This lesson is being piloted (Beta version)

Intermediate Research Software Development (/az-intermediate-software-skills-course/index.html): Additional Material: Databases

Databases

A **database** is an organised collection of data, usually organised in some way to mimic the structure of the entities it represents. There are several major families of database model, but the dominant form is the **relational database**.

Relational databases focus on describing the relationships between entities in the data, similar to the object oriented paradigm. The key concepts in a relational database are:

Tables

Within a database we can have multiple tables - each table usually represents all entities of a single type.
e.g. We might have a `patients` table to represent all of our patients.

Columns / Fields

Each table has columns - each column has a name and holds data of a specific type
e.g. We might have a `name` column in our `patients` table which holds text data representing the names of our patients.

Rows

Each table has rows - each row represents a single entity and has a value for each field.
e.g. Each row in our `patients` table represents a single patient - the value of the `name` field in this row is our patient's name.

Primary Keys

Each row has a primary key - this is a unique ID that can be used to select this from from the data.
e.g. Each patient might have a `patient_id` which can be used to distinguish two patients with the same name.

Foreign Keys

A relationship between two entities is described using a foreign key - this is a field which points to the primary key of another row / table.
e.g. Each patient might have a foreign key field called `doctor` pointing to a row in a `doctors` table representing the doctor responsible for them - i.e. this doctor *has a* patient.

📌 SQLAlchemy

For more information, see SQLAlchemy's ORM tutorial (https://docs.sqlalchemy.org/en/13/orm/tutorial.html).

While relational databases are typically accessed using **SQL queries**, we're going to use a library to help us translate between Python and the database. SQLAlchemy is a popular Python library which contains an **Object Relational Mapping** (ORM) framework.

Our first step is to install SQLAlchemy, then we can create our first **mapping**.

Bash

```
$ pip3 install sqlalchemy
```

A mapping is the core component of an ORM - it's this that describes how to convert between our Python classes and the contents of our database tables. Typically, we can take our existing classes and convert them into mappings with a little modification, so we don't have to start from scratch.

Python

```
# file: inflammation/models.py
from sqlalchemy import Column, create_engine, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

...

class Patient(Base):
    __tablename__ = 'patients'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.observations = []
        if 'observations' in kwargs:
            self.observations = kwargs['observations']
```

Now that we've defined how to translate between our Python class and a database table, we need to hook our code up to an actual database.

The library we're using, SQLAlchemy, does everything through a database **engine**. This is essentially a wrapper around the real database, so we don't have to worry about which particular database software is being used - we just need to write code for a generic relational database.

For these lessions we're going to use the SQLite engine as this requires almost no configuration and no external software. Most relational database software runs as a separate service which we can connect to from our code. This means that in a large scale environment, we could have the database and our software running on different

computers - we could even have the database spread across several servers if we have particularly high demands for performance or reliability. Some examples of databases which are used like this are PostgreSQL, MySQL and MSSQL.

On the other hand, SQLite runs entirely within our software and uses only a single file to hold its data. It won't give us the extremely high performance or reliability of a properly configured PostgreSQL database, but it's good enough in many cases and much less work to get running.

Lets write some test code to setup and connect to an SQLite database. For now we'll store the database in memory rather than an actual file - it won't actually allow us to store data after the program finishes, but it allows us not to worry about **migrations**.

📌 Migrations

When we make changes to our mapping (e.g. adding / removing columns), we need to get the database to update its tables to make sure they match the new format. This is what the `Base.metadata.create_all` method does - creates all of these tables from scratch because we're using an in-memory database which we know will be removed between runs.

If we're actually storing data persistently, we need to make sure that when we change the mapping, we update the database tables without damaging any of the data they currently contain. We could do this manually, by running SQL queries against the tables to get them into the right format, but this is error-prone and can be a lot of work.

In practice, we generate a migration for each change. Tools such as Alembic (https://alembic.sqlalchemy.org/en/latest/) will compare our mappings to the known state of the database and generate a Python file which updates the database to the necessary state.

