# Read.me

## Commands

Clone the RealEstateDB.git on your local directory, enter the directory and run the following commands to run and test the query on the database.

*python3.6 -m venv .venv* \
*source .venv/bin/activate* \
*pip3 install -r requirements.txt* \
*python3 create.py* \
*python3 insert_data.py* \
*python3 transaction.py* \
*python3 query.py*

However, there are still some minor bugs when I translate work done in RealEstateDB.ipynb into py files and some of these files do not work even though they work in RealEstateDB.ipynb. **To see the result of the database, please go to RealEstateDB.ipynb file directly.**

*create.py* creates the database schema.
*insert_data.py* inserts mock data to initialize tables.
*transaction.py* creates mock transactions that sold all the houses on the list.
*query.py* tested the required queries and generate reports.

## Database Design

I consider both the principles of database design, what are the needs of different people (e.g. real estate agent and company), and what queries and data needs to be reported together the most when designing this database.

The principles I follow to design the database is the second normalization, which data are separated enough so that every column in a table only depends on its primary keys. However, this principle is not followed blindly in the design because some denormalization is applied in two tables (Office_Sale, Agent_Commission) to so that queries on these two tables do not have to join other tables.

For real estate agents, they need to 1) submit/update new house listing, and 2) to submit/update transaction details. For the company, it wants to be able to analyze data easily on the unit of

agent (e.g. commission per agent), office (e.g. total sale per office), region (e.g. average sale price per zip code), and the company itself (e.g. total sale). Therefore, I designed the architecture of the database as follow:

The tables in the database are organized as follow:

Office
- ID (PK)
- address
- area

Agent
- ID (PK)
- name
- email
- phone

Buyer
- buyerID (PK)
- name
- email
- phone

Seller
- ID (PK)
- name
- email
- phone

HouseListing
- ID (PK)
- sellerID (FK)
- agentID (FK)
- officeID (FK)
- nbedroom
- nbathroom
- saleprice
- zipcode
- listingdate

- sold

SoldRecord
- buyerID (FK)
- agentID (FK)
- HouseListingID (FK)
- saledate

Agent_Office (many-to-many)
- agentID (FK)
- officeID (FK)

Office_Sale (one-one)
- OfficeID (PK) (FK)
- OfficeTotalSale
- month

Agent_Commission (one-one)
- AgentID (PK) (FK)
- Commission
- month

The advantage of this design is a clear separation of concerns. Updating individual information is separated from updating house listing or transaction records. If we want to extend the separation further, we can give different privileges to different types of users. In this way, agents can be restrained to only update HouseListing, SoldRecord, and Buyer tables, while administrators can access and change information in other tables. Table Office_Sale and Agent_Commission are designed to analyze total sale, sale per office, and commission per agent conveniently to generate reports in a designated month as requested. In these two tables, the foreign keys are also primary keys since they uniquely identify rows in the table.

I want to highlight a few details in my implementation of the database:

1) I leveraged a *get_or_create* function to prevent duplicate records in the database. If an instance already exists (e.g. a buyer, a sold record), it will only return the instance rather than creating a new one.
2) For the transaction triggered by a house being sold, I used two tricks to prevent data from being half transported. One is to use try and except syntax so that if the transaction failed in the halfway, the except statement will catch it and roll back the transaction so that changes half-made will be undone. The other is to put the query that updates the listing

status (mark as sold) at the end of the transaction because this query can have some obvious changes in the front end (e.g. a text box changes from available to sold). Therefore, if the transaction failed, the user should have direct observations on the effect.

3) To improve the speed of matching and searching id when joining tables, I index every primary key in all the tables.
4) The assignment requirement asks to update a summary table containing the total sale when a house is sold. I found this requirement confusing because it does not make sense to create a table only to store one cell or one row. It is very easy to just calculate the total sale by summing up the sale per office in the Office_Sale table. This is also part of the reason why I create table Office_Sale. Technically we can still get the sum using Sold_Record table alone, but since we need to constantly generate reports for sale per office and total sale in the company, having an independent table of Office_Sale makes both queries easier.

