INTRODUCTION TO DATABASES IN PYTHON

Creating Databases and Tables

Creating Databases

- Varies by the database type
- Databases like PostgreSQL and MySQL have command line tools to initialize the database
- With SQLite, the `create_engine()` statement will create the database and file is they do not already exist

Building a Table

```
In [1]: from sqlalchemy import (Table, Column, String,
   ...:     Integer, Decimal, Boolean)

In [2]: employees = Table('employees', metadata,
   ...:     Column('id', Integer()),
   ...:     Column('name', String(255)),
   ...:     Column('salary', Decimal()),
   ...:     Column('active', Boolean()))

In [3]: metadata.create_all(engine)

In [4]: engine.table_names()
Out[4]: [u'employees']
```

Creating Tables

- Still uses the Table object like we did for reflection
- Replaces the autoload keyword arguments with Column objects
- Creates the tables in the actual database by using the `create_all()` method on the MetaData instance
- You need to use other tools to handle database table updates, such as Alembic or raw SQL

Creating Tables - Additional Column Options

- `unique` forces all values for the data in a column to be unique
- `nullable` determines if a column can be empty in a row
- `default` sets a default value if one isn't supplied.

Building a Table with Additional Options

```
In [1]: employees = Table('employees', metadata,
   ...:     Column('id', Integer()),
   ...:     Column('name', String(255), unique=True,
   ...:            nullable=False),
   ...:     Column('salary', Float(), default=100.00),
   ...:     Column('active', Boolean(), default=True))

In [2]: employees.constraints
Out[2]: {CheckConstraint(...
Column('name', String(length=255), table=<employees>,
       nullable=False),
Column('salary', Float(), table=<employees>,
       default=ColumnDefault(100.0)),
Column('active', Boolean(), table=<employees>,
       default=ColumnDefault(True)) ...
UniqueConstraint(Column('name', String(length=255),
                 table=<employees>, nullable=False))}
```

Let’s practice!

Inserting Data into a Table

Adding Data to a Table

- Done with the `insert()` statement
- `Insert()` takes the table we are loading data into as the argument
- We add all the values we want to insert in with the `values` clause as `column=value` pairs
- Doesn’t return any rows, so no need for a fetch method

Inserting One Row

```
In [1]: from sqlalchemy import insert

In [2]: stmt = insert(employees).values(id=1,
            name='Jason', salary=1.00, active=True)

In [3]: result_proxy = connection.execute(stmt)

In [4]: print(result_proxy.rowcount)
Out[4]: 1
```

Inserting Multiple Rows

- Build an insert statement without any values
- Build a list of dictionaries that represent all the values clauses for the rows you want to insert
- Pass both the stmt and the values list to the execute method on connection

Inserting Multiple Rows

```
In [1]: stmt = insert(employees)

In [2]: values_list = [
            {'id': 2, 'name': 'Rebecca', 'salary': 2.00,
             'active': True},
            {'id': 3, 'name': 'Bob', 'salary': 0.00,
             'active': False}
        ]

In [3]: result_proxy = connection.execute(stmt,
            values_list)

In [4]: print(result_proxy.rowcount)
Out[4]: 2
```

Let’s practice!

Updating Data in a Table

Updating Data in a Table

- Done with the `update` statement
- Similar to the insert statement but includes a `where` clause to determine what record will be updated
- We add all the values we want to update with the `values` clause as `column=value` pairs

Updating One Row

```
In [1]: from sqlalchemy import update

In [2]: stmt = update(employees)

In [3]: stmt = stmt.where(employees.columns.id == 3)

In [4]: stmt = stmt.values(active=True)

In [5]: result_proxy = connection.execute(stmt)

In [6]: print(result_proxy.rowcount)
Out[6]: 1
```

Updating Multiple Rows

- Build a where clause that will select all the records you want to update

Inserting Multiple Rows

```
In [1]: stmt = update(employees)

In [2]: stmt = stmt.where(
            employees.columns.active == True
        )

In [3]: stmt = stmt.values(active=False, salary=0.00)

In [4]: result_proxy = connection.execute(stmt)

In [5]: print(result_proxy.rowcount)
Out[5]: 3
```

Correlated Updates

```
In [1]: new_salary = select([employees.columns.salary])

In [2]: new_salary = new_salary.order_by(desc(
   ...:      employees.columns.salary)
              )

In [3]: new_salary = new_salary.limit(1)

In [4]: stmt = update(employees)

In [5]: stmt = stmt.values(salary=new_salary)

In [6]: result_proxy = connection.execute(stmt)

In [7]: print(result_proxy.rowcount)
Out[7]: 3
```

Correlated Updates

- Uses a `select()` statement to find the value for the column we are updating
- Commonly used to update records to a maximum value or change a string to match an abbreviation from another table

Let’s practice!

Introduction to Databases in Python

Deleting Data from a Database

Deleting Data from a Table

- Done with the `delete()` statement
- `delete()` takes the table we are loading data into as the argument
- A `where()` clause is used to choose which rows to delete
- Hard to undo so BE CAREFUL!!!

Deleting all Data from a Table

```
In [1]: from sqlalchemy import delete

In [2]: stmt = select([
            func.count(extra_employees.columns.id)])

In [3]: connection.execute(stmt).scalar()
Out[3]: 3

In [4]: delete_stmt = delete(extra_employees)

In [5]: result_proxy = connection.execute(delete_stmt)

In [6]: result_proxy.rowcount
Out[6]: 3
```

Deleting Specific Rows

- Build a where clause that will select all the records you want to delete

Deleting Specific Rows

```
In [1]: stmt = delete(employees).where(
             employees.columns.id == 3)

In [2]: result_proxy = connection.execute(stmt)

In [3]: result_proxy.rowcount
Out[3]: 1
```

Dropping a Table Completely

- Uses the `drop` method on the table
- Accepts the engine as an argument so it knows where to remove the table from
- Won’t remove it from metadata until the python process is restarted

Dropping a table

```
In [1]: extra_employees.drop(engine)

In [2]: print(extra_employees.exists(engine))
Out[2]: False
```

Dropping all the Tables

- Uses the `drop_all()` method on MetaData

Dropping all the Tables

```
In [1]: metadata.drop_all(engine)

In [2]: engine.table_names()
Out[2]: []
```

Let’s practice!