INTRODUCTION TO DATABASES IN PYTHON

Introduction to Databases

A database consists of tables

Census

state	sex	age	pop2000	pop2008
New York	F	0	120355	122194
New York	F	1	118219	119661
New York	F	2	119577	116413

State_Fact

name	abbreviation	type
New York	NY	state
Washington DC	DC	capitol
Washington	WA	state

Table consist of columns and rows

Census

state	sex	age	pop2000	pop2008
New York	F	0	120355	122194
New York	F	1	118219	119661
New York	F	2	119577	116413

Tables can be related

Census

state	sex	age	pop2000	pop2008
New York	F	0	120355	122194
New York	F	1	118219	119661
New York	F	2	119577	116413

State_Fact

name	abbreviation	type
New York	NY	state
Washington DC	DC	capitol
Washington	WA	state

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Connecting to a Database

Meet SQLAlchemy

- Two Main Pieces
  - Core (Relational Model focused)
  - ORM (User Data Model focused)

There are many types of databases

- SQLite
- PostgreSQL
- MySQL
- MS SQL
- Oracle
- Many more

Connecting to a database

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

In [2]: engine = create_engine('sqlite:///census_nyc.sqlite')

In [3]: connection = engine.connect()
```

- Engine: common interface to the database from SQLAlchemy
- Connection string: All the details required to find the database (and login, if necessary)

A word on connection strings

- `'sqlite:///census_nyc.sqlite'`

  Driver+Dialect Filename

What's in your database?

- Before querying your database, you'll want to know what is in it: what the tables are, for example:

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

In [2]: engine = create_engine('sqlite:///census_nyc.sqlite')

In [3]: print(engine.table_names())
Out[3]: ['census', 'state_fact']
```

Reflection

- Reflection reads database and builds SQLAlchemy Table objects

```
In [1]: from sqlalchemy import MetaData, Table

In [2]: metadata = MetaData()

In [3]: census = Table('census', metadata, autoload=True,
autoload_with=engine)

In [4]: print(repr(census))
Out[4]:
Table('census', MetaData(bind=None), Column('state',
VARCHAR(length=30), table=<census>), Column('sex',
VARCHAR(length=1), table=<census>), Column('age', INTEGER(),
table=<census>), Column('pop2000', INTEGER(), table=<census>),
Column('pop2008', INTEGER(), table=<census>), schema=None)
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Introduction to SQL Queries

SQL Statements

- Select, Insert, Update & Delete data
- Create & Alter data

Basic SQL querying

- `SELECT column_name FROM table_name`
- `SELECT pop2008 FROM People`
- `SELECT * FROM People`

Basic SQL querying

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

In [2]: engine = create_engine('sqlite:///census_nyc.sqlite')

In [3]: connection = engine.connect()

In [4]: stmt = 'SELECT * FROM people'

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

In [6]  results = result_proxy.fetchall()
```

ResultProxy vs ResultSet

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

In [6]: results = result_proxy.fetchall()
```

- ResultProxy
- ResultSet

Handling ResultSets

```
In [1]: first_row = results[0]

In [2]: print(first_row)
Out[2]: ('Illinois', 'M', 0, 89600, 95012)

In [4]: print(first_row.keys())
Out[4]: ['state', 'sex', 'age', 'pop2000', 'pop2008']

In [6]: print(first_row.state)
Out[6]: 'Illinois'
```

SQLAlchemy to Build Queries

- Provides a Pythonic way to build SQL statements
- Hides differences between backend database types

SQLAlchemy querying

```
In [4]: from sqlalchemy import Table, MetaData

In [5]: metadata = MetaData()

In [6]: census = Table('census', metadata, autoload=True,
autoload_with=engine)

In [7]: stmt = select([census])

In [8]: results = connection.execute(stmt).fetchall()
```

SQLAlchemy Select Statement

- Requires a list of one or more Tables or Columns
- Using a table will select all the columns in it

```
In [9]: stmt = select([census])

In [10]: print(stmt)
Out[10]: 'SELECT * from CENSUS'
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Congratulations!