Migrations can be quite complex, so we won't be using them here - but you may find it useful to read about them later.

Python

```
# file: tests/test_models.py

...

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

...

def test_sqlalchemy_patient_search():
    """Test that we can save and retrieve patient data from a database."""
    from inflammation.models import Base, Patient

    # Setup a database connection - we're using a database stored in memory here
    engine = create_engine('sqlite:///:memory:', echo=True)
    Session = sessionmaker(bind=engine)
    session = Session()
    Base.metadata.create_all(engine)

    # Save a patient to the database
    test_patient = Patient(name='Alice')
    session.add(test_patient)

    # Search for a patient by name
    queried_patient = session.query(Patient).filter_by(name='Alice').first()
    self.assertEqual(queried_patient.name, 'Alice')
    self.assertEqual(queried_patient.id, 1)

    # Wipe our temporary database
    Base.metadata.drop_all(engine)
```

For this test, we've imported our models inside the test function, rather than at the top of the file like we normally would. This is not recommended in normal code, as it means we're paying the performance cost of importing every time we run the function, but can be useful in test code. Since each test function only runs once per test session, this performance cost isn't as important as a function we were going to call many times. Additionally, if we try to import something which doesn't exist, it will fail - by imporing inside the test function, we limit this to that specific test failing, rather than the whole file failing to run.

Relationships

Relational databases don't typically have an 'array of numbers' column type, so how are we going to represent our observations of our patients' inflammation? Well, our first step is to create a table of observations. We can then use a **foreign key** to point from the observation to a patient, so we know which patient the data belongs to. The table also needs a column for the actual measurement - we'll call this `value` - and a column for the day the measurement was taken on.

We can also use the ORM's `relationship` helper function to allow us to go between the observations and patients without having to do any of the complicated table joins manually.

Python

```
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

...

class Observation(Base):
    __tablename__ = 'observations'

    id = Column(Integer, primary_key=True)
    day = Column(Integer)
    value = Column(Integer)
    patient_id = Column(Integer, ForeignKey('patients.id'))

    patient = relationship('Patient', back_populates='observations')


class Patient(Base):
    __tablename__ = 'patients'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    observations = relationship('Observation',
                                order_by=Observation.day,
                                back_populates='patient')
```

📌 Time is Hard

We're using an integer field to store the day on which a measurement was taken. This keeps us consistent with what we had previously as it's essentialy the position of the measurement in the numpy array. It also avoids us having to worry about managing actual date / times.

The Python `datetime` module we've used previously in the Academics example would be useful here, and most databases have support for 'date' and 'time' columns, but to reduce the complexity, we'll just use integers here.

Our test code for this is going to look very similar to our previous test code, so we can copy-paste it and make a few changes. This time, after setting up the database, we need to add a patient and an observation. We then test that we can get the observations from a patient we've searched for.

Python

```
# file: tests/test_models.py

...

def test_sqlalchemy_observations():
    """Test that we can save and retrieve inflammation observations from a database."""
    from inflammation.models import Base, Observation, Patient

    # Setup a database connection - we're using a database stored in memory here
    engine = create_engine('sqlite:///:memory:', echo=True)
    Session = sessionmaker(bind=engine)
    session = Session()
    Base.metadata.create_all(engine)

    # Save a patient to the database
    test_patient = Patient(name='Alice')
    session.add(test_patient)

    test_observation = Observation(patient=test_patient, day=0, value=1)
    session.add(test_observation)

    queried_patient = session.query(Patient).filter_by(name='Alice').first()
    first_observation = queried_patient.observations[0]
    self.assertEqual(first_observation.patient, queried_patient)
    self.assertEqual(first_observation.day, 0)
    self.assertEqual(first_observation.value, 1)

    # Wipe our temporary database
    Base.metadata.drop_all(engine)
```

Finally, let's put in a way to convert all of our observations into a numpy array, so we can use our previous analysis code. We'll use the `property` decorator here again, to create a method that we can use as if it was a normal data attribute.

