Saving data

This lab covers

- [Storing data in relational databases]
- [Using the Python DB-API]
- [Accessing databases through an Object Relational Mapper (ORM)]
- [Understanding NoSQL databases and how they differ from relational databases]

When you have data and have it cleaned, it's likely that you'll want to store it. You'll not only want to store it, but also be able to get at it in the future with as little hassle as possible. The need to store and retrieve significant amounts of data usually calls for some sort of database. Relational databases such as PostgreSQL, MySQL, and SQL Server have been established favorites for data storage for decades, and they can still be great options for many use cases. In recent years, NoSQL databases, including MongoDB and Redis, have found favor and can be very useful for a variety of use cases. A detailed discussion of databases would take several books, so in this lab I look at some scenarios to show how you can access both SQL and NoSQL databases with Python.

Relational databases

Relational databases have long been a standard for storing and manipulating data. They're a mature technology and a ubiquitous one. Python can connect with a number relational databases, but I don't have the time or the inclination to go through the specifics of each one in this course. Instead, because Python handles databases in a mostly consistent way, I illustrate the basics with one of them---sqlite3---and then discuss some differences and considerations in choosing and using a relational database for data storages.

The Python Database API

As I mention, Python handles SQL database access very similarly across several database implementations because of PEP-249 (www.python.org/dev/peps/pep-0249/), which specifies some common practices for connecting to SQL databases. Commonly called the Database API or DB-API, it was created to encourage "code that is generally more portable across databases, and a broader reach of database connectivity." Thanks to the DB-API, the examples of SQLite that you see in this lab are quite similar to what you'd use for PostgreSQL, MySQL, or several other databases.

SQLite: Using the sqlite3 database

Although Python has modules for many databases, in the following examples I look at sqlite3. Although it's not suited for large, high-traffic applications, sqlite3 has two advantages:

- [Because it's part of the standard library, it can be used anywhere you need a database without worrying about adding dependencies.]
- [sqlite3 stores all of its records in a local file, so it doesn't need both a client and server, which would be the case for PostgreSQL, MySQL, and other larger databases.]

These features make sqlite3 a handy option for both smaller applications and quick prototypes.

To use a sqlite3 database, the first thing you need is a [Connection] object. Getting a [Connection] object requires only calling the [connect] function with the name of file that will be used to store the data:

```
>>> import sqlite3
>>> conn = sqlite3.connect("datafile.db")
```

It's also possible to hold the data in memory by using [":memory:"] as the filename. For storing Python integers, strings, and floats, nothing more is needed. If you want sqlite3 to automatically convert query results for some columns into other types, it's useful to include the [detect_types] parameter set to [sqlite3.PARSE_DECLTYPES |sqlite3.PARSE_COLNAMES], which directs the [Connection] object to parse the name and types of columns in queries and attempts to match them with converters you've already defined.

The second step is creating a [Cursor object] from the connection:

```
>>> cursor = conn.cursor()
>>> cursor
<sqlite3.Cursor object at 0xb7a12980>
```

At this point, you're able to make queries against the database. In the current situation, because the database has no tables or records yet, you first need to create a table and insert a couple of records:

```
>>> cursor.execute("create table people (id integer primary key, name text,
     count integer)")
>>> cursor.execute("insert into people (name, count) values ('Bob', 1)")
>>> cursor.execute("insert into people (name, count) values (?, ?)",
...                ("Jill", 15))
>>> conn.commit()
```

The last [insert] query illustrates the preferred way to make a query with variables. Rather than constructing the query string, it's more secure to use a [?] for each variable and then pass the variables as a tuple parameter to the [execute] method. The advantage is that you don't need to worry about incorrectly escaping a value; sqlite3 takes care of it for you.

You can also use variable names prefixed with [:] in the query and pass in a corresponding dictionary with the values to be inserted:

```
>>> cursor.execute("insert into people (name, count) values (:username, \
                   :usercount)", {"username": "Joe", "usercount": 10})
```

After a table is populated, you can query the data by using SQL commands, again using either [?] for variable binding or names and dictionaries:

```
>>> result = cursor.execute("select * from people")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 15), ('Joe', 10)]
>>> result = cursor.execute("select * from people where name like :name",
...                         {"name": "bob"})
>>> print(result.fetchall())
[('Bob', 1)]
>>> cursor.execute("update people set count=? where name=?", (20, "Jill"))
>>> result = cursor.execute("select * from people")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 20), ('Joe', 10)]
```