You already

- Know about the relational model
- Can make basic SQL queries

Coming up next...

- Beef up your SQL querying skills
- Learn how to extract all types of useful information from your databases using SQLAlchemy
- Learn how to create and write to relational databases
- Deep dive into the US census dataset!

INTRODUCTION TO DATABASES IN PYTHON

See you in the next chapter!

INTRODUCTION TO DATABASES IN PYTHON

Filtering and Targeting Data

Where Clauses

```
In [1]: stmt = select([census])

In [2]: stmt = stmt.where(census.columns.state ==
            'California')

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

In [4]: for result in results:
   ...:     print(result.state, result.age)
Out[4]:
California 0
California 1
California 2
California 3
California 4
California 5
...
```

Where Clauses

- Restrict data returned by a query based on boolean conditions
- Compare a column against a value or another column
- Often used comparisons: '==', '<=', '>=', or '!='

Expressions

- Provide more complex conditions than simple operators
- Eg. `in_()`, `like()`, `between()`
- Many more in documentation
- Available as method on a Column

Expressions

```
In [1]: stmt = select([census])

In [2]: stmt = stmt.where(
            census.columns.state.startswith('New'))

In [3]: for result in connection.execute(stmt):
   ...:     print(result.state, result.pop2000)
Out[3]:
New Jersey 56983
New Jersey 56686
New Jersey 57011
...
```

Conjunctions

- Allow us to have multiple criteria in a where clause
- Eg. `and_()`, `not_()`, `or_()`

Conjunctions

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

In [2]: stmt = select([census])

In [3]: stmt = stmt.where(
   ...:     or_(census.columns.state == 'California',
   ...:         census.columns.state == 'New York'
   ...:     )
   ...: )

In [4]: for result in connection.execute(stmt):
   ...:     print(result.state, result.sex)
Out[4]:
New York M
…
California F
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Ordering Query Results

Order by Clauses

- Allows us to control the order in which records are returned in the query results
- Available as a method on statements `order_by()`

Order by Ascending

```
In [1]: print(results[:10])
Out[1]: [('Illinois',), …]

In [3]: stmt = select([census.columns.state])

In [4]: stmt = stmt.order_by(census.columns.state)

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

In [6]: print(results[:10])
Out[6]: [('Alabama',), …]
```

Order by Descending

- Wrap the column with `desc()` in the `order_by()` clause

Order by Multiple

- Just separate multiple columns with a comma
- Orders completely by the first column
- Then if there are duplicates in the first column, orders by the second column
- repeat until all columns are ordered

Order by Multiple

```
In [6]: print(results)
Out[6]: ('Alabama', 'M')

In [7]: stmt = select([census.columns.state,
   ...:     census.columns.sex])

In [8]: stmt = stmt.order_by(census.columns.state,
   ...:     census.columns.sex)

In [9]: results = connection.execute(stmt).first()

In [10]: print(results)
Out[10]:('Alabama', 'F')
('Alabama', 'F')
…
('Alabama', 'M')
```

INTRODUCTION TO DATABASES IN PYTHON

Let's practice!

INTRODUCTION TO DATABASES IN PYTHON

Counting, Summing and Grouping Data

SQL Functions

- E.g. Count, Sum
- from sqlalchemy import func
- More efficient than processing in Python
- Aggregate data

Sum Example

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

In [2]: stmt = select([func.sum(census.columns.pop2008)])

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

In [4]: print(results)
Out[4]: 302876613
```

Group by

- Allows us to group row by common values

Group by

```
In [1]: stmt = select([census.columns.sex,
   ...:     func.sum(census.columns.pop2008)
   ...: ])

In [2]: stmt = stmt.group_by(census.columns.sex)

