**11- Advanced-Data-Storage-and-Retrieval**

SEE LESSON PLANS root:tongs2 = username:pw for mySQL server

Today’s 9/15/18 Objectives:

SQLAlchemy library for Python.

Class Objectives from LessonPlan.md

* connect to a SQL database using SQLAlchemy
* perform basic SQL queries using engine.execute()
* create Python classes and objects
* CRUD = create, read, update, and delete data

from a SQL database using SQLAlchemy's ORM

\* Interact with SQL databases with Python

\* Introduce SQLAlchemy

\* Understand what ORM stands for and how to use it

\* Introduction to cloud-based databases

\* Introduction to SQLite

\* Introduction to Object-Oriented (OO) programming

Technologies installed:

conda install -c anaconda sqlalchemy

conda install -c anaconda pymysql

conda install -c anaconda sqlite

sqlite viewer: <https://sqlitebrowser.org/>

this allowed us to view a db we created in 03-Ins\_ToSQL

Heroku…

Settings for dmarcus9-example1

Connection Info

Connection String

mysql://pecsey73ly4s6ges:abir3npwr365rwh4@mgs0iaapcj3p9srz.cbetxkdyhwsb.us-east-1.rds.amazonaws.com:3306/sx6r6ac8oxckzmc0

PreActivity 1: (5 minutes)

What is an ORM?

Ties a concept of an object to a physical thing

Middle layer bt our actions & database

**Object-relational mapping** (**ORM**, **O/RM**, and **O/R mapping tool**) in [computer science](https://en.wikipedia.org/wiki/Computer_science) is a [programming](https://en.wikipedia.org/wiki/Computer_programming) technique for converting data between incompatible [type systems](https://en.wikipedia.org/wiki/Type_system) using [object-oriented](https://en.wikipedia.org/wiki/Object-oriented) programming languages. This creates, in effect, a "virtual [object database](https://en.wikipedia.org/wiki/Object_database)" that can be used from within the programming language.

There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to construct their own ORM tools.

What are the benefits to using an ORM?

Can save programming time

Ex: sqlalchemy

Used in web apps

Data analysis not as much

From lesson plan:

* Being able to work across different SQL dialects using the same basic Python query
* Being able to create command line interfaces which allow users to construct SQL queries without having to know the language.

What are some of the disadvantages to using an ORM?

Overhead…lots of calls to db

SQLAlchemy is a Python ORM - Object Relational Mapper - for SQL databases.

SQLAlchemy essentially allows Python developers to use external scripts to modify SQL databases.

* there are many different SQL dialects out there. MySQL, PostgreSQL, & SQLite...etc.
* SQLAlchemy is able to bridge the gaps between these similar dialects. This means that a single Python program written using SQLAlchemy can be used to affect different databases without having to write new queries each time.

<https://docs.sqlalchemy.org/en/latest/dialects/>

this specific link leads to a list of the SQL dialects that are compatible with SQLAlchemy.

To the left side of the page, find the complete documentation for the SQLAlchemy library.

* SQLite is a SQL dialect that shares much the same syntax as MySQL but that it is entirely serverless.
* How can a database be serverless? Well, SQLite reads and writes directly to ordinary disk files which can in-turn be stored on a computer's hard drive. This makes it amazingly easy to perform tests with and share between users.
  1. Basic SQL connection

1. # SQL Alchemy...to create a connection to a SQL database,
2. # the `create\_engine` module will need to be imported.
3. from sqlalchemy import create\_engine
4. # PyMySQL...For Python to read MySQL databases, `pymysql` must also
5. # be imported and then the `pymysql.install\_as\_MySQLdb()` method must be used.
6. import pymysql
7. pymysql.install\_as\_MySQLdb()
8. # Create Engine and Pass in MySQL Connection...After importing in all of the necessary libraries/modules,
9. # the connection engine can be created using the `create\_engine()` method and passing a connection string into it.
10. engine = create\_engine("mysql://k5xunpkmojyzse51:ifagg1gp7e2xyapi@ffn96u87j5ogvehy.cbetxkdyhwsb.us-east-1.rds.amazonaws.com:3306/tq6h098h0ym00zp6")
11. # The connection string is composed of 5 distinct parts and
12. # can be constructed by following the following syntax:

`<Dialect>://<Username>:<Password>@<Host Address>:<Port>/<Database>`

1. # Query All Records in the the Database
2. # data stored within a variable and then looped through so as to print out the rows
3. data = engine.execute("SELECT \* FROM Census\_Data")
4. for record in data:
5. print(record)

1.2-Stu\_IceCreamStore

\*need to open mySQL workbench, set up database, then run it from Jupyter notebook,

putting in db name in step 3

# SQL Alchemy

from sqlalchemy import create\_engine

# PyMySQL

import pymysql

pymysql.install\_as\_MySQLdb()

# Create Engine and Pass in MySQL Connection

engine = create\_engine("mysql://root:stongs2@localhost/icecream\_db")

# Query All Records in the the Database

data = engine.execute("SELECT \* FROM icecreamstore")

for record in data:

print(record)

# Query Single Record in the the Database

data = engine.execute("SELECT \* FROM icecreamstore WHERE Price>=1.25;")

for record in data:

print(record)

1.3-Ins\_ReadSQL

One of the most impressive aspects of SQLAlchemy is how it integrates with Pandas.

# Pandas

import pandas as pd

# SQL Alchemy

from sqlalchemy import create\_engine

# PyMySQL

import pymysql

pymysql.install\_as\_MySQLdb()

# Create Engine and Pass in MySQL Connection

engine = create\_engine("mysql://k5xunpkmojyzse51:ifagg1gp7e2xyapi@ffn96u87j5ogvehy.cbetxkdyhwsb.us-east-1.rds.amazonaws.com:3306/tq6h098h0ym00zp6")

conn = engine.connect()

# Query All Records in the the Database

data = pd.read\_sql("SELECT \* FROM Census\_Data", conn)

# Preview the Data

data.head()

1.3-Ins\_ToSQL

NOT WORKING?

1.4-Stu\_ReadAllTheSQLs

# Pandas

import pandas as pd

# SQL Alchemy

from sqlalchemy import create\_engine

# PyMySQL

import pymysql

pymysql.install\_as\_MySQLdb()

# Create Engine and Pass in MySQL Connection

engine = create\_engine("mysql://k5xunpkmojyzse51:ifagg1gp7e2xyapi@ffn96u87j5ogvehy.cbetxkdyhwsb.us-east-1.rds.amazonaws.com:3306/tq6h098h0ym00zp6")

conn = engine.connect()

# Query All Records in the the City Table

census\_data = pd.read\_sql("SELECT \* FROM Census\_Data", conn)

# Query All Records in the Zip Table

zip\_data = pd.read\_sql("SELECT \* FROM Zip\_Census", conn)

census\_data.head()

zip\_data.head()

# Merge the columns

combined\_data = pd.merge(census\_data, zip\_data, on="CityState", how="inner")

# Combined Data

combined\_data.head()

1.5-Ins\_Preview\_SQL\_Alchemy

An **integrated development environment** (**IDE**) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools, and a debugger. Most modern IDEs have intelligent code completion.

