Pydantic** is a Python library for **data validation**. To use it, a class inherits from `BaseModel`. Validation is important when building apps that handle user or external data.
- **Object-Relational Mapping (ORM)** is a technique that lets you interact with a database using Python objects instead of writing raw SQL. ORMs map database tables to classes and rows to objects, making data handling more intuitive.
- **SQLModel** is ORM that is build on Pydantic and SQLAlchemy.
- In Pydantic and SQLModel, a **model** is a Python class that defines the structure and types of data. It 'models' the data that an application will use.
- Python classes, Pydantic models, and SQLModel classes vary in syntax, which can be confusing. Focus on learning SQLModel syntax, as that will be most relevant for your coursework.

Note: for coursework 2 you are welcome to use SQLAlchemy instead of SQLModel if you prefer. For those who wish to do so the next activity (5-08-sqlalchemy.md) provides an overview.

8 Optional: Using SQLAlchemy instead of SQLModel

For the coursework you can use SQLAlchemy instead of SQLModel, particularly if you already know SQLAlchemy.

Define a SQLAlchemy model class

SQLAlchemy supports different syntax or styles of mapping Python classes to database tables (https://docs.sqlalchemy.org/en/20/orm/mapping_styles.html): declarative style or imperative style. Imperative style is the original syntax and is used in some older tutorials. The declarative style can also be used with Python type annotations.

The following are examples so you recognise the differences.

Declarative (with type annotations)

First the declarative style (https://docs.sqlalchemy.org/en/20/orm/declarative_tables.html#declarative-table-with-mapped-column).

This is used in many recent tutorials and is used the COMP0035 teaching activities.

```
from typing import Optional
from sqlalchemy import Integer, String, ForeignKey
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column


# Create a declarative base class
class Base(DeclarativeBase):
    pass


# An example class using mapping. The class inherits the Base class.
class User(Base):
    __tablename__ = "user"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(50))
    fullname: Mapped[Optional[str]]
```

Declarative (without type annotation)

```
from sqlalchemy import Integer, String
from sqlalchemy.orm import DeclarativeBase, mapped_column


class Base(DeclarativeBase):
    pass


class User(Base):
    __tablename__ = "user"

    id = mapped_column(Integer, primary_key=True)
    name = mapped_column(String(50), nullable=False)
    fullname = mapped_column(String)
```

You may also see the following style which defines the columns using `Column` which does not support all the configuration that `mapped_column` does.

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

# Define the base sqlalchemy class you will inherit in your models
        using declarative style
Base = declarative_base()


class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
```

Imperative

```
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy.orm import registry

mapper_registry = registry()

user_table = Table(
    "user",
    mapper_registry.metadata,
    Column("id", Integer, primary_key=True),
    Column("name", String(50)),
    Column("fullname", String(50)),
)


class User:
    pass


mapper_registry.map_imperatively(User, user_table)
```

Activity: Define a sqlalchemy model class

Use a declarative style and write a model class for **Person**:

Person			
int	id	PK	
string	first_name		
string	last_name		
date	date_of_birth		optional

You will need the following syntax:

- Declare the base class:

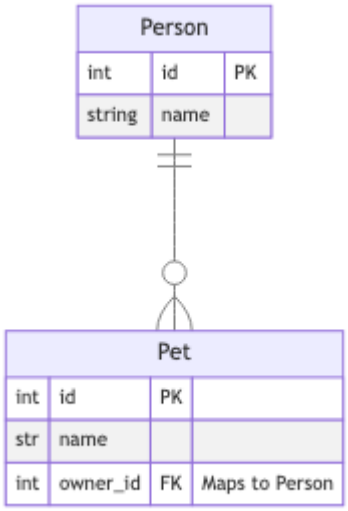
```
class Base(DeclarativeBase):
    pass
```

- `def ClassName(Base):` inherit the base when you define your class
- `__tablename__ = 'class_name'` define the table name. This is optional, if ou don't specify it will default to the class name in lowercase.
- `id = mapped_column(Integer, primary_key=True)` for a PK field

- `name: Mapped[str] = mapped_column(String(50), nullable=True)` for a non-key field. `nullable=True` denotes an optional field, i.e. one that can have a null value, `nullable=False` means a value is required and prevents null. `nullable=False` is the default for primary keys and that it's good practice to explicitly set it for required fields.

Relationships

Consider that a person may have one or more pets:



To define relationships you define the ForeignKey field e.g. `owner_id = mapped_column(ForeignKey("person.id"))`

You can also define a relationship using `sqlalchemy.orm.relationship` that lets the `Person` know which `Pets` they own, and each `Pet` know who its owner (`Person`) is.

This is an example from the SQLAlchemy documentation showing how to define this for a `User` with `Addresses` (email address).