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

In [4]: print(results)
Out[4]: [('F', 153959198), ('M', 148917415)]
```

Group by

- Supports multiple columns to group by with a pattern similar to `order_by()`
- Requires all selected columns to be grouped or aggregated by a function

Group by Multiple

```
In [1]: stmt = select([census.columns.sex,
   ...:     census.columns.age,
   ...:     func.sum(census.columns.pop2008)
   ...: ])

In [2]: stmt = stmt.group_by(census.columns.sex,
   ...:     census.columns.age)

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

In [3]: print(results)
Out[3]:
[('F', 0, 2105442), ('F', 1, 2087705), ('F', 2, 2037280), ('F', 3,
2012742), ('F', 4, 2014825), ('F', 5, 1991082), ('F', 6, 1977923),
('F', 7, 2005470), ('F', 8, 1925725), …
```

Handling ResultSets from Functions

- SQLAlchemy auto generates “column names” for functions in the ResultSet
- The column names are often `func_#` such as `count_1`
- Replace them with the `label()` method

Using label()

```
In [1]: print(results[0].keys())
Out[1]: ['sex', u'sum_1']

In [2]: stmt = select([census.columns.sex,
   ...:     func.sum(census.columns.pop2008).label(
   ...:         'pop2008_sum')
   ...: ])

In [3]: stmt = stmt.group_by(census.columns.sex)

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

In [5]: print(results[0].keys())
Out[5]: ['sex', 'pop2008_sum']
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

SQLAlchemy and Pandas for Visualization

SQLAlchemy and Pandas

- DataFrame can take a SQLAlchemy ResultSet
- Make sure to set the DataFrame columns to the ResultSet keys

DataFrame Example

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame(results)

In [3]: df.columns = results[0].keys()

In [4]: print(df)
Out[4]:
    sex  pop2008_sum
0     F      2105442
1     F      2087705
2     F      2037280
3     F      2012742
4     F      2014825
5     F      1991082
```

Graphing

- We can graph just like we would normally

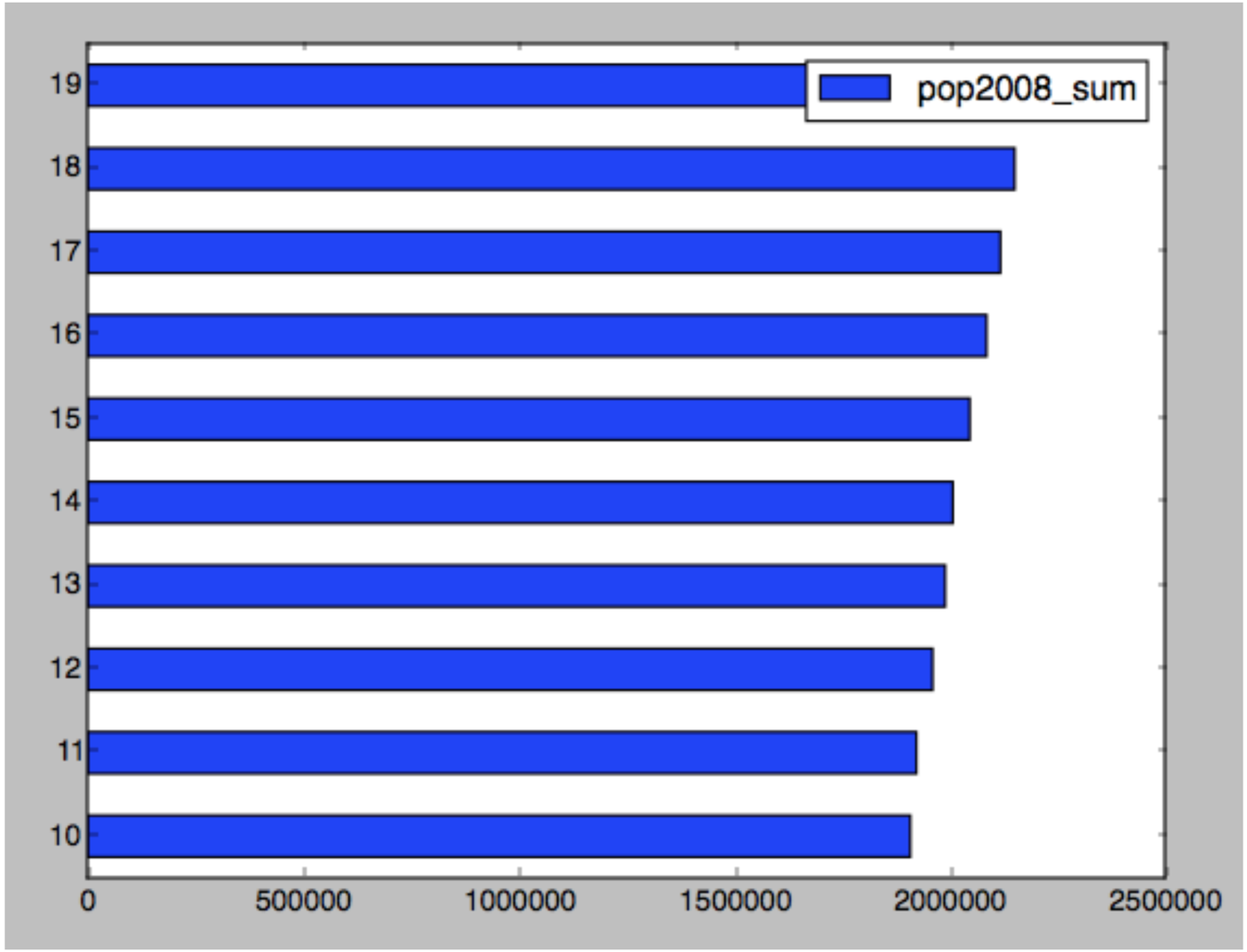
Graphing Example