# Dependencies

# ----------------------------------

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

from sqlalchemy import Column, Integer, String, Float

# Create Dog and Cat Classes

# ----------------------------------

class Dog(Base):

\_\_tablename\_\_ = 'dog'

id = Column(Integer, primary\_key=True)

name = Column(String(255))

color = Column(String(255))

age = Column(Integer)

class Cat(Base):

\_\_tablename\_\_ = 'cat'

id = Column(Integer, primary\_key=True)

name = Column(String(255))

color = Column(String(255))

age = Column(Integer)

# Create a Specific Instance of the Dog and Cat classes

# ----------------------------------

dog = Dog(name="Fido", color='Brown', age=4)

cat = Cat(name="Whiskers", color="Gray", age=7)

# Create Database Connection

# ----------------------------------

# PyMySQL

import pymysql

pymysql.install\_as\_MySQLdb()

engine = create\_engine("mysql://k5xunpkmojyzse51:ifagg1gp7e2xyapi@ffn96u87j5ogvehy.cbetxkdyhwsb.us-east-1.rds.amazonaws.com:3306/tq6h098h0ym00zp6")

conn = engine.connect()

# Create a "Metadata" Layer That Abstracts our SQL Database

# ----------------------------------

Base.metadata.create\_all(engine)

# Use this to clear out the db

# ----------------------------------

# Base.metadata.drop\_all(engine)

# Create a Session Object to Connect to DB

# ----------------------------------

from sqlalchemy.orm import Session

session = Session(bind=engine)

# Add Records to the Appropriate DB

# ----------------------------------

session.add(dog)

session.add(cat)

session.commit()

# Query the Tables

# ----------------------------------

dog\_list = session.query(Dog)

for doggy in dog\_list:

print(doggy.name)

cat\_list = session.query(Cat)

for kitty in cat\_list:

print(kitty.name)

**Classes**

are essentially blueprints for Python objects. In other words, they allow developers to create organized variables with keys, values, and methods on the fly.

SQLAlchemy uses Python classes as its primary means to communicate and make changes to SQL databases. This is what makes SQLAlchemy an ORM as it uses objects to map changes to SQL tables/databases.

1.6-Ins\_Classes

Object oriented programming (OOP) is a style of coding based around the concept of "objects". These objects may contain data, often known as attributes, and functions, often known as methods.

Python is a class-based programming language. This means that objects can be created according to user-created blueprints, thus allowing developers to rapidly create objects of similar structure/purpose but with differing values.

# Define a class

class Dog():

# Utilize the Python constructor to initialize the object

def \_\_init\_\_(self, name, color):

self.name = name

self.color = color

# Create an instance of a class

dog = Dog('Fido', 'brown')

# Print the object's attributes

print(dog.name)

print(dog.color)

1.7-Stu\_Surfer\_Class

# Define the Surfer Class

class Surfer():

# Initialize the Surfer constructor

def \_\_init\_\_(self, name, hometown, rank):

self.name = name + " " + "Dude"

self.hometown = hometown + " " + "Waves"

self.rank = rank

# Create an instance of the Surfer Class

surfer = Surfer('Kelly Slater', 'Cocoa Beach', 1)

# Print the object's attributes

print(surfer.name)

print(surfer.hometown)

print(surfer.rank)

# ----BONUS----

# Variable to keep track of changes to while loop

go = True

# While loop runs so long as go is True

while go:

# Ask for user input and store answers within variables

name = input("What is the surfer's name? ")

hometown = input("What is the surfer's hometown? ")

rank = int(input("What is the surfer's rank? "))

# Create a new instance of the Surfer class using these values

surfer = Surfer(name, hometown, rank)

# Print the object's attributes

print(surfer.name)

print(surfer.hometown)

print(surfer.rank)

# Check to see if the user would like to continue

check = input("Would you like to continue? (y/n) ")

if(check.lower() == "y"):

go = True

else:

go = False

1.8-Ins\_Classes\_With\_Methods

# Define the Film class

class Film():

# A required function to initialize the class object

def \_\_init\_\_(self, name, length, release\_year, language):

self.name = name

self.length = length

self.release\_year = release\_year

self.language = language

# An object belonging to the Film class

star\_wars = Film("Star Wars", 121, 1977, "English")

# Define the Expert class

class Expert():

# A required function to initialize the class object

def \_\_init\_\_(self, name):

self.name = name

# A method that takes another object as its argument

def boast(self, obj):

# Print out Expert object's name

print("Hi. My name is", self.name)

# Print out the name of the Film class object

print("I know a lot about", obj.name)

print("It is", obj.length, "minutes long")

print("It was released in", obj.release\_year)

print("It is in", obj.language)

# An object belonging to the Film class

superfan = Expert("George Lucas")

superfan.boast(star\_wars)

1.9-Stu\_Surfer\_Class\_Extended

# Define the Surfer Class

class Surfer():

# Keep track of surfer count as they are created

surferCount = 0

# Constructor

# --------------------------------------------------------------------------------

# Initialize the surfer and assign each surfer a new surfer count upon creation

def \_\_init\_\_(self, name, hometown, rank, wipeouts=0):

self.name = name

self.hometown = hometown

self.rank = rank

self.wipeouts = wipeouts

Surfer.surferCount += 1

# Class Methods

# --------------------------------------------------------------------------------

# Prints what number surfer they are based on when they were created

def surfer\_count(self):

print("Total surfers shredding waves %d" % Surfer.surferCount)

# Prints out simple string

def speak(self):

print("Hang loose bruh!")

# Interpolates based on their attributes

def biography(self):

print(f"My name is {self.name}, I am from {self.hometown} and rank{self.rank}, I've wiped out {self.wipeouts} times!")

# Check how many wipeouts and print out a statement

def cheer(self):

if self.wipeouts == 0:

print('I totally rock man, no wipeouts!')

else:

print('Bummer bruh, keep on keeping on!')

# Create Surfers

# --------------------------------------------------------------------------------

surfer = Surfer('Kelly Slater', 'Cocoa Beach', 1,)

print(surfer.name)

print(surfer.hometown)

print(surfer.rank)

print(surfer.wipeouts)

surfer.speak()

surfer.biography()

surfer.cheer()

surfer.surfer\_count()

surfer = Surfer('John Breezy', 'Spring Lake', 1, 10)

print(surfer.name)

print(surfer.hometown)

print(surfer.rank)

print(surfer.wipeouts)

surfer.speak()

surfer.biography()

surfer.cheer()

surfer.surfer\_count()

1.10-Ins\_SQL\_Alchemy\_Revisited

# Dependencies

# ----------------------------------

# Imports the method used for connecting to DBs

from sqlalchemy import create\_engine

# Imports the methods needed to abstract classes into tables

from sqlalchemy.ext.declarative import declarative\_base

# Allow us to declare column types

from sqlalchemy import Column, Integer, String, Float

# PyMySQL

import pymysql

pymysql.install\_as\_MySQLdb()