# Code

# RealEstateSQLAlchemy

April 7, 2019

### 0.0.1 Code References:

- https://docs.sqlalchemy.org/en/latest/orm/
- https://www.pythoncentral.io/sqlalchemy-orm-examples/ (many to many)
- https://www.pythonsheets.com/notes/python-sqlalchemy.html
- http://www.rmunn.com/sqlalchemy-tutorial/tutorial.html
- https://stackoverflow.com/questions/270879/efficiently-updating-database-using-sqlalchemy-orm
- https://gist.github.com/knu2xs/8ca7e0a39bf26f736ef7 (get_random_date function)
- https://stackoverflow.com/questions/44511046/sqlalchemy-prevent-duplicate-rows (get_or_create function)
- https://stackoverflow.com/questions/2128505/whats-the-difference-between-filter-and-filter-by-in-sqlalchemy (difference between filter and filter_by)

### 0.0.2 Import Libraries

In [103]:
```
import sqlalchemy
from sqlalchemy import create_engine, update, Column, Text, Integer, String, Float, I
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker
from sqlalchemy.orm.exc import NoResultFound
from sqlalchemy.sql.expression import extract
from sqlalchemy.sql import func, text
import pandas as pd
from datetime import datetime
import random
import numpy as np
```

In [2]:
```
def get_random_date(month, year):

    # try to get a date
    try:
        return datetime.strptime('{} {} {}'.format(random.randint(1, 30), month, year)

    # if the value happens to be in the leap year range, try again
    except ValueError:
        return get_random_date(year)
```