```
In [1]: import matplotlib.pyplot as plt

In [2]: df[10:20].plot.barh()

In [3]: plt.show()
```

Graphing Output



INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Calculating Values in a Query

Math Operators

- addition +
- subtraction -
- multiplication *
- division /
- modulus %
- Work differently on different data types

Calculating Difference

```
In [1]: stmt = select([census.columns.age,
   ...:     (census.columns.pop2008-
   ...:     census.columns.pop2000).label('pop_change')
   ...: ])

In [2]: stmt = stmt.group_by(census.columns.age)

In [3]: stmt = stmt.order_by(desc('pop_change'))

In [4]: stmt = stmt.limit(5)

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

In [6]: print(results)
Out[6]: [(61, 52672), (85, 51901), (54, 50808), (58, 45575), (60,
44915)]
```

Case Statement

- Used to treat data differently based on a condition
- Accepts a list of conditions to match and a column to return if the condition matches
- The list of conditions ends with an else clause to determine what to do when a record doesn't match any prior conditions

Case Example

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

In [2]: stmt = select([
   ...:     func.sum(
   ...:         case([
   ...:             (census.columns.state == 'New York',
   ...:              census.columns.pop2008)
   ...:         ], else_=0))])

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

In [4]: print(results)
Out[4]:[(19465159,)]
```

Cast Statement

- Converts data to another type
- Useful for converting
  - integers to floats for division
  - strings to dates and times
- Accepts a column or expression and the target Type