# Create Dog and Cat Classes

# ----------------------------------

# Sets an object to utilize the default declarative base in SQL Alchemy

Base = declarative\_base()

# Creates Classes which will serve as the anchor points for our Tables

class Dog(Base):

\_\_tablename\_\_ = 'dog'

id = Column(Integer, primary\_key=True)

name = Column(String(255))

color = Column(String(255))

age = Column(Integer)

class Cat(Base):

\_\_tablename\_\_ = 'cat'

id = Column(Integer, primary\_key=True)

name = Column(String(255))

color = Column(String(255))

age = Column(Integer)

# Create a Specific Instance of the Dog and Cat classes

# ----------------------------------

# Calls the Pet Constructors to create "Dog" and "Cat" objets

dog = Dog(name='Rex', color='Brown', age=4)

cat = Cat(name="Felix", color="Gray", age=7)

# Create Database Connection

# ----------------------------------

# Creates a connection to our DB using the MySQL Connect Engine

# engine = create\_engine("sqlite://k5xunpkmojyzse51:ifagg1gp7e2xyapi@ffn96u87j5ogvehy.cbetxkdyhwsb.us-east-1.rds.amazonaws.com:3306/tq6h098h0ym00zp6")

engine = create\_engine("sqlite:///pets.sqlite")

conn = engine.connect()

# Create a "Metadata" Layer That Abstracts our SQL Database

# ----------------------------------

# Create (if not already in existence) the tables associated with our classes.

Base.metadata.create\_all(engine)

# Use this to clear out the db

# ----------------------------------

# Base.metadata.drop\_all(engine)

# Create a Session Object to Connect to DB

# ----------------------------------

# Session is a temporary binding to our DB

from sqlalchemy.orm import Session

session = Session(bind=engine)

# Add Records to the Appropriate DB

# ----------------------------------

# Use the SQL ALchemy methods to run simple "INSERT" statements using the classes and objects

session.add(dog)

session.add(cat)

session.commit()

# Query the Tables

# ----------------------------------

# Perform a simple query of the database

dog\_list = session.query(Dog)

for doggy in dog\_list:

print(doggy.name)

cat\_list = session.query(Cat)

for kitty in cat\_list:

print(kitty.name)

1.11-Stu\_Surfer\_SQL

# Import SQL Alchemy

from sqlalchemy import create\_engine

# Import PyMySQL (Not needed if mysqlclient is installed)

import pymysql

pymysql.install\_as\_MySQLdb()

# Import and establish Base for which classes will be constructed

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

# Import modules to declare columns and column data types

from sqlalchemy import Column, Integer, String, Float

# Create Surfer and Board classes

# ----------------------------------

class Surfer(Base):

\_\_tablename\_\_ = 'surfers'

id = Column(Integer, primary\_key=True)

name = Column(String(255))

hometown = Column(String(255))

wipeouts = Column(Integer)

rank = Column(Integer)

class Board(Base):

\_\_tablename\_\_ = 'surfboards'

id = Column(Integer, primary\_key=True)

surfer\_id = Column(Integer)

board\_name = Column(String(255))

color = Column(String(255))

length = Column(Integer)

# Create specific instances of the Surfer and Board classes

# ----------------------------------

# Create a new surfer named "Bruno"

surfer = Surfer(name='Bruno', hometown="LA", rank=10)

# Create a new board and associate it with a surfer's ID

board = Board(surfer\_id=1, board\_name="Awwwyeah", color="Blue", length=68)

# Create Database Connection

# ----------------------------------

# Establish Connection to MySQL

# engine = create\_engine('mysql://root:mypassword@data-bootcamp-rutgers.cn6jzamkgcqr.us-west-2.rds.amazonaws.com:3306/surfingdb')

engine = create\_engine("sqlite:///surfer.sqlite")

conn = engine.connect()

# Create both the Surfer and Board tables within the database

Base.metadata.create\_all(conn)

# To push the objects made and query the server we use a Session object

from sqlalchemy.orm import Session

session = Session(bind=engine)

# Add "Bruno" to the current session

session.add(surfer)

# Add "Awwwyeah" to the current session

session.add(board)

# Commit both objects to the database

session.commit()

# Query the database and collect all of the surfers in the Surfer table

surfer\_list = session.query(Surfer)

for bro in surfer\_list:

print(bro.name)

print(bro.hometown)

print(bro.rank)

9/17/18 Mon. day 2 – AD

In sqlalchemy, you need to create a “class” to create a table

A class is a data structure/framework that holds attributes, methods that are expected to be used repeatedly (allows you to write less code)

Similar to function & to a table, but not visualized

Sqlalchemy is a package to make working with mySQL databases easier in Python

SQLAlchemy is a Python ORM - Object Relational Mapper - for SQL databases.

In much more basic terms, SQLAlchemy essentially allows Python developers to use external scripts to modify SQL databases.

* there are many different SQL dialects out there. MySQL, PostgreSQL, and SQLite...
* SQLAlchemy is able to bridge the gaps between these similar dialects. This means that a single Python program written using SQLAlchemy can be used to affect different databases without having to write new queries each time.

<https://docs.sqlalchemy.org/en/latest/dialects/>

**### Class Objectives**

\* use the SQLAlchemy ORM to create classes that model tables.

\* perform database CRUD operations using the SQLAlchemy ORM.

\* reflect existing databases.

\* use the SQLAlchemy Inspector to view table names in the database.

\* plot the query results from the ORM.

2.1-Ins\_Basic\_Querying

from sqlalchemy import create\_engine

from sqlalchemy import Column, Integer, String, Float

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

class BaseballPlayer(Base):

\_\_tablename\_\_ = "player"

player\_id = Column(String, primary\_key=True)

birth\_year = Column(Integer)

birth\_month = Column(Integer)

birth\_day = Column(Integer)

birth\_country = Column(String)

birth\_state = Column(String)

birth\_city = Column(String)

name\_first = Column(String)

name\_last = Column(String)

name\_given = Column(String)

weight = Column(Integer)

height = Column(Integer)

bats = Column(String)

throws = Column(String)

debut = Column(String)

final\_game = Column(String)

# Create Database Connection

engine = create\_engine('sqlite:///../Resources/database.sqlite')

Base.metadata.create\_all(engine)

from sqlalchemy.orm import Session

session = Session(bind=engine)

# Print all of the player names in the database

players = session.query(BaseballPlayer)

for player in players:

print(player.name\_given)

# Find the number of players from the USA

usa = session.query(BaseballPlayer).\

filter(BaseballPlayer.birth\_country == 'USA').count()

print("There are {} players from the USA".format(usa))

# Find those players who were born before 1990

born\_before\_1990 = session.query(BaseballPlayer).\

filter(BaseballPlayer.birth\_year < 1990).count()

print("{} players were born before 1990".format(born\_before\_1990))

# Find those players from the USA who were born after 1989

born\_after\_1989 = session.query(BaseballPlayer).\

filter(BaseballPlayer.birth\_year > 1989).filter(BaseballPlayer.birth\_country == "USA").\

count()

print("{} USA players were born after 1989".format(born\_after\_1989))