In addition to the [fetchall] method, the [fetchone] method gets one row of the result, and [fetchmany] returns an arbitrary number of rows. For convenience, it's also possible to iterate over a cursor object's rows similarly to iterating over a file:

```
>>> result = cursor.execute("select * from people")
>>> for row in result:
...     print(row)
...
('Bob', 1)
('Jill', 20)
('Joe', 10)
```

Finally, by default, sqlite3 doesn't immediately commit transactions. This fact means that you have the option of rolling back a transaction if it fails, but it also means that you need to use the [Connection] object's [commit] method to ensure that any changes made have been saved. Doing so before you close a connection to a database is a particularly good idea because the [close] method doesn't automatically commit any active transactions:

```
>>> cursor.execute("update people set count=? where name=?", (20, "Jill"))
>>> conn.commit()
>>> conn.close()
```

Using MySQL, PostgreSQL, and other relational databases

As I mentioned earlier in this lab, several other SQL databases have client libraries that follow the DB-API. As a result, accessing those databases in Python is quite similar, but there are a couple of differences to look out for:

- [Unlike SQLite, those databases require a database server that the client connects to and that may or may not be on a different machine, so the connection requires more parameters---usually including host, account name, and password.]
- [The way in which parameters are interpolated into queries, such as ["select * from test where name like :name"], could use a different format---something like [?, %s 5(name)s].]

These changes aren't huge, but they tend to keep code from being completely portable across different databases.

Making database handling easier with an ORM

There are a few problems with the DB-API database client libraries mentioned earlier in this lab and their requirement to write raw SQL:

- [Different SQL databases have implemented SQL in subtly different ways, so the same SQL statements won't always work if you move from one database to another, as you might want to do if, say, you do local development against sqlite3 and then want to use MySQL or PostgreSQL in production. Also, as mentioned earlier, the different implementations have different ways of doing things like passing parameters into queries.]
- [The second drawback is the need to use raw SQL statements. Including SQL statements in your code can make your code more difficult to maintain, particularly if you have a lot of them. In that case, some of the statements will be boilerplate and routine; others will be complex and tricky; and all of them need to be tested, which can get cumbersome.]
- [The need to write SQL means that you need to think in at least two languages: Python and a specific SQL variant. In plenty of cases, it's worth these hassles to use raw SQL, but in many other cases, it isn't.]

Given those issues, people wanted a way to handle databases in Python that was easier to manage and didn't require anything more than writing regular Python code. The solution is an Object Relational Mapper (ORM), which converts, or maps, relational database types and structures to objects in Python. Two of the most common ORMs in the Python world are the Django ORM and SQLAlchemy, although of course there are many others. The Django ORM is rather tightly integrated with the Django web framework and usually isn't used outside it. Because I'm not delving into Django in this course, I won't discuss the Django ORM other than to note that it's the default choice for Django applications and a good one, with fully developed tools and generous community support.

SQLAlchemy

SQLAlchemy is the other big-name ORM in the Python space. SQLAlchemy's goal is to automate redundant database tasks and provide Python object-based interfaces to the data while still allowing the developer control of the database and access to the underlying SQL. In this section, I look at some basic examples of storing data into a relational database and then retrieving it with SQLAlchemy.

You can install SQLAlchemy in your environment with [pip]:

```
> pip install sqlalchemy
```

Note

In working with SQLAlchemy and its related tools from this point, it will be more convenient to have two shell windows open in the same virtual environment: one for Python and one for your system's command line.

SQLAlchemy offers several ways to interact with database and its tables. Although an ORM lets you write SQL statements if you want or need to, the strength of an ORM is doing what the name suggests: mapping the relational database tables and columns to Python objects.

Use SQLAlchemy to replicate what you did in [section 23.2]: Create a table, add three rows, query the table, and update one row. You need to do a bit more setup to use the ORM, but in larger projects, this effort is more than worth it.