Python

```
# file: inflammation/models.py

...

class Patient(Base):
    __tablename__ = 'patients'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    observations = relationship('Observation',
                               order_by=Observation.day,
                               back_populates='patient')

    @property
    def values(self):
        """Convert inflammation data into numpy array."""
        last_day = self.observations[-1].day
        values = np.zeros(last_day + 1)

        for observation in self.observations:
            values[observation.day] = observation.value

        return values
```

Once again we'll copy-paste the test code and make some changes. This time we want to create a few observations for our patient and test that we can turn them into a numpy array.

Python

```
# file: tests/test_models.py

def test_sqlalchemy_observations_to_array():
    """Test that we can save and retrieve inflammation observations from a database."""
    from inflammation.models import Base, Observation, Patient

    # Setup a database connection - we're using a database stored in memory here
    engine = create_engine('sqlite:///:memory:')
    Session = sessionmaker(bind=engine)
    session = Session()
    Base.metadata.create_all(engine)

    # Save a patient to the database
    test_patient = Patient(name='Alice')
    session.add(test_patient)

    for i in range(5):
        test_observation = Observation(patient=test_patient, day=i, value=i)
        session.add(test_observation)

    queried_patient = session.query(Patient).filter_by(name='Alice').first()
    npt.assert_array_equal([0, 1, 2, 3, 4], queried_patient.values)

    # Wipe our temporary database
    Base.metadata.drop_all(engine)
```

Further Array Testing

There's an important feature of the behaviour of our `Patient.values` property that's not currently being tested. What is this feature? Write one or more extra tests to cover this feature.

Hint The `Patient.values` property creates an array of zeroes, then fills it with data from the table. If a measurement was not taken on a particular day, that day's value will be left as zero. If this is intended behaviour, it would be useful to write a test for it, to ensure that we don't break it in future. Using tests in this way is known as **regression testing**.

Refactoring for Reduced Redundancy

You've probably noticed that there's a lot of replicated code in our database tests. It's fine if some code is replicated a bit, but if you keep needing to copy the same code, that's a sign it should be refactored.

Refactoring is the process of changing the structure of our code, without changing its behaviour, and one of the main benefits of good test coverage is that it makes refactoring easier. If we've got a good set of tests, it's much more likely that we'll detect any changes to behaviour - even when these changes might be in the tests themselves.

Try refactoring the database tests to see if you can reduce the amount of replicated code by moving it into one or more functions at the top of the test file.

Advanced Challenge: Connecting More Views

We've added the ability to store patient records in the database, but not actually connected it to any useful views. There's a common pattern in data management software which is often refered to as **CRUD** - Create, Read, Update, Delete. These are the four fundamental views that we need to provide to allow people to manage their data effectively.

Each of these applies at the level of a single record, so for both patients and observations we should have a view to: create a new record, show an existing record, update an existing record and delete an existing record. It's also sometimes useful to provide a view which lists all existing records for each type - for example, a list of all patients would probably be useful, but a list of all observations might not be.

Pick one (or several) of these views to implement - you may want to refer back to the section where we added our initial patient read view.

Advanced Challenge: Managing Dates Properly

Try converting our existing models to use actual dates instead of just a day number. The Python datetime module documentation (https://docs.python.org/3/library/datetime.html) and SQLAlchemy Column and Data Types page (https://docs.sqlalchemy.org/en/13/core/type_basics.html) will be useful to you here.

Edit on GitHub (https://github.com/softwaresaved/az-intermediate-software-skills-course/edit/gh-pages/_extras/databases.md) / Contributing (https://github.com/softwaresaved/az-intermediate-software-skills-course/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/softwaresaved/az-intermediate-software-skills-course/) / Cite (https://github.com/softwaresaved/az-intermediate-software-skills-course/blob/gh-pages/CITATION) / Contact (mailto:info@software.ac.uk)

Using The Carpentries theme (https://github.com/carpentries/carpentries-theme/) — Site last built on: 2022-02-22 13:47:01 +0000.