2.2-Stu\_SharkSearch

NOT WORKING IN NOTEBOOK…?

from sqlalchemy import create\_engine

import pymysql

pymysql.install\_as\_MySQLdb()

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

from sqlalchemy import Column, Integer, String, Float

from config import dbuser, dbpasswd, dburi, dbport, dbname

from sqlalchemy import create\_engine

engine = create\_engine(f"mysql://{dbuser}:{dbpasswd}@{dburi}:{dbport}/{dbname}")

Base.metadata.create\_all(engine)

# In case your sql server doesn't require a password

# from sqlalchemy import create\_engine

# engine = create\_engine(f"mysql://{dbuser}@{dburi}:{dbport}/{dbname}")

# Base.metadata.create\_all(engine)

from sqlalchemy.orm import Session

session = Session(bind=engine)

# create your shark class

class Sharks(Base):

\_\_tablename\_\_ = 'sharks'

id = Column(Integer, primary\_key=True)

case\_number = Column(String)

date = Column(String)

year = Column(Integer)

type = Column(String)

country = Column(String)

area = Column(String)

location= Column(String)

activity = Column(String)

name = Column(String)

sex = Column(String)

age = Column(Integer)

injury = Column(String)

fatal\_y\_n = Column(String)

time = Column(String)

species = Column(String)

investigator\_or\_source = Column(String)

pdf = Column(String)

original\_order = Column(Integer)

# print all locations of shark attacks

attacks = session.query(Sharks)

for attack in attacks:

print(attack.location)

# find the number of provoked attacks

provoked = session.query(Sharks).filter\_by(type='provoked').count()

print(provoked)

# find the number of attacks in USA

usa = session.query(Sharks).filter\_by(country='USA').count()

print(usa)

# find the number of attacks in 2017

year\_2017 = session.query(Sharks).filter\_by(year=2017).count()

print(year\_2017)

# find the number of attacks while surfing

surfing = session.query(Sharks).filter\_by(activity='surfing').count()

print(surfing)

# find the number of fatal attacks

fatal = session.query(Sharks).filter\_by(fatal\_y\_n='Y').count()

print(fatal)

# find the number of fatal attacks while surfing

fatal\_surfing = session.query(Sharks).\

filter\_by(fatal\_y\_n='Y').\

filter(Sharks.area == "Eastern Cape Province").count()

print(fatal\_surfing)

# find the number of fatal attacks in 2017 in Australia

fatal\_surfing = session.query(Sharks).\

filter\_by(fatal\_y\_n='Y').\

filter(Sharks.country == "MOZAMBIQUE").\

filter(Sharks.activity == 'Spearfishing').count()

print(fatal\_surfing)

2.3-Ins\_Basic\_Updating

from sqlalchemy import create\_engine, Column, Integer, String

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

# Define our pet table

class Pet(Base):

\_\_tablename\_\_ = 'pet'

id = Column(Integer, primary\_key=True)

name = Column(String)

type = Column(String)

age = Column(Integer)

# Right now, this table only exists in python and not in the actual database

Base.metadata.tables

# Create our database engine

engine = create\_engine('sqlite:///pets.sqlite')

# This is where we create our tables in the database

Base.metadata.create\_all(engine)

# The ORM’s “handle” to the database is the Session.

from sqlalchemy.orm import Session

session = Session(engine)

# CREATE

# Note that adding to the session does not update the table. It queues up those queries.

session.add(Pet(name='Justin Timbersnake', type='snek', age=2))

session.add(Pet(name='Pawtrick Stewart', type='good boy', age=10))

session.add(Pet(name='Godzilla', type='iguana', age=1))

session.add(Pet(name='Marshmallow', type='polar bear', age=4))

# The data hasn't been added yet

engine.execute('select \* from pet').fetchall()

# We can use the new attribute to see the queue of data ready to go into the database

session.new

# commit() flushes whatever remaining changes remain to the database, and commits the transaction.

session.commit()

# Nothing new to add

session.new

# query the database

session.query(Pet.name, Pet.type, Pet.age).all()

# UPDATE

# Create a query and then run update on it

pet = session.query(Pet).filter\_by(name="Marshmallow").first()

pet.age += 1

# For modifications, we can use the dirty attribute

session.dirty

# Commit Transaction

session.commit()

# Session is up-to-date

session.dirty

session.query(Pet.id, Pet.name, Pet.type, Pet.age).all()

# DELETE

# Create a query and then delete the row collected

pet = session.query(Pet).filter\_by(id=4).delete()

session.commit()

session.query(Pet.id, Pet.name, Pet.type, Pet.age).all()

2.4-Par\_CruddyDB

# Import SQL Alchemy

from sqlalchemy import create\_engine

# Import and establish Base for which classes will be constructed

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

# Import modules to declare columns and column data types

from sqlalchemy import Column, Integer, String, Float

# Create the Garbage class

class Garbage(Base):

\_\_tablename\_\_ = 'garbage\_collection'

id = Column(Integer, primary\_key=True)

item = Column(String(255))

weight = Column(Float)

collector = Column(String(255))

# Create a connection to a SQLite database

engine = create\_engine('sqlite:///garbage.db')

# Create the garbage\_collection table within the database

Base.metadata.create\_all(engine)

# To push the objects made and query the server we use a Session object

from sqlalchemy.orm import Session

session = Session(bind=engine)

# Create some instances of the Garbage class

garbage\_one = Garbage(item="Sofa", weight=90.5, collector="Jacob")

garbage\_two = Garbage(item="Broken TV", weight=10.75, collector="Paul")

garbage\_three = Garbage(item="Burger", weight=0.55, collector="Phil")

# Add these objects to the session

session.add(garbage\_one)

session.add(garbage\_two)

session.add(garbage\_three)

# Commit the objects to the database

session.commit()

# Update two rows of data

update\_one = session.query(Garbage).filter(Garbage.id == 1).first()

update\_one.collector = "Jacob Deming"

update\_two = session.query(Garbage).filter(Garbage.id == 2).first()

update\_two.weight = 11.25

# Commit the updates to the database

session.commit()

# Delete the row with the lowest weight

session.query(Garbage).filter(Garbage.id == 3).delete()

# Commit the delete to the database

session.commit()

# Collect all of the items and print their information

items = session.query(Garbage)

for item in items:

print("-"\*12)

print(f"id: {item.id}")

print(f"item: {item.item}")

print(f"weight: {item.weight}")

print(f"collector: {item.collector}")

2.5-Ins\_**Reflection**

\* as data analysts, developers often need to analyze already existing data sources. This would mean having to create SQLAlchemy classes according to a table's columns by hand every single time.

\* Thankfully SQLAlchemy provides tools for automatically creating ORM classes from an existing database.

\* these tools will load the data from an existing database and use that data to infer how to write ORM classes for use "automagically".

\* this process is called **\*\*reflection\*\***.