```
from __future__ import annotations
from typing import List

from sqlalchemy import ForeignKey, Integer, String
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column,
        relationship


class Base(DeclarativeBase):
    pass


class User(Base):
    __tablename__ = "user"

    id = mapped_column(Integer, primary_key=True)
    name: Mapped[str]
    # Relationship
    addresses: Mapped[List["Address"]] =
        relationship(back_populates="user")


class Address(Base):
    __tablename__ = "address"

    id = mapped_column(Integer, primary_key=True)
    user_id = mapped_column(ForeignKey("user.id"))
    email_address: Mapped[str]
    # Relationship
    user: Mapped["User"] = relationship(back_populates="addresses")
```

Cascading delete

You can configure a relationship so that when a parent object is deleted, its related child objects are also deleted automatically. This is done using the `cascade` parameter:

```
pets: Mapped[List["Pet"]] = relationship(
    back_populates="owner",
    cascade="all, delete-orphan"
)
```

This means:

- `all`: apply all default cascade behaviors (save-update, merge, refresh-expire, expunge, delete).
- `delete-orphan`: delete child objects that are no longer associated with a parent.

This is optional but can be useful.

Activity: add Pets table and relationships

1. Define the pet table and add the id from the Person table as a foreign key attribute.
2. Add a relationship to Person that is a list of Pet objects associated with them.
3. Add a relationship to Pet that is a single Person object associated with them.

Create a database using the tables

This is similar to the SQLModel approach, create an engine (https://docs.sqlalchemy.org/en/20/core/engines.html) to access the sqlite database file (https://docs.sqlalchemy.org/en/20/core/engines.html#sqlite) and use the `create_all()` method (https://docs.sqlalchemy.org/en/20/core/metadata.html#sqlalchemy.schema.MetaData.create_all) to create all the model classes.

The model classes need to either be imported before the `create_all()` is called, or be in the same code file as the `create_all()` code.

```
from sqlalchemy import create_engine, MetaData

# import the models if not in the same file
# import the Base class you declared if not in the same file

# an Engine to connect to the database (this points to a path in the
        current directory)
engine = create_engine("sqlite:///pets.db")

# create the tables that are defined in the schema
Base.metadata.create_all(engine)
```

Activity: create the database

Add the code to create the database.

You can either add this to the same file as the model classes, or create a new file with the code that creates the database. If you choose the latter route, don't forget to import the models and the Base class before you call `create_all()`.

Add data

This is similar to the SQLModel pattern in activity 6.

Create the objects, then use the session to add (https://docs.sqlalchemy.org/en/20/orm/session_basics.html#adding-new-or-existing-items) them to the database.

For example:

```
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

# Create an object
hero = User(username="Deadpond")

# Create an engine with the url of the database
engine = create_engine("sqlite:///database.db")

# Create a session
with Session(engine) as session:
    # Add the object
    session.add(hero)
    session.commit()
```

Where you have a relationship then you can add the related object through the relationship.

For example:

```
user = User(username="joebloggs")
email_address = Address(email="jo@bloggs.com")
# Associate the user with this email address using the 'addresses'
        relationship attribute. Since they can have many addresses this
        is a list of addresses so you can append to the list.
user.addresses.append(email_address)

with Session(engine) as session:
    # Adding the user object also adds the associatred email_address
    session.add(user)
    session.commit()
```

Activity: Add a person and their pet to the database

Write code to:

1. Create a person object
2. Create a pet object
3. Append the pet to the 'pets' relationship of the person
4. Add the person in the session and commit