Percentage Example

```
In [1]: from sqlalchemy import case, cast, Float

In [2]: stmt = select([
   ...:     (func.sum(
   ...:         case([
   ...:             (census.columns.state == 'New York',
   ...:              census.columns.pop2008)
   ...:         ], else_=0)) /
   ...:      cast(func.sum(census.columns.pop2008),
   ...:           Float) * 100).label('ny_percent')])

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

In [4]: print(results)
Out[4]: [(Decimal('6.4267619765'),)]
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

SQL Relationships

Relationships

- Allow us to avoid duplicate data
- Make it easy to change things in one place
- Useful to break out information from a table we don’t need very often

Relationships

Census

state	sex	age	pop2000	pop2008
New York	F	0	120355	122194
New York	F	1	118219	119661
New York	F	2	119577	116413

State_Fact

name	abbreviation	type
New York	NY	state
Washington DC	DC	capitol
Washington	WA	state

Automatic Joins

```
In [1]: stmt = select([census.columns.pop2008,
   ...:     state_fact.columns.abbreviation])

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

In [3]: print(results)
Out[3]: [(95012, u'IL'),
 (95012, u'NJ'),
 (95012, u'ND'),
 (95012, u'OR'),
 (95012, u'DC'),
 (95012, u’WI'),
…
```

Join

- Accepts a Table and an optional expression that explains how the two tables are related
- The expression is not needed if the relationship is predefined and available via reflection
- Comes immediately after the `select()` clause and prior to any `where()`, `order_by` or `group_by()` clauses

Select_from

- Used to replace the default, derived FROM clause with a join
- Wraps the join() clause

Select_from Example

```
In [1]: stmt = select([func.sum(census.columns.pop2000)])

In [2]: stmt = stmt.select_from(census.join(state_fact))

In [3]: stmt = stmt.where(state_fact.columns.circuit_court
             == '10')

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

In [5]: print(result)
Out[5]: 14945252
```

Joining Tables without Predefined Relationship

- Join accepts a Table and an optional expression that explains how the two tables are related
- Will only join on data that match between the two columns
- Avoid joining on columns of different types

Select_from Example

```
In [1]: stmt = select([func.sum(census.columns.pop2000)])

In [2]: stmt = stmt.select_from(
   ...:     census.join(state_fact, census.columns.state
   ...:     == state_fact.columns.name))

In [3]: stmt = stmt.where(
   ...:     state_fact.columns.census_division_name ==
   ...:     'East South Central')

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

In [5]: print(result)
Out[5]: 16982311
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Working with Hierarchical Tables

Hierarchical Tables

- Contain a relationship with themselves
- Commonly found in:
  - Organizational
  - Geographic
  - Network
  - Graph

Hierarchical Tables - Example

Employees

id	name	job	manager
1	Johnson	Admin	6
2	Harding	Manager	9
3	Taft	Sales I	2
4	Hoover	Sales I	2

Hierarchical Tables - alias()

- Requires a way to view the table via multiple names
- Creates a unique reference that we can use

Querying Hierarchical Data

```
In [1]: managers = employees.alias()

In [2]: stmt = select(
   ...:     [managers.columns.name.label('manager'),
   ...:      employees.columns.name.label('employee')])

In [3]: stmt = stmt.select_from(employees.join(
   ...:     managers, managers.columns.id ==
   ...:     employees.columns.manager)

In [4]: stmt = stmt.order_by(managers.columns.name)

In [5]: print(connection.execute(stmt).fetchall())
Out[5]: [(u'FILLMORE', u'GRANT'),
 (u'FILLMORE', u'ADAMS'),
 (u'HARDING', u'TAFT'), ...
```

Group_by and Func

- It's important to target `group_by()` at the right alias
- Be careful with what you perform functions on
- If you don't find yourself using both the alias and the table name for a query, don't create the alias at all

Querying Hierarchical Data

```
In [1]: managers = employees.alias()

In [2]: stmt = select([managers.columns.name,
   ...:     func.sum(employees.columns.sal)])

In [3]: stmt = stmt.select_from(employees.join(
   ...:     managers, managers.columns.id ==
   ...:     employees.columns.manager)

In [4]: stmt = stmt.group_by(managers.columns.name)

In [5]: print(connection.execute(stmt).fetchall())
Out[5]: [(u'FILLMORE', Decimal('96000.00')),
 (u'GARFIELD', Decimal('83500.00')),
 (u'HARDING', Decimal('52000.00')),
 (u'JACKSON', Decimal('197000.00'))]
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Handling Large ResultSets

Dealing with Large ResultSets

- `fetchmany()` lets us specify how many rows we want to act upon
- We can loop over `fetchmany()`
- It returns an empty list when there are no more records
- We have to close the ResultProxy afterwards

Fetching Many Rows