4 steps:

\* 1) import `automap\_base` in from the SQLAlchemy library

\* 2) create an `engine` against the existing database that should be reflected

\* 3) create a `Base` by calling `Base = automap\_base()`

\* 4) call `Base.prepare` with the `engine` from Step 2 and `reflect=True` as its parameters

# Python SQL toolkit and Object Relational Mapper (1)

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine

# Create engine using the `demographics.sqlite` database file (2)

engine = create\_engine("sqlite:///../Resources/dow.sqlite")

# Declare a Base using `automap\_base()` (3)

Base = automap\_base()

# Use the Base class to reflect the database tables (4)

Base.prepare(engine, reflect=True)

# Print all of the classes mapped to the Base

Base.classes.keys()

# Assign the dow class to a variable called `Dow`

Dow = Base.classes.dow

# Create a session

session = Session(engine)

# Display the row's columns and data in dictionary format

first\_row = session.query(Dow).frst()

first\_row.\_\_dict\_\_

# Use the session to query Dow table and display the first 5 trade volumes

for row in session.query(Dow.stock, Dow.volume).limit(15).all():

print(row)

2.6-Stu\_ReflectingOnSQL

# Python SQL toolkit and Object Relational Mapper

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine

# Create engine using the `demographics.sqlite` database file

engine = create\_engine("sqlite:///../Resources/demographics.sqlite")

# Declare a Base using `automap\_base()`

Base = automap\_base()

# Use the Base class to reflect the database tables

Base.prepare(engine, reflect=True)

# Print all of the classes mapped to the Base

Base.classes.keys()

# Assign the demographics class to a variable called `Demographics`

Demographics = Base.classes.demographics

# Create a session

session = Session(engine)

# Use the session to query Demographics table and display the first 5 locations

for row in session.query(Demographics, Demographics.location).limit(5).all():

print(row)

# BONUS: Query and print the number of unique Locations

# Hints: Look into counting and grouping operations in SQLAlchemy

locations = session.query(Demographics).group\_by(Demographics.location).count()

print(locations)

2.7-Ins\_Exploration

\* The inspector tool allows SQLAlchemy developers to look through a connected database and explore its contents.

\* Unlike session queries, the inspector is primarily used to look up tables, columns, and datatypes.

Looking up the specific values stored within a table is where queries should be used.

# Import SQLAlchemy `automap` and other dependencies

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, inspect

# Create the connection engine

engine = create\_engine("sqlite:///../Resources/dow.sqlite")

# Create the inspector and connect it to the engine

inspector = inspect(engine)

# Collect the names of tables within the database

inspector.get\_table\_names()

# Using the inspector to print the column names within the 'dow' table and its types

columns = inspector.get\_columns('dow')

for column in columns:

print(column["name"], column["type"])

2.8-Stu\_SalaryExplore - Inspector

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, inspect

# Create the connection engine

engine = create\_engine("sqlite:///../Resources/database.sqlite")

# Create the inspector and connect it to the engine

inspector = inspect(engine)

# Collect the names of tables within the database

inspector.get\_table\_names()

# Using the inspector to print the column names within the 'Salaries' table and its types

columns = inspector.get\_columns('Salaries')

for column in columns:

print(column["name"], column["type"])

2.9-Par\_EmojiPlotting

# Import Matplot lib

import matplotlib

from matplotlib import style

style.use('seaborn')

import matplotlib.pyplot as plt

import pandas as pd

# Import SQLAlchemy `automap` and other dependencies here

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, inspect, func

# Create an engine for the `emoji.sqlite` database

engine = create\_engine("sqlite:///../Resources/emoji.sqlite", echo=False)

# Use the Inspector to explore the database and print the table names

inspector = inspect(engine)

inspector.get\_table\_names()

# Use Inspector to print the column names and types

columns = inspector.get\_columns('emoji')

for c in columns:

print(c['name'], c["type"])

# Use `engine.execute` to select and display the first 10 rows from the emoji table

engine.execute('SELECT \* FROM emoji LIMIT 10').fetchall()

# Reflect Database into ORM class

Base = automap\_base()

Base.prepare(engine, reflect=True)

Emoji = Base.classes.emoji

# Start a session to query the database

session = Session(engine)

# Query Emojis for `emoji\_char`, `emoji\_id`, and `score` and save the query into results

results = session.query(Emoji.emoji\_char, Emoji.emoji\_id, Emoji.score).\

order\_by(Emoji.score.desc()).all()

# Unpack the `emoji\_id` and `scores` from results and save into separate lists

emoji\_id = [result[1] for result in results[:10]]

scores = [int(result[2]) for result in results[:10]]

# Create a horizontal bar chart and plot the `emoji\_id` on the y-axis and the `score` on the x-axis

# Challenge: Try to plot the scores in descending order on the graph (The largest score is at the top)

fig, ax = plt.subplots()

ypos = range(1, len(scores)+1)

ax.barh(ypos, scores[::-1])

ax.set\_xlabel("score")

ax.set\_ylabel("emoji")

ax.set\_yticks(ypos)

ax.set\_yticklabels(emoji\_id[::-1])

ax.set\_title("Emoji Scores")

fig.tight\_layout()

plt.show()

# Load the results into a PANDAS dataframe. Set the index to the `emoji\_id`

df = pd.DataFrame(results[:10], columns=['emoji\_char', 'emoji\_id', 'score'])

df.set\_index('emoji\_id', inplace=True, )

df.head(10)

# Plot the dataframe as a horizontal bar chart using pandas plotting

df.iloc[::-1].plot.barh(title="emoji ranking")

plt.tight\_layout()

plt.show()

# BONUS: Use Pandas `read\_sql\_query` to load a query statement directly into the DataFrame

stmt = session.query(Emoji).\

order\_by(Emoji.score.desc()).statement

df2 = pd.read\_sql\_query(stmt, session.bind)

df2.head(10)

9/19/18 Mon. day 3 – Manuel

**Introduction to Flask & Serving Data with APIs**

the fundamentals of the web and client-server architecture;

how to use Flask to create a database-backed server;

and how to use the same to design and implement API endpoints.

**### Class Objectives**

\* use Flask to create and run a server

\* define endpoints using Flask's @app.route decorator

\* learn to extract query variable path values from GET requests

\* use variable paths to execute database queries on behalf of the client

\* learn to return JSONified query results from API endpoints

Flask app is a way to use Python for web design

Already Installed thru terminal

<http://flask.pocoo.org/docs/0.12/quickstart/#a-minimal-application>

Todays Objectives:

\* Querying dates

\* Flask - Backend server written in Python

\* return JSON

\* return html

\* accept query parameters (variable rules)

\* integrate with SQLAlchemy

Resources:

SQLALchemy Notes

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>

SQLAlchemy Joins

<http://docs.sqlalchemy.org/en/latest/orm/extensions/automap.html#specifying-classes-explicitly>

<https://www.pythonsheets.com/notes/python-sqlalchemy.html#object-relational-join-two-tables>

Flask (webserver) routes - Connects function to url endpoint

<http://flask.pocoo.org/docs/1.0/quickstart/#routing>

JSONify - servers and clients can only communicate by passing strings between each other. A common data format for this is JSON (JavaScript Object Notation)

Variable Rules - You can add variable sections to a URL by marking sections with <variable\_name>.

<http://flask.pocoo.org/docs/1.0/quickstart/#variable-rules>

3.1-Ins\_Joins

# Python SQL toolkit and Object Relational Mapper

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, inspect

engine = create\_engine("sqlite:///../Resources/mammal\_masses.sqlite", echo=False)

# Reflect Database into ORM classes

Base = automap\_base()

Base.prepare(engine, reflect=True)

Base.classes.keys()

# Map Europe class

EA = Base.classes.ea

# Map North American class

NA = Base.classes.na

# create a session

session = Session(engine)

# Filters are the "WHERE" clause for your select statement.

# filter North American mammals whose genus is "Antilocapra"

# query, loop over and print out animals.

mammals = session.query(NA).filter(NA.genus == 'Antilocapra').all()

for mammal in mammals:

print("Family: {0}, Genus: {1}".format(mammal.family, mammal.genus))

inspector = inspect(engine)

inspector.get\_table\_names()

# Get a list of column names and types

columns = inspector.get\_columns('ea')

for c in columns:

print(c['name'], c["type"])

session.query(EA.sporder, NA.sporder).limit(100).all()

same\_sporder = session.query(EA, NA).filter(EA.sporder == NA.sporder).limit(10).all()

for record in same\_sporder:

(ea, na) = record

print(ea.sporder)

print(na.sporder)

# Return all animals from EA and NA belonging to the same sporder.

# This JOINs the data in the two tables together into a single dataset (here in the form of a tuple).

# Note: We are going to limit the results to 10 for printing

sel = [EA.family, EA.genus, EA.species, NA.family, NA.genus, NA.species]

same\_sporder = session.query(\*sel).filter(EA.sporder == NA.sporder).limit(10).all()

for record in same\_sporder:

(ea\_fam, ea\_gen, ea\_spec, na\_fam, na\_gen, na\_spec) = record

print(

f"The European animal '{ea\_fam} {ea\_gen} {ea\_spec}'"

f"belongs to the same sporder as the North American animal '{na\_fam} {na\_gen} {na\_spec}'.")

3.2-Ins\_Dates

<https://docs.sqlalchemy.org/en/latest/dialects/sqlite.html>

# Python SQL toolkit and Object Relational Mapper

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, inspect, func

engine = create\_engine("sqlite:///../Resources/dow.sqlite", echo=False)

inspector = inspect(engine)

inspector.get\_table\_names()

# Get a list of column names and types

columns = inspector.get\_columns('dow')

for c in columns:

print(c['name'], c["type"])

engine.execute('SELECT \* FROM dow LIMIT 5').fetchall()

# Reflect Database into ORM class

Base = automap\_base()

Base.prepare(engine, reflect=True)

Dow = Base.classes.dow

session = Session(engine)

# Total dates

session.query(func.count(Dow.date)).all()

# Earliest Date

session.query(Dow.date).order\_by(Dow.date).first()

# Latest Date

session.query(Dow.date).order\_by(Dow.date.desc()).first()

session.query(Dow.date).\

filter(Dow.date > '2011-03-01').\

order\_by(Dow.date).all()

import datetime as dt

# Print a date object and a datetime object

print(dt.date.today())

print(dt.date(2017, 1 ,31))

print(dt.datetime.utcnow())

print(dt.datetime(2017, 1, 31))

# date 1 week ago from today

week\_ago = dt.date.today() - dt.timedelta(days=7)

# Query for the Dow closing price for `CSCO` 1 week before `2011-04-08` using the datetime library

query\_date = dt.date(2011, 4, 8) - dt.timedelta(days=7)

print("Query Date: ", query\_date)

session.query(Dow.date, Dow.close\_price).\

filter(Dow.stock == 'CSCO').\

filter(Dow.date == query\_date).all()

# Parse out just the day from the datetime object

dt.date.today().strftime("%d")

# Query for all dates matching the

# following date string in the format `%d`

date\_str = "14"

session.query(Dow.date).\

filter(func.strftime("%d", Dow.date) == date\_str).all()

3.3-Stu\_Dates

import matplotlib

from matplotlib import style

style.use('fivethirtyeight')

import matplotlib.pyplot as plt

import pandas as pd

# Python SQL toolkit and Object Relational Mapper

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, inspect, func

engine = create\_engine("sqlite:///../Resources/dow.sqlite", echo=False)

engine.execute('SELECT \* FROM dow LIMIT 5').fetchall()

inspector = inspect(engine)

columns = inspector.get\_columns('dow')

for c in columns:

print(c['name'], c["type"])

# Reflect Database into ORM class

Base = automap\_base()

Base.prepare(engine, reflect=True)

Dow = Base.classes.dow

session = Session(engine)

# Query for the stock and average prices (open, high, low, close)

# for all stock in the month of May

# Sort the result by stock name

sel = [Dow.stock,

func.avg(Dow.open\_price),

func.avg(Dow.high\_price),

func.avg(Dow.low\_price),

func.avg(Dow.close\_price)]

may\_averages = session.query(\*sel).\

filter(func.strftime("%m", Dow.date) == "05").\

group\_by(Dow.stock).\

order\_by(Dow.stock).all()

may\_averages

# Plot the Results in a Matplotlib bar chart

df = pd.DataFrame(may\_averages, columns=['stock', 'open\_avg', 'high\_avg', 'low\_avg', 'close\_avg'])

df.set\_index('stock', inplace=True)

df.plot.bar()

plt.tight\_layout()

plt.show()

# Design a query to calculate the PTP for stock `IBM` after May, 2011

import datetime as dt

import numpy as np

date = dt.datetime(2011, 5, 31)

results = session.query(Dow.high\_price - Dow.low\_price).\

filter(Dow.date > date).filter(Dow.stock == 'IBM').all()

ptps = list(np.ravel(results))

# List Comprehension Solution

# ptps = [result[0] for result in results]

ptps

# Load the query into a dataframe, set the index to the date, and plot the ptps

import numpy as np

fig, ax = plt.subplots()

x = range(len(ptps))

ax.boxplot(ptps, patch\_artist=True)

ax.set\_title('IBM PTPs')

fig.tight\_layout()

plt.show()

\* the Internet is built on model of *\_clients\_* (whoever asks for information)

requesting data from *\_servers\_*.

\* when a person uses an API to fetch data,

we have a tendency to consider the *\_person\_* the client.

\* *\_strictly speaking\_*, this isn't accurate: A *\_****program****\_* makes a request on behalf of the person.

\* a **\*\*browser\*\*** is an example of a program that makes requests on behalf of a user.

\* the same holds true for **servers**:

A *\_server\_* is simply a process running on a remote machine, that listens for, and knows how to respond to, incoming requests.

\* The point to emphasize is that a server is, essentially, *\_just a program\_*.

\* when we create an API for others to use, the code they write acts as a *\_client\_* to our API server.

\* we have no control over the code our consumers write.

\* this means that, as API developers, **\*\*we do not write client code\*\***.

\* this means we will focus on writing the code that runs the server.

\* this is the code responsible for retrieving and returning whatever data that users requests.

\* [Flask](http://flask.pocoo.org/) is the tool that we'll use to implement our server.

\* it is an extremely intuitive library that makes it easy to develop APIs for distributing our data.

\* servers are programs that *\_listen\_* for *\_requests\_* to particular *\_URLS\_*, or **\*\*endpoints\*\***.

\* Flask makes creating and starting a server trivial, and defining endpoints, easy:

It takes less than 10 lines of code to define a functional index route!

An API is like a waiter connecting you to the kitchen at a restaurant…

API’s connect Kayak.com to United’s database…

.py file… run out of Visual Studio Code,

View – Terminal,

folder window on the left – rt. click on file opened – Open in Terminal

In terminal, to run file, type:

python ‘app.py’ = ‘name of file’

3.4-Ins\_First\_Steps\_with\_Flask

# 1. import Flask (a Python module)

from flask import Flask

# 2. Create a web app, being sure to pass \_\_name\_\_

app = Flask(\_\_name\_\_)

# 3. Define what to do when a user hits the index route

@app.route("/")

def home():

print("Server received request for 'Home' page...")

return "Welcome to my 'Home' page!"

# 4. Define what to do when a user hits the /about route

@app.route("/about")

def about():

print("Server received request for 'About' page...")

return "Welcome to my 'About' page!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

passing `debug=True` makes development *\_much\_* easier, but, in production, best practices demand that `debug` *\_must* **\*\*always\*\*** *be false\_*...

in public-facing projects, otherwise they can easily leak data about their users and application accidentally.

**Note**: this above is using flask / python to create web pages.

“print” statement is in terminal

“return” statement is on the page

to illustrate and emphasize the relationship between the *\_client\_*—which receives a request handler's return value—and the *\_server\_*—where the functions associated with the response to a request are actually executed.

3.5-Stu\_Hello\_Web

# 1. Import Flask

from flask import Flask

# 2. Create an app

app = Flask(\_\_name\_\_)

# 3. Define static routes

@app.route("/")

def index():

return "Hello, world!"

@app.route("/about")

def about():

name = "Peleke"

location = "Tien Shan"

return f"My name is {name}, and I live in {location}."

@app.route("/contact")

def contact():

email = "peleke@example.com"

return f"Questions? Comments? Complaints? Shoot an email to {email}."

# 4. Define main behavior

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

\* All of the the routes we've written thus far have returned *\_string\_* responses.

\* The APIs we've dealt with do *\_not\_* return raw text: rather, they return JSON data.

\* Fortunately, Python dictionaries map naturally to JSON.

\* Flask has a built-in method to automatically convert a dictionary into a properly formatted JSON response: `jsonify`.

\* Remind students that routes must return HTTP responses.

\* This means we can't simply return the dictionary itself.

\* We can use `jsonify` to create an HTTP response with the dictionary data we want to send back to the client.

3.6-Ins\_Jsonify

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

hello\_dict = {"Hello": "World!"}

@app.route("/")

def home():

return "Hi"

# doesn’t work…

@app.route("/normal")

def normal():

return hello\_dict

@app.route("/jsonified")

def jsonified():

return jsonify(hello\_dict)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

3.7-Stu\_Justice\_League

create an API route that returns the superhero name and real name for every member of the

DC Comics Justice League.

from flask import Flask, jsonify

# Dictionary of Justice League

justice\_league\_members = [

{"superhero": "Aquaman", "real\_name": "Arthur Curry"},

{"superhero": "Batman", "real\_name": "Bruce Wayne"},

{"superhero": "Cyborg", "real\_name": "Victor Stone"},

{"superhero": "Flash", "real\_name": "Barry Allen"},

{"superhero": "Green Lantern", "real\_name": "Hal Jordan"},

{"superhero": "Superman", "real\_name": "Clark Kent/Kal-El"},

{"superhero": "Wonder Woman", "real\_name": "Princess Diana"}

]

# Flask Setup

#################################################

app = Flask(\_\_name\_\_)

# Flask Routes

#################################################

@app.route("/api/v1.0/justice-league")

def justice\_league():

"""Return the justice league data as json"""

return jsonify(justice\_league\_members)

@app.route("/")

def welcome():

return (

f"Welcome to the Justice League API!<br/>"

f"Available Routes:<br/>"

f"/api/v1.0/justice-league")

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

\* our current API is only capable of returning the *\_entire\_* Justice League data set.

\* It would be better if users could specify a particular character of interest.

\* ideally, consumers would be able to specify a character of interest in the URL, and expect either:

\* A JSON response with the character data, if it's in the data set; or

\* A JSON response with error information, indicating that the server couldn't find the character the user requested.

3.8-Ins\_Variable\_Rule

from flask import Flask, jsonify

justice\_league\_members = [

{"superhero": "Aquaman", "real\_name": "Arthur Curry"},

{"superhero": "Batman", "real\_name": "Bruce Wayne"},

{"superhero": "Cyborg", "real\_name": "Victor Stone"},

{"superhero": "Flash", "real\_name": "Barry Allen"},

{"superhero": "Green Lantern", "real\_name": "Hal Jordan"},

{"superhero": "Superman", "real\_name": "Clark Kent/Kal-El"},

{"superhero": "Wonder Woman", "real\_name": "Princess Diana"}

]

# Flask Setup

#################################################

app = Flask(\_\_name\_\_)

# Flask Routes

#################################################

@app.route("/api/v1.0/justice-league")

def justice\_league():

"""Return the justice league data as json"""

return jsonify(justice\_league\_members)

@app.route("/")

def welcome():

return (

f"Welcome to the Justice League API!<br/>"

f"Available Routes:<br/>"

f"/api/v1.0/justice-league<br/>"

f"/api/v1.0/justice-league/Arthur%20Curry<br/>"

f"/api/v1.0/justice-league/Bruce%20Wayne<br/>"

f"/api/v1.0/justice-league/Victor%20Stone<br/>"

f"/api/v1.0/justice-league/Barry%20Allen<br/>"

f"/api/v1.0/justice-league/Hal%20Jordan<br/>"

f"/api/v1.0/justice-league/Clark%20Kent/Kal-El<br/>"

f"/api/v1.0/justice-league/Princess%20Diana")

@app.route("/api/v1.0/justice-league/<real\_name>")

def justice\_league\_character(real\_name):

"""Fetch the Justice League character whose real\_name matches

the path variable supplied by the user, or a 404 if not."""

canonicalized = real\_name.replace(" ", "").lower()

for character in justice\_league\_members:

search\_term = character["real\_name"].replace(" ", "").lower()

if search\_term == canonicalized:

return jsonify(character)

return jsonify({"error": f"Character with real\_name {real\_name} not found."}), 404

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

3.9-Stu\_Variable\_Rule

from flask import Flask, jsonify

justice\_league\_members = [

{"superhero": "Aquaman", "real\_name": "Arthur Curry"},

{"superhero": "Batman", "real\_name": "Bruce Wayne"},

{"superhero": "Cyborg", "real\_name": "Victor Stone"},

{"superhero": "Flash", "real\_name": "Barry Allen"},

{"superhero": "Green Lantern", "real\_name": "Hal Jordan"},

{"superhero": "Superman", "real\_name": "Clark Kent/Kal-El"},

{"superhero": "Wonder Woman", "real\_name": "Princess Diana"}

]

# Flask Setup

#################################################

app = Flask(\_\_name\_\_)

# Flask Routes

#################################################

@app.route("/api/v1.0/justice-league")

def justice\_league():

"""Return the justice league data as json"""

return jsonify(justice\_league\_members)

@app.route("/")

def welcome():

return (

f"Welcome to the Justice League API!<br/>"

f"Available Routes:<br/>"

f"/api/v1.0/justice-league<br/>"

f"/api/v1.0/justice-league/superhero/batman")

@app.route("/api/v1.0/justice-league/real\_name/<real\_name>")

def justice\_league\_by\_real\_name(real\_name):

"""Fetch the Justice League character whose real\_name matches

the path variable supplied by the user, or a 404 if not."""

canonicalized = real\_name.replace(" ", "").lower()

for character in justice\_league\_members:

search\_term = character["real\_name"].replace(" ", "").lower()

if search\_term == canonicalized:

return jsonify(character)

return jsonify({"error": f"Character with real\_name {real\_name} not found."}), 404

@app.route("/api/v1.0/justice-league/superhero/<superhero>")

def justice\_league\_by\_superhero\_\_name(superhero):

"""Fetch the Justice League character whose superhero matches

the path variable supplied by the user, or a 404 if not."""

canonicalized = superhero.replace(" ", "").lower()

for character in justice\_league\_members:

search\_term = character["superhero"].replace(" ", "").lower()

if search\_term == canonicalized:

return jsonify(character)

return jsonify({"error": "Character not found."}), 404

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

\* any useful API must make queries against data sets much too large to load into memory.

\* we'll next see how to perform ORM queries within their Flask routes.

3.10-Ins\_Flask\_with\_ORM

import numpy as np

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, func

from flask import Flask, jsonify

# Database Setup

#################################################

engine = create\_engine("sqlite:///titanic.sqlite")

# reflect an existing database into a new model

Base = automap\_base()

# reflect the tables

Base.prepare(engine, reflect=True)

# Save reference to the table

Passenger = Base.classes.passenger

# Create our session (link) from Python to the DB

session = Session(engine)

# Flask Setup

#################################################

app = Flask(\_\_name\_\_)

# Flask Routes

#################################################

@app.route("/")

def welcome():

"""List all available api routes."""

return (

f"Available Routes:<br/>"

f"/api/v1.0/names<br/>"

f"/api/v1.0/passengers")

@app.route("/api/v1.0/names")

def names():

"""Return a list of all passenger names"""

# Query all passengers

results = session.query(Passenger.name).all()

# Convert list of tuples into normal list

all\_names = list(np.ravel(results))

return jsonify(all\_names)

@app.route("/api/v1.0/passengers")

def passengers():

"""Return a list of passenger data including the name, age, and sex of each passenger"""

# Query all passengers

results = session.query(Passenger).all()

# Create a dictionary from the row data and append to a list of all\_passengers

all\_passengers = []

for passenger in results:

passenger\_dict = {}

passenger\_dict["name"] = passenger.name

passenger\_dict["age"] = passenger.age

passenger\_dict["sex"] = passenger.sex

all\_passengers.append(passenger\_dict)

return jsonify(all\_passengers)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

\* we will be analyzing invoice data from the [Chinook database]

(https://chinookdatabase.codeplex.com/wikipage?title=Chinook\_Schema&referringTitle=Home).

\* we will design SQLAlchemy ORM queries to answer specific questions about the invoice data.

3.11-Stu\_Chinook

# Ignore SQLITE warnings related to Decimal numbers in the Chinook database

import warnings

warnings.filterwarnings('ignore')

# Import Dependencies

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine

from sqlalchemy import func

# Create an engine for the chinook.sqlite database

engine = create\_engine("sqlite:///../Resources/chinook.sqlite", echo=False)

# Reflect Database into ORM classes

Base = automap\_base()

Base.prepare(engine, reflect=True)

Base.classes.keys()

# Save a reference to the invoices table as `Invoices`

Invoices = Base.classes.invoices

# Create a database session object

session = Session(engine)

# List all of the countries found in the invoices table

session.query(Invoices.BillingCountry).group\_by(Invoices.BillingCountry).all()

# Alternative Solution using `distinct`

# session.query(Invoices.BillingCountry).distinct().all()

# Design a query that lists the invoices totals for each billing country

# and sort the output in descending order.

session.query(Invoices.BillingCountry, func.sum(Invoices.Total)).\

group\_by(Invoices.BillingCountry).\

order\_by(func.sum(Invoices.Total).desc()).all()

# Save a reference to the invoice\_items table as `Items`

Items = Base.classes.invoice\_items

# List all of the Billing Postal Codes for the USA.

results = session.query(Invoices.BillingPostalCode).\

filter(Invoices.BillingCountry == 'USA').group\_by(Invoices.BillingPostalCode).all()

results

# Calculate the Item Totals (sum(UnitPrice \* Quantity)) for the USA

session.query(func.sum(Items.UnitPrice \* Items.Quantity)).\

filter(Invoices.InvoiceId == Items.InvoiceId).\

filter(Invoices.BillingCountry == 'USA').scalar()

# Calculate the Item Totals `sum(UnitPrice \* Quantity)` for each Billing Postal Code in the USA

# Sort the results in descending order by Total

session.query(Invoices.BillingPostalCode, func.sum(Items.UnitPrice \* Items.Quantity)).\

filter(Invoices.InvoiceId == Items.InvoiceId).\

filter(Invoices.BillingCountry == 'USA').\

group\_by(Invoices.BillingPostalCode).\

order\_by(func.sum(Items.UnitPrice \* Items.Quantity).desc()).all()

**### Objectives** from StudentGuide.md

\* Connect to a SQL database using SQLAlchemy.

\* Perform basic SQL queries using engine.execute().

\* Create Python classes and objects.

\* Create, read, update, and delete data from a SQL database using SQLAlchemy's ORM.

\* Reflect existing databases.

\* Use the SQLAlchemy ORM to create classes that model tables.

\* Use the ORM define relationships and foreign key constraints.

\* Use joins to query related data.

\* Use Flask to create and run a server.

\* Define endpoints using Flask's @app.route decorator.

\* Extract query variable path values from GET requests.

\* Use variable paths to execute database queries on behalf of the client.

\* Return JSONified query results from API endpoints.

**### Helpful Links**

\* [Essential SQLAlchemy Book](http://shop.oreilly.com/product/0636920035800.do)

\* [Introduction to SQLAlchemy](https://www.youtube.com/watch?v=woKYyhLCcnU)

\* [Flask Mega-Tutorial](<https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>)

W3schools.com (from Anselmo)

<https://www.pythonsheets.com/notes/python-sqlalchemy.html>