### 0.0.3 Create Database Schema

In [3]:
```
#create database engine and tables

engine = create_engine('sqlite:///realestate.db')
engine.connect()

Base = declarative_base()

class Office(Base):
    __tablename__ = 'office'
    id = Column(Integer, primary_key = True, index=True)
    name = Column(String)
    address = Column(String)
    area = Column(String)
    agents = relationship('Agent', secondary = 'agent_office') #first class name, seco

    def __repr__(self):
        return "<office(id=%s, name='%s', address='%s', area='%s'>" % (
            self.id, self.name, self.address, self.area)

class Agent(Base):
    __tablename__ = 'agent'
    id = Column(Integer, primary_key = True, index=True)
    name = Column(String)
    email = Column(String)
    phone = Column(Integer)
    offices = relationship('Office', secondary = 'agent_office') #reference to class

    def __repr__(self):
        return "<agent(id=%s, name='%s', email='%s', phone=%s>" % (
            self.id, self.name, self.email, self.phone)

class Buyer(Base):
    __tablename__ = 'buyer'
    id = Column(Integer, primary_key = True, index=True)
    name = Column(String)
    email = Column(String)
    phone = Column(Integer)

    def __repr__(self):
        return "<buyer(id=%s, name='%s', email='%s', phone=%s)>" % (
            self.id, self.name, self.email, self.phone)

class Seller(Base):
    __tablename__ = 'seller'
    id = Column(Integer, primary_key = True, index=True)
    name = Column(String)
```

```
    email = Column(String)
    phone = Column(Integer)

    def __repr__(self):
        return "<seller(id=%s, name='%s', email='%s', phone=%s>" % (
            self.id, self.name, self.email, self.phone)

class HouseListing(Base):
    __tablename__ = 'houseListing'
    id = Column(Integer, primary_key = True, index=True)
    sellerID = Column(Integer, ForeignKey('seller.id'))
    agentID = Column(Integer, ForeignKey('agent.id'))
    officeID = Column(Integer, ForeignKey('office.id'))
    nbedroom = Column(Integer)
    nbathroom = Column(Integer)
    saleprice = Column(Float(2))
    zipcode = Column(Integer)
    listdate = Column(DateTime())
    sold = Column(Boolean)

    def __repr__(self):
        return '''<houseListing(
        id=%s, sellerID=%s, agentID=%s, officeID=%s, nbedroom=%s,
        nbathroom=%s, saleprice=%s, zipcode=%s, listdate='%s', sold='%s'>''' % (
            self.id, self.sellerID, self.buyerID, self.agentID,
            self.officeID, self.houseID, self.listdate)

class SoldRecord(Base):
    __tablename__ = 'soldRecord'
    id = Column(Integer, primary_key = True, index=True)
    buyerID = Column(Integer, ForeignKey('buyer.id'))
    agentID = Column(Integer, ForeignKey('agent.id'))
    houseListingID = Column(Integer, ForeignKey('houseListing.id'))
    saledate = Column(DateTime())

    def __repr__(self):
        return '''<soldRecord(
        id=%s, buyerID=%s, agentID=%s, houseListingID=%s, saledate='%s'>'''% (
            self.id, self.buyerID, self.agentID, self.houseListingID, self.saledate)

class Agent_Office(Base):
    __tablename__ = 'agent_office'
    agentID = Column(Integer, ForeignKey('agent.id'), primary_key = True) #foreign key
    officeID = Column(Integer, ForeignKey('office.id'), primary_key = True)

    def __repr__(self):
        return "<agent_office(agentID=%s, officeID=%s>" % (self.agentID, self.officeID)
```

```
class Office_Sale(Base):
    __tablename__ = 'office_sale'
    officeID = Column(Integer, ForeignKey('office.id'), primary_key = True, index=True)
    officeTotalSale = Column(Float(2))
    month = Column(Integer)

    def __repr__(self):
        return "<office_sale(officeID=%s, officeTotalSale='%s', month='%s'>" % (
            self.officeID, self.officeTotalSale, self.month)

class Agent_Commission(Base):
    __tablename__ = 'agent_commission'
    agentID = Column(Integer, ForeignKey('agent.id'), primary_key = True, index=True)
    commission = Column(Float(2))
    month = Column(Integer)

    def __repr__(self):
        return "<agent_commission(agentID=%s, commission='%s', month='%s'>" % (
            self.officeID, self.officeTotalSale, self.month)

#drop_all line is to start with a new table that recognizes editions made to declarati
#The metadata create_all() method does not replace existing tables.
Base.metadata.drop_all(bind=engine)
Base.metadata.create_all(bind=engine)
Session = sessionmaker(bind=engine)
```