```
In [1]: while more_results:
    ...:     partial_results = results_proxy.fetchmany(50)
    ...:     if partial_results == []:
    ...:         more_results = False
    ...:
    ...:     for row in partial_results:
    ...:         state_count[row.state] += 1

In [2]: results_proxy.close()
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

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))}
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

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
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

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

INTRODUCTION TO DATABASES IN PYTHON

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]: []
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Census Case Study

Census Case Study

- Preparing SQLAlchemy and the Database
- Loading Data into the Database
- Solving Data Science Problems with Queries

Part 1: Preparing SQLAlchemy and the Database

- Create an Engine and MetaData object

```
In [1]: from sqlalchemy import create_engine, MetaData

In [2]: engine = create_engine('sqlite:///census_nyc.sqlite')

In [3]: metadata = MetaData()
```

Part 1: Preparing SQLAlchemy and the Database

- Create and save the census table

```
In [4]: from sqlalchemy import (Table, Column, String,
   ...:     Integer, Decimal, Boolean)

In [5]: employees = Table('employees', metadata,
   ...:     Column('id', Integer()),
   ...:     Column('name', String(255)),
   ...:     Column('salary', Decimal()),
   ...:     Column('active', Boolean()))

In [6]: metadata.create_all(engine)
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Populating the Database

Part 2: Populating the Database

- Load a CSV file into a values list

```
In [7]: values_list = []

In [8]: for row in csv_reader:
   ...:     data = {'state': row[0], 'sex': row[1],
   ...:             'age': row[2], 'pop2000': row[3],
   ...:             'pop2008': row[4]}
   ...:     values_list.append(data)
```

Part 2: Populating the Database

- Insert the values list into the census table

```
In  [9]: from sqlalchemy import insert

In [10]: stmt = insert(employees)

In [11]: result_proxy = connection.execute(stmt,
    ...:     values_list)

In [12]: print(result_proxy.rowcount)
Out[12]: 2
```

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

Introduction to Databases in Python

Querying the Database

Part 3: Answering Data Science Questions with Queries

- Determine Average Age for Males and Females

```
In [13]: from sqlalchemy import select

In [14]: stmt = select([census.columns.sex,
    ...:     (func.sum(census.columns.pop2008 *
    ...:      census.columns.age) /
    ...:      func.sum(census.columns.pop2008)
    ...:     ).label('average_age')])

In [15]: stmt = stmt.group_by('census.columns.sex')

In [16]: results = connection.execute(stmt).fetchall()
```

Part 3: Answering Data Science Questions with Queries

- Determine the percentage of Females for each state

```
In [17]: from sqlalchemy import case, cast, Float

In [18]: stmt = select([
    ...:     (func.sum(
    ...:         case([
    ...:             (census.columns.state == 'New York',
    ...:              census.columns.pop2008)
    ...:         ], else_=0)) /
    ...:      cast(func.sum(census.columns.pop2008),
    ...:           Float) * 100).label('ny_percent')])
```

Part 3: Answering Data Science Questions with Queries

- Determine the top 5 states by population change from 2000 to 2008

```
In [19]: stmt = select([census.columns.age,
    ...:     (census.columns.pop2008-
    ...:     census.columns.pop2000).label('pop_change')
    ...: ])

In [20]: stmt = stmt.order_by('pop_change')

In [21]: stmt = stmt.limit(5)
```

Introduction to Databases in Python

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Congratulations!