First, you need to import the components you need to connect to the database and map a table to Python objects. From the base [sqlalchemy] package, you need the [create_engine] and [select] methods and the [MetaData] and [Table] classes. But because you need to specify the schema information when you create the [table] object, you also need to import the [Column] class and the classes for the data type of each column---in this case, [Integer] and [String]. From the [sqlalchemy.orm] subpackage, you also need the [sessionmaker] function:

```
>>> from sqlalchemy import create_engine, select, MetaData, Table, Column,
     Integer, String
>>> from sqlalchemy.orm import sessionmaker
```

Now you can think about connecting to the database:

```
>>> dbPath = 'datafile2.db'
>>> engine = create_engine('sqlite:///%s' % dbPath)
>>> metadata = MetaData(engine)
>>> people  = Table('people', metadata,
...                 Column('id', Integer, primary_key=True),
...                 Column('name', String),
...                 Column('count', Integer),
...                )
>>> Session = sessionmaker(bind=engine)
>>> session = Session()
>>> metadata.create_all(engine)
```

To create and connect, you need to create a database engine appropriate for your database; then you need a [MetaData] object, which is a container for managing tables and their schemas. Create a [Table] object called [data], giving the table's name in the database, the [MetaData] object you just created, and the column you want to create, as well as their data types. Finally, you use the [sessionmaker] function to create a [Session] class for your engine and use that class to instantiate a session object. At this point, you're connected to the database, and the last step is to use the [create_all] method to create the table.

When the table is created, the next step is inserting some records. Again, you have many options for doing this in SQLAlchemy, but you'll be fairly explicit in this example. Create an [insert] object, which you then execute:

```
>>> people_ins = people.insert().values(name='Bob', count=1)
>>> str(people_ins)
'INSERT INTO people (name, count) VALUES (?, ?)'
>>> session.execute(people_ins)
<sqlalchemy.engine.result.ResultProxy object at 0x7f126c6dd438>
>>> session.commit()
```

Here, you use the [insert()] method to create an [insert] object, also specifying the fields and values you want to insert. [people_ins] is the [insert] object, and you use the [str()] function to show that behind the scenes, you created the correct SQL command. Then you use the session object's [execute] method to perform the insertion and the [commit] method to commit it to the database:

```
>>> session.execute(people_ins, [
...     {'name': 'Jill', 'count':15},
...     {'name': 'Joe', 'count':10}
... ])
<sqlalchemy.engine.result.ResultProxy object at 0x7f126c6dd908>
>>> session.commit()
>>> result = session.execute(select([people]))
>>> for row in result:
...     print(row)
...
(1, 'Bob', 1)
(2, 'Jill', 15)
(3, 'Joe', 10)
```

You can streamline things a bit and perform multiple inserts by passing in a list of dictionaries of the field names and values for each insert:

```
>>> result = session.execute(select([people]).where(people.c.name == 'Jill'))
>>> for row in result:
...     print(row)
...
(2, 'Jill', 15)
```

You can also use the [select()] method with a [where()] method to find a particular record. In the example, you're looking for any records in which the [name] column equals ['Jill']. Note that the [where] expression uses [people.c.name], with the [c] indicating that [name] is a column in the people table:

```
>>> result = session.execute(people.update().values(count=20).where
     (people.c.name == 'Jill'))
>>> session.commit()
>>> result = session.execute(select([people]).where(people.c.name == 'Jill'))
>>> for row in result:
...     print(row)
...
(2, 'Jill', 20)
>>>
```

Finally, you can combine an [update()] method with the [where()] method to update just one row.

Mapping table objects to classes

So far, you've used table objects directly, but it's also possible to use SQLAlchemy to map a table directly to a class. This technique has the advantage that the columns are mapped directly to class attributes. For illustration, make a class [People]:

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> Base = declarative_base()
>>> class People(Base):
...     __tablename__ = "people"
```

```
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     count = Column(Integer)
...
>>> results = session.query(People).filter_by(name='Jill')
>>> for person in results:
...     print(person.id, person.name, person.count)
...
2 Jill 20
```

Inserts can be done just by creating an instance of the mapped class and adding it to the session:

```
>>> new_person = People(name='Jane', count=5)
>>> session.add(new_person)
>>> session.commit()
>>>
>>> results = session.query(People).all()
>>> for person in results:
...     print(person.id, person.name, person.count)
...
1 Bob 1
2 Jill 20
3 Joe 10
4 Jane 5
```

Updates are also fairly straightforward. You retrieve the record you want to update, change the values on the mapped instance, and then add the updated record to the session to be written back to the database:

```
>>> jill = session.query(People).filter_by(name='Jill').first()
>>> jill.name
'Jill'
>>> jill.count = 22
>>> session.add(jill)
>>> session.commit()
>>> results = session.query(People).all()
>>> for person in results:
...     print(person.id, person.name, person.count)
...
1 Bob 1
2 Jill 22
3 Joe 10
4 Jane 5
```

Deleting is similar to updating; you fetch the record to be deleted and then use the session's [delete()] method to delete it:

```
>>> jane = session.query(People).filter_by(name='Jane').first()
>>> session.delete(jane)
>>> session.commit()
>>> jane = session.query(People).filter_by(name='Jane').first()
>>> print(jane)
None
```

Using SQLAlchemy does take a bit more setup than just using raw SQL, but it also has some real benefits. For one thing, using the ORM means that you don't need to worry about any subtle differences in the SQL supported by different databases. The example works equally well with sqlite3, MySQL, and PostgreSQL without making any changes in the code other than giving the string to the create engine and making sure that the correct database driver is available.

Another advantage is that the interaction with the data can happen through Python objects, which may be more accessible to coders who lack SQL experience. Instead of constructing SQL statements, they can use Python objects and their methods.

Try this: Using an ORM

Using the database from earlier, write an SQLAlchemy class to map to the data table, and use it to read the records from the table.

Using Alembic for database schema changes

In the course of developing code that uses a relational database it's quite common, if not universal, to have to change the structure or schema of the database after you've started work. Fields need to be added, or their types need to be changed, and so on. It's possible, of course, to manually make the changes to both the database tables and to the code for the ORM that accesses them, but that approach has some drawbacks. For one thing, such changes are difficult to roll back if you need to, and it's hard to keep track of the configuration of the database that goes with a particular version of your code.

The solution is to use a database migration tool to help you make the changes and track them. Migrations are written as code and should include code both to apply the needed changes and to reverse them. Then the changes can be tracked and applied or reversed in the correct sequence. As a result, you can reliably upgrade or downgrade your database to any of the states it was in over the course of development.

As an example, this section looks briefly at Alembic, a popular lightweight migration tool for SQLAlchemy. To start, switch to the system command-line window in your project directory, install Alembic, and create a generic environment by using [alemic init]:

```
> pip install alembic
> alembic init alembic
```

This code creates the file structure you need to use Alembic for data migrations. There's an alembic.ini file that you need to edit in at least one place. The [squalchemy.url] line needs to be updated to match your current situation:

```
sqlalchemy.url = driver://user:pass@localhost/dbname
```

Change the line to

```
sqlalchemy.url = sqlite:///datafile.db
```

Because you're using a local sqlite file, you don't need a username or password.

The next step is creating a revision by using Alembic's revision command:

```
> alembic revision -m "create an address table"
Generating /home/naomi/qpb_testing/alembic/versions/
     384ead9efdfd_create_a_test_address_table.py ... done
```

This code creates a revision script, 384ead9efdfd_create_a_test_address_table.py, in the alembic/versions directory. This file looks like this:

```
"""create an address table

Revision ID: 384ead9efdfd
Revises:
Create Date: 2017-07-26 21:03:29.042762

"""
from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision = '384ead9efdfd'
down_revision = None
branch_labels = None
depends_on = None


def upgrade():
    pass


def downgrade():
    pass
```

You can see that the file contains the revision ID and date in the header. It also contains a [down_revision] variable to guide the rollback of each version. If you make a second revision, its [down_revision] variable should contain this revision's ID.

To perform the revision, update the revision script to supply both the code describing how to perform the revision in the [upgrade()] method and the code to reverse it in the [downgrade()] method:

```
def upgrade():
    op.create_table(
        'address',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('address', sa.String(50), nullable=False),
        sa.Column('city', sa.String(50), nullable=False),
        sa.Column('state', sa.String(20), nullable=False),
    )

def downgrade():
    op.drop_table('address')
```

When this code is created, you can apply the upgrade. But first, switch back to the Python shell window to see what tables you have in your database:

```
>>> print(engine.table_names())
['people']
```

As you might expect, you have only the one table you created earlier. Now you can run Alembic's [upgrade] command to apply the upgrade and add a new table. Switch over to your system command line, and run

```
> alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 384ead9efdfd, create an
     address table
```

If you pop back to Python and check, you see that the database has two additional tables:

```
>>> engine.table_names()
['alembic_version', 'people', 'address'
```

The first new table, ['alembic version'], is created by Alembic to help track which version your database is currently on (for reference for future upgrades and downgrades). The second new table, ['address'], is the table you added through your upgrade and is ready to use.

If you want to roll back the state of the database to what it was before, all you need to do is run Alembic's [downgrade] command in the system window. You give the downgrade command [-1] to tell Alembic that you want to downgrade by one version:

```
> alembic downgrade -1
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running downgrade 384ead9efdfd -> , create
     an address table
```

Now if you check in your Python session, you'll be back to where you started except that the version tracking table remains:

```
>>> engine.table_names()
['alembic_version', 'people']
```

If you want to, of course, you can run the upgrade again to put the table back, add further revisions, make upgrades, and so on.

Try this: Modifying a database with Alembic
Experiment with creating an Alembic upgrade that adds a state table to your database, with columns for ID, state name, and abbreviation. Upgrade and downgrade. What other changes would be needed if you were going to use the state table along with the existing data table?

NoSQL databases

In spite of their longstanding popularity, relational databases aren't the only ways to think about storing data. Although relational databases are all about normalizing data within related tables, other approaches look at data differently. Quite commonly, these types of databases are referred to as *NoSQL* databases, because they usually don't adhere to the row/column/table structure that SQL was created to describe.

Rather than handle data as collections of rows, columns, and tables, NoSQL databases can look at the data they store as key-value pairs, as indexed documents, and even as graphs. Many NoSQL databases are available, all with somewhat different ways of handling data. In general, they're less likely to be strictly normalized, which can make retrieving information faster and easier. As examples, in this lab I look at using Python to access two common NoSQL databases: Redis and MongoDB. What follows barely scratches the surface of what you can do with NoSQL databases and Python, but it should give you a basic idea of the possibilities. If you're already familiar with Redis or MongoDB, you'll see a little of how the Python client libraries work, and if you're new to NoSQL databases, you'll at least get an idea of how databases like these work.

key:value stores with Redis

Redis is an in-memory networked key:value store. Because the values are in memory, lookups can be quite fast, and the fact that it's designed to be accessed over the network makes it useful in a variety of situations. Redis is commonly used for caching, as a message broker, and for quick lookups of information. In fact, the name (which comes from Remote Dictionary Server) is an excellent way to think of it; it behaves much like a Python dictionary translated to a network service.

The following example gives you an idea of how Redis works with Python. If you're familiar with the Redis command-line interface or have used a Redis client for another language, these short examples should get you well on your way to using Redis with Python. If Redis is new to you, the following gives you an idea of how it works; you can explore more at https://redis.io.

Although several Python clients are available for Redis, at this writing the way to go (according to the Redis website) is one called redis-py. You can install it with [pip install redis].

Running a Redis server

To experiment, you need to have a Redis server running. Although you could use cloud-based Redis services, for experimentation your best choices are using a Docker instance or installing a server on a machine.

If you have Docker installed, using the Redis Docker instance is probably the quickest and easiest way to get a server up and running. You should be able to launch a Redis instance from the command line with a command like [> docker run -p 6379:6379 redis].

On Linux systems, it should be fairly easy to install Redis by using the system package manager, and on Mac systems, [brew install redis] should work. On Windows systems, you should check the https://redis.io website or search online for the current options for running Redis on Windows. When Redis is installed, you may need to look online for instructions to make sure that the Redis server is running.

When you get a server running, the following are examples of simple Redis interactions with Python. First, you need to import the Redis library and create a Redis connection object:

```
>>> import redis
>>> r = redis.Redis(host='localhost', port=6379)
```

You can use several connection options when creating a Redis connection, including the host, port, and password or SSH certificate. If the server is running on localhost on the default port of 6379, no options are needed. When you have the connection, you can use it to access the key:value store.

One of the first things you might do is use the [keys()] method to get a list of the keys in the database, which returns a list of keys currently stored (if any). Then you can set some keys of different types and try some ways to retrieve their values:

```
>>> r.keys()
[]
>>> r.set('a_key', 'my value')
True
>>> r.keys()
[b'a_key']
>>> v = r.get('a_key')
>>> v
b'my value'
>>> r.incr('counter')
1
>>> r.get('counter')
```

```
b'1'
>>> r.incr('counter')
2
>>> r.get('counter')
b'2'
```

These examples show how you can get a list of the keys in the Redis database, how to set a key with a value, and how to set a key with a [counter] variable and increment it.

These examples deal with storing arrays or lists:

```
>>> r.rpush("words", "one")
1
>>> r.rpush("words", "two")
2
>>> r.lrange("words", 0, -1)
[b'one', b'two']
>>> r.rpush("words", "three")
3
>>> r.lrange("words", 0, -1)
[b'one', b'two', b'three']
>>> r.llen("words")
3
>>> r.lpush("words", "zero")
4
>>> r.lrange("words", 0, -1)
[b'zero', b'one', b'two', b'three']
>>> r.lrange("words", 2, 2)
[b'two']
>>> r.lindex("words", 1)
b'one'
>>> r.lindex("words", 2)
b'two'
```

When you start the key, ["words"] isn't in the database, but the act of adding or pushing a value to the end (from the right, the [r] in [rpush]) creates the key, makes an empty list as its value, and then appends the value ['one']. Using [rpush] again adds another word to the end. To retrieve the values in the list, you can use the [lrange()] function, giving the key and both a starting index and an ending index, with [-1] indicating the end of the list.

Also note that you can add to the beginning, or left side, of the list with [lpush()]. You can use [lindex()] to retrieve a single value in the same way as [lranger()], except that you give it the index of the value you want.

Expiration of values

One feature of Redis that makes it particularly useful for caching is the ability to set an expiration for a key-value pair. After that time has elapsed, the key and value are removed. This technique is particularly useful for using Redis as a cache. You can set the timeout value in seconds when you set the value for a key:

```
>>> r.setex("timed", "10 seconds", 10)
True
>>> r.pttl("timed")
7165
>>> r.pttl("timed")
5208
```

```
>>> r.pttl("timed")
1542
>>> r.pttl("timed")
>>>
```

In this case, you set the expiration of ["timed"] to 10 seconds. Then, as you use the [pttl()] method, you can see the time remaining before expiration in milliseconds. When the value expires, both the key and value are automatically removed from the database. This feature and the fine-grained control of it that Redis offers are really useful. For simple caches, you may not need to write much more code to have your problem solved.

It's worth noting that Redis holds its data in memory, so keep in mind that the data isn't persistent; if the server crashes, some data is likely to be lost. To mitigate the possibility of data loss, Redis has options to manage persistence---everything from writing every change to disk as it occurs to making periodic snapshots at predetermined times to not saving to disk at all. You can also use the Python client's [save()] and [bgsave()] methods to programmatically force a snapshot to be saved, either blocking until the save is complete with [save()] or saving in the background in the case of [bgsave()].

In this lab, I've only touched on a small part of what Redis can do, as well as its data types and the ways it can manipulate them. If you're interested in finding out more, several sources of documentation are available online, including at https://redislabs.com and https://redis-py.readthedocs.io.

Quick Check: Uses of Key:Value stores

What sorts of data and applications would benefit most from a key:value store like Redis?

Documents in MongoDB

Another popular NoSQL database is MongoDB, which is sometimes called a document-based database because it isn't organized in rows and columns but instead stores documents. MongoDB is designed to scale across many nodes in multiple clusters while potentially handling billions of documents. In the case of MongoDB, a document is stored in a format called BSON (Binary JSON), so a document consists of key-value pairs and looks like a JSON object or Python dictionary. The following examples give you a taste of how you can use Python to interact with MongoDB collections and documents, but a word of warning is appropriate. In situations requiring scale and distribution of data, high insert rates, complex and unstable schemas, and so on, MongoDB is an excellent choice. However, MongoDB isn't the best choice in many situations, so be sure to investigate your needs and options thoroughly before choosing.

Running a MongoDB server

As with Redis, if you want to experiment with MongoDB, you need to have access to a MongoDB server. Numerous cloud-hosted Mongo services are available, but again, if you're just experimenting, you'll probably be better off running a Docker instance or installing on a server you own.

As is the case with Redis, the easiest solution is to run a Docker instance. All you need to do if you have Docker is enter [> docker run -p 27017:27017 mongo] at the command line. On a Linux system, your package manager should do the job, and the Mac's [brew install mongodb] will do it. On Windows systems, check on www.mongodb.com for the Windows version and installation instructions. As with Redis, search online for any instructions on how to configure and start the server.

As is the case with Redis, several Python client libraries connect to MongoDB databases. To give you an idea of how they work, look at pymongo. The first step in using pymongo is installing it, which you can do with pip:

```
> pip install pymongo
```

When you have pymongo installed, you can connect to a MongoDB server by creating an instance of MongoClient and specifying the usual connection details:

```
>>> from pymongo import MongoClient
>>> mongo = MongoClient(host='localhost', port=27017)
```

MongoDB is organized in terms of a database which contains collections, each of which can contain documents. Databases and collections don't need to be created before you try to access them, however. If they don't exist, they're created as you insert into them, or they simply return no results if you try to retrieve records from them.

To test the client, make a sample document, which can be a Python dictionary:

```
>>> import datetime
>>> a_document = {'name': 'Jane',
...               'age': 34,
...               'interests': ['Python', 'databases', 'statistics'],
...               'date_added': datetime.datetime.now()
... }
>>> db = mongo.my_data
>>> collection = db.docs
>>> collection.find_one()
>>> db.collection_names()
[]
```

Here, you connect to a database and a collection of documents. In this case, they don't exist, but they'll be created as you access them. Note that no exceptions were raised even though the database and collection didn't exist. When you asked for a list of the collections, however, you got an empty list because nothing has been stored in your collection. To store a document, use the collection's [insert()] method, which returns the document's unique [ObjectId] if the operation is successful:

```
>>> collection.insert(a_document)
ObjectId('59701cc4f5ef0516e1da0dec')
>>> db.collection_names()
['docs']
```

Now that you've stored a document in the [docs] collection, it shows up when you ask for the collection names in your database. When the document is stored in the collection, you can query for it, update it, replace it, and delete it:

```
>>> collection.find_one()
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Jane', 'age': 34,
     'interests': ['Python', 'databases', 'statistics'], 'date_added':
     datetime.datetime(2017, 7, 19, 21, 59, 32, 752000)}
>>> from bson.objectid import ObjectId
>>> collection.find_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Jane',
     'age': 34, 'interests': ['Python', 'databases',
     'statistics'], 'date_added': datetime.datetime(2017,
     7, 19, 21, 59, 32, 752000)}
>>> collection.update_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')},
     {"$set": {"name":"Ann"}})
<pymongo.results.UpdateResult object at 0x7f4ebd601d38>
>>> collection.find_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Ann', 'age': 34,
     'interests': ['Python', 'databases', 'statistics'], 'date_added':
      datetime.datetime(2017, 7, 19, 21, 59, 32, 752000)}
>>> collection.replace_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')},
```

```
      {"name":"Ann"})
<pymongo.results.UpdateResult object at 0x7f4ebd601750>
>>> collection.find_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
{'_id': ObjectId('59701cc4f5ef0516e1da0dec'), 'name': 'Ann'}
>>> collection.delete_one({"_id":ObjectId('59701cc4f5ef0516e1da0dec')})
<pymongo.results.DeleteResult object at 0x7f4ebd601d80>
>>> collection.find_one()
```

First, notice that MongoDB matches according to dictionaries of the fields and their values to match. Dictionaries are also used to indicate operators, such as [$lt] (less than) and [$gt] (greater than), as well as commands such as [$set] for the update. The other thing to notice is that even though the record has been deleted and the collection is now empty, it still exists unless it's specifically dropped:

```
>>> db.collection_names()
['docs']
>>> collection.drop()
>>> db.collection_names()
[]
```

MongoDB can do many other things, of course. In addition to operating on one record, versions of the same commands cover multiple records, such as [find_many] and [update_many]. MongoDB also supports indexing to improve performances and has several methods to group, count, and aggregate data, as well as a built in map-reduce method.

Quick Check: Uses of MONGODB

Thinking back over the various data samples you've seen so far and other types of data in your experience, which do you think would be well suited to being stored in a database like MongoDB? Would others clearly not be suited, and if so, why not?

Lab 23: Create a database

Choose one of the datasets I've discussed in the past few chapters, and decide which type of database would be best for storing that data. Create that database, and write the code to load the data into it. Then choose the two most common and/or likely types of search criteria, and write the code to retrieve both single and multiple matching records.

Summary

- [Python has a Database API (DB-API) that provides a generally consistent interface for clients of several relational databases.]
- [Using an Object Relational Mapper (ORM) can make database code even more standard across databases.]
- [Using an ORM also lets you access relational databases through Python code and objects rather than SQL queries.]
- [Tools such as Alembic work with ORMs to use code to make reversible changes to a relational database schema.]
- [Key:value stores such as Redis provide quick in-memory data access.]
- [MongoDB provides scalability without the strict structure of relational databases.]