### 0.0.4 Initialized Tables and Insert Mock Data

In [4]:
```
#A transaction to add mock data
session = Session()

#create three offices
office1 = Office(name = "New York Office", address = "14 Broadway Street, NY", area = '
office2 = Office(name = "San Francisco Office", address = "851 California Street, SF",
office3 = Office(name = "Atlanta Office", address = "123 Martin Luther King Street, AT'

#each office has two agents
agent11 = Agent(name = "John", email="john@gmail.com", phone="11111")
agent12 = Agent(name = "Michael", email="michael@gmail.com", phone="11112")

agent21 = Agent(name = "Alice", email="alice@gmail.com", phone="21111")
agent22 = Agent(name = "Mike", email="mike@gmail.com", phone="21112")

agent31 = Agent(name = "Ben", email="ben@gmail.com", phone="31111")
agent32 = Agent(name = "Kelly", email="kelley@gmail.com", phone="31112")


#some agents belong to more than one office
```

```
agent11.offices.append(office2)
agent12.offices.append(office3)

office1.agents.append(agent11)
office1.agents.append(agent12)

office2.agents.append(agent21)
office2.agents.append(agent22)

office3.agents.append(agent31)
office3.agents.append(agent32)

#create one seller
seller1 = Seller(name = "Rockefeller", email="rockefeller@gmail.com", phone="12345")

#add two house listings for each agent
listing1 = HouseListing(sellerID = 1, agentID = 1, officeID=1,
                        nbedroom=2, nbathroom=2, saleprice=80000, zipcode=99999,
                        listdate=get_random_date(11, 2018), sold=False)

listing2 = HouseListing(sellerID = 1, agentID = 1, officeID=2,
                        nbedroom=2, nbathroom=1, saleprice=100000, zipcode=88888,
                        listdate=get_random_date(11, 2018), sold=False)

listing3 = HouseListing(sellerID = 1, agentID = 2, officeID=2,
                        nbedroom=5, nbathroom=1, saleprice=200000, zipcode=88887,
                        listdate=get_random_date(11, 2018), sold=False)

listing4 = HouseListing(sellerID = 1, agentID = 3, officeID=2,
                        nbedroom=1, nbathroom=1, saleprice=250000, zipcode=88887,
                        listdate=get_random_date(11, 2018), sold=False)

listing5 = HouseListing(sellerID = 1, agentID = 3, officeID=2,
                        nbedroom=2, nbathroom=3, saleprice=400000, zipcode=88887,
                        listdate=get_random_date(11, 2018), sold=False)

listing6 = HouseListing(sellerID = 1, agentID = 4, officeID=1,
                        nbedroom=3, nbathroom=2, saleprice=500000, zipcode=99998,
                        listdate=get_random_date(11, 2018), sold=False)

listing7 = HouseListing(sellerID = 1, agentID = 4, officeID=3,
                        nbedroom=3, nbathroom=2, saleprice=1200000, zipcode=77777,
                        listdate=get_random_date(11, 2018), sold=False)

listing8 = HouseListing(sellerID = 1, agentID = 5, officeID=3,
                        nbedroom=1, nbathroom=1, saleprice=600000, zipcode=77776,
                        listdate=get_random_date(11, 2018), sold=False)
```

```
listing9 = HouseListing(sellerID = 1, agentID = 6, officeID=3,
                        nbedroom=1, nbathroom=1, saleprice=600000, zipcode=77776,
                        listdate=get_random_date(11, 2018), sold=False)

#initialize Office_Sale Table
officesale1 = Office_Sale(officeID = 1, officeTotalSale = 0.00, month=12)
officesale2 = Office_Sale(officeID = 2, officeTotalSale = 0.00, month=12)
officesale3 = Office_Sale(officeID = 3, officeTotalSale = 0.00, month=12)

#initialize Agent_Commission Table
agentcommission1 = Agent_Commission(agentID = 1, commission = 0.00, month=12)
agentcommission2 = Agent_Commission(agentID = 2, commission = 0.00, month=12)
agentcommission3 = Agent_Commission(agentID = 3, commission = 0.00, month=12)
agentcommission4 = Agent_Commission(agentID = 4, commission = 0.00, month=12)
agentcommission5 = Agent_Commission(agentID = 5, commission = 0.00, month=12)
agentcommission6 = Agent_Commission(agentID = 6, commission = 0.00, month=12)

session.add_all([office1, office2, office3, seller1, listing1, listing2, listing3, list
                 listing5, listing6, listing7, listing8, listing9, officesale1, officesa
                 officesale3, agentcommission1, agentcommission2, agentcommission3,
                 agentcommission4, agentcommission5, agentcommission6])

session.commit()
```

**Checking if tables created successfully**

```
In [5]: pd.read_sql(session.query(Office).statement, session.bind)
```

Out[5]:

| | id | name | address | area |
|---|---|---|---|---|
| 0 | 1 | New York Office | 14 Broadway Street, NY | New York |
| 1 | 2 | San Francisco Office | 851 California Street, SF | San Francisco |
| 2 | 3 | Atlanta Office | 123 Martin Luther King Street, AT | Atlanta |

```
In [6]: pd.read_sql(session.query(Agent_Office).statement, session.bind).sort_values(by='agentI
```

Out[6]:

| | agentID | officeID |
|---|---|---|
| 0 | 1 | 1 |
| 6 | 1 | 2 |
| 4 | 2 | 2 |
| 5 | 3 | 2 |
| 1 | 4 | 1 |
| 7 | 4 | 3 |
| 2 | 5 | 3 |
| 3 | 6 | 3 |

```
In [7]: pd.read_sql(session.query(Seller).statement, session.bind)
```

Out[7]:

| | id | name | email | phone |
|---|---|---|---|---|
| 0 | 1 | Rockefeller | rockefeller@gmail.com | 12345 |

```
In [8]: pd.read_sql(session.query(HouseListing).statement, session.bind)
```

```
Out[8]:    id  sellerID  agentID  officeID  nbedroom  nbathroom  saleprice  zipcode  \
        0   1         1        1         1         2          2      80000    99999
        1   2         1        1         2         2          1     100000    88888
        2   3         1        2         2         5          1     200000    88887
        3   4         1        3         2         1          1     250000    88887
        4   5         1        3         2         2          3     400000    88887
        5   6         1        4         1         3          2     500000    99998
        6   7         1        4         3         3          2    1200000    77777
        7   8         1        5         3         1          1     600000    77776
        8   9         1        6         3         1          1     600000    77776

            listdate   sold
        0 2018-11-16  False
        1 2018-11-07  False
        2 2018-11-27  False
        3 2018-11-14  False
        4 2018-11-08  False
        5 2018-11-13  False
        6 2018-11-29  False
        7 2018-11-05  False
        8 2018-11-07  False
```

```
In [9]: pd.read_sql(session.query(Office_Sale).statement, session.bind)
```

```
Out[9]:    officeID  officeTotalSale  month
        0         1                0     12
        1         2                0     12
        2         3                0     12
```

```
In [10]: pd.read_sql(session.query(Agent_Commission).statement, session.bind)
```

```
Out[10]:    agentID  commission  month
         0        1           0     12
         1        2           0     12
         2        3           0     12
         3        4           0     12
         4        5           0     12
         5        6           0     12
```

### 0.0.5 Make Transactions

**Testing transaction assuming one house is sold**

```
In [11]: #define a get_or_create function to prevent duplication of records

         def get_or_create(model, **kwargs):
             """
```

```
        Usage:
        class Employee(Base):
            __tablename__ = 'employee'
            id = Column(Integer, primary_key=True)
            name = Column(String, unique=True)

        get_or_create(Employee, name='bob')
        """
        instance = get_instance(model, **kwargs)
        if instance is None:
            instance = create_instance(model, **kwargs)
        return instance

    def create_instance(model, **kwargs):
        """create instance"""
        try:
            instance = model(**kwargs)
            session.add(instance)
            session.flush()
        except Exception as msg:
            session.rollback()
            raise(msg)
        return instance


    def get_instance(model, **kwargs):
        """Return first instance found."""
        try:
            return session.query(model).filter_by(**kwargs).first()
        except NoResultFound:
            return
```

In [12]:
```
session = Session()

    def Transaction(listingid):
        #Assume House Listing #1 is sold. The staff should mark the listing as sold, and

        try:

            #create a record for the buyer if not exist
            buyer = get_or_create(Buyer, name = "Trump", email = "trump@gmail.com", phone

            session.add(buyer)

            #sync with database so buyer.id is available
            session.flush()

            #find corresponding agentID, officeID, and sale price from the House Listing
            agentid = session.query(
```

```
    HouseListing.agentID
).filter(
    HouseListing.id == listingid
).first()[0]

officeid = session.query(
    HouseListing.officeID
).filter(
    HouseListing.id == listingid
).first()[0]

saleprice = session.query(
    HouseListing.saleprice
).filter(
    HouseListing.id == listingid
).first()[0]

#create a sold record if not exist
soldRecord = get_or_create(SoldRecord, buyerID=buyer.id, agentID=agentid,
                           houseListingID=listingid, saledate=get_random_date

month = session.query(
    SoldRecord.saledate
).filter(
    SoldRecord.houseListingID == listingid
).first()[0].month

#update office_sale
curr = session.query(
    Office_Sale.officeTotalSale