```
Boiler Plate
Changeable
```
#Comments & explanations

Connecting to a Database

- **Database path format**:

```
"<dialect>://<username>:<password>@<host address>:<port>/<database>"
```

  - example

```
"mysql://root:<password>@localhost/sportsdb"
```

- **Sqlite**: unlike regular MySQL which obscures how its data is scored, Sqlite saves each database into a single local .sqlite file. It loads in a similar way pandas loads a csv, does not require host address/port, etc. The standard free release does not allow password protection of the database. If a .sqlite file exists at the specified file path, SQLite would read and modify the existing database. If a .sqlite file does not exist at the specified path, SQLite would create a new database.
  - **SQLite path format**

```
"sqlite:///<directory/filename>"
```

  - example

```
"sqlite:///../Resources/testDb.sqlite"
```

```
"sqlite:///newDb.sqlite"
```

- **Connect to a database and basic querying**

```
from sqlalchemy import create_engine
import pymysql
pymysql.install_as_MySQLdb()

dialect = 'mysql'
user = 'root'
password = '<pwd>'
host = 'localhost'
port = 3306
database = '<db name>'

engine =
create_engine(f'{dialect}://{user}:{password}@{host}:{port}/{database}')
# uses the function create_engine from sqlalchemy to create an object
# engine is the object that will initiate the queries to the database

data = engine.execute("<SQLSTATEMENT>;")
# the object engine uses the method .execute to connect to the predefined database and run the
query inside the parentheses, store the retrieved info in an object called data

for record in data:  #print each line of query results stored in data
    print(record)
```

- **Connect to a data base and load data into a pandas dataframe**

```
import pandas as pd
from sqlalchemy import create_engine
import pymysql
pymysql.install_as_MySQLdb()

engine = create_engine("<DB_PATH>")
conn = engine.connect()

df = pd.read_sql("<SQLSTATEMENT>", conn)
```

Introduction to Object Oriented Programming in Python

*apologies to my good friends in the bootcamp for making them part of my little demo >.< *

We have learned a few data types in python so far, e.g. *lists*, *strings*, or *dictionaries.* We've also learned some of the things we can do to those data types, e.g., `.append()` for lists, `.split()` for strings, `.pop()` for dictionaries. These commands are called "methods"

Defining a new class is basically creating a new data type, and you can define functions, aka "methods", that can be performed on the data capsulated inside this new data type.

A class is a blueprint, a template for its objects.

- Any objects in that class will have the same format as the structures defined in the class
  - Lists, dictionaries, strings, etc, are python's built-in classes

An object is a specific instance of class.

- It contains a collection of properties, expressed as variables and functions
- These variables are called "instance variables" or "attributes"
- These functions are called "methods", they can only operate on objects of this class, they can't be called any other way
  - `.append() .split() .pop()` etc are "methods" specific to their respective class.

Here's an example of class:
Say I want to build a new class that can be used to encapsulate information about people in our bootcamp.

```
class Bootcampbuddy():
    def __init__(self, firstname, current_age, current_job, interest):
        self.name = firstname
        self.age = current_age
        self.job = current_job
        self.interest = interest
        self.skills = ["VBA", "Python"]

    def introduce_self(self):
        print("Hi, my name is " + self.name)

    def learn_new_things(self, newskill):
        self.skills.append(newskill)

    def what_do_you_know(self):
        skill_string = ', '.join(self.skills)
        print(f"{self.name} now knows {skill_string}")
```

`__init__` is an initializer, which "initializes" an object of this class. The class Bootcampbuddy is a template to be filled in with information about a person in our bootcamp.

`self` is the designation for the object that would be created from this class template.

From the class definition, we can see that to create an object in this class, we need to give the input argument `full_name, current_age, current_job, interest`. Those arguments get fed into the initiation function, and become attributes of the object. `.name .age .job` etc are a collection of variables, aka attributes, stored in the object.

Note that `self.skills` doesn't refer to an input argument. It's basically saying, any object created from the class *Bootcampbuddy*, automatically has skills "VBA" and "Python".

`introduce_self`, `learn_new_things`, `what_do_you_know` are functions within the class, aka "methods". These methods can only be performed on objects belonging to this class.

When defining a method inside a class, you have to include `self` as an input argument. This is because when a method is called, python silently passes in the object itself in as an argument, without user input. We've seen this behavior before:

`df.head()` When we work with pandas dataframe, the dataframe itself is a object belonging to a class that pandas has defined. When we call the method `.head()`, although we don't put in an argument inside the parentheses, python silently pass the object `df` as an argument into the method. Therefore, all methods inside the class have to expect self as an input argument.

Onto creating an actual object out of the class:

```
buddy1 = Bootcampbuddy("Deryck ", 25, "IT specialist", "cats")
```

This line of code create an object belonging to the class `Bootcampbuddy`, and stores the object in a variable called buddy1. It specifies the attributes of the object through the 4 input arguments.

We can check out the collection of variables, aka attributes stored in buddy1 by using print.
`print(buddy1.name)` would return `Deryck`
`print(buddy1.age)` would return `25`
`print(buddy1.job)` would return `IT Specialist`
`print(buddy1.interest)` would return `cats`
`print(buddy1.skills)` would return `['VBA', 'Python']` because although we didn't specify the skills in the input argument, according to the class template, every object in this class by default has skills `['VBA', 'Python']`

`buddy1` is just one specific instance of the class `Bootcampbuddy`. Using the same blueprint/template, we can create many more objects.

```
buddy2 = Bootcampbuddy ("Satoshi", 35, 'Director', 'baseball')
```

```
buddy3 = Bootcampbuddy ("Julia", 29, 'Unemployed Bum', 'videogames')
```

And so on…

As for calling the methods defined in the class, it's the same concept as using df.head()
For example, call `introduce_self` method for the object `buddy1`

```
buddy1.introduce_self()
```

This would return `Hi, my name is Deryck`

Now call `what_do_you_know`

```
buddy1.what_do_you_know()
```

This would return `Deryck now knows VBA, Python`
(Note the line `skill_string = ', '.join(self.skills)` simply converts the list stored in `self.skills` into a string, it's unrelated to objected oriented programming.)

Next, call `learn_new_things`. This is a little different from the two previous methods. Looking at the method definition in class, in addition to the hidden `self` input argument, it also requires an additional input from the user, so this time, when we call the method, there has to be something inside the parentheses.

```
buddy1.learn_new_things("SQL")
```

Executing this method would then append the string "`SQL`" into the list stored in `buddy1.skills`

We can check what the `skills` attribute now looks like for `buddy1`
`print(buddy1.skills)` would now return `['VBA', 'Python', 'VBA']`

And if we called `what_do_you_know` again, it would return `Deryck now knows VBA, Python, SQL`

In sum, objects are a way to encapsulate a collection of variables (aka attributes) and functions (aka methods). The values stored in the variables, aka attributes, stored in an object cannot be modified normally, except using the methods defined in the class that the object belongs to. It's a way to make sure others would interact with the stored variables only in ways we want them to.

Using ORM to Interact with a Database

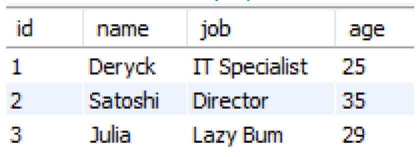
ORM (Object-relational mapping): maps objects into classes, a framework/mental model that converts classes/objects into tables/rows in a database.

- **Class and Database analogy:**
  class = table
  object = row
  attribute = column in a row

- **Using classes to create database table**

```
from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String, Float
# sqlalchemy provides these classes Column, Integer, String, Float to help you create tables, columns,
fields of specific types

from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
# Base is a superclass loaded from sqlalchemy. It will be passed into the class definition for your custom
class to inherit. Anytime when you declare a class,  Base remembers you created this table.

engine = create_engine("<DB_PATH>")

import pymysql
pymysql.install_as_MySQLdb()
# these two lines uses the pymysql module to help sqlalchemy to work with mysql

class Bootcampbuddy(Base):
    __tablename__ = 'buddies'
    id = Column(Integer, primary_key=True)
    name = Column(String(255))
    job = Column(String(255))
    age = Column(Integer)
# defines the structure inside the data table. Integer and String are both special classes from
sqlalchemy, unrelated to regular python command int and str. Id, name, job, and age  are all column
headers in the table.
# this is class will be used as a "container" for the data in the database. It's a template for the actual
data (specific instances).

buddy1 = Bootcampbuddy(name="Deryck", job='IT Specialist', age=25)
# a specific instance of the Bootcampbuddy class. This will be one row of data inserted into the table
buddies
buddy2 = Bootcampbuddy (name="Satoshi", job='Director', age=35)
buddy3 = Bootcampbuddy (name="Julia", job='Lazy Bum', age=29)

Base.metadata.create_all(engine)
# this creates the empty table in the connected database, with column defined in class Bootcampbuddy
```

```
from sqlalchemy.orm import Session
session_obj = Session(bind=engine)
# create an object called session_obj out of a built-in class Session in sqlalchemy. This object represents
the connection between the script and the database.

session_obj.add(buddy1)
session_obj.add(buddy2)
session_obj.add(buddy3)
# queue up the three objects buddy1, buddy2 and buddy3 to be added to the connected database. At
this point the data contained in these objects are not inserted into the database yet

session_obj.commit()
# adds the queued up data to the database

buddylist = session_obj.query(Bootcampbuddy)
# queries the database for the entire table associated with the class Bootcampbuddy (i.e. buddies), store
the returned data in variable buddylist

for person in buddylist:
    print(person.name)
# iterates through each object (a row in data table) and print the name attribute (column)

#final result on mysqlbench
```



id	name	job	age
1	Deryck	IT Specialist	25
2	Satoshi	Director	35
3	Julia	Lazy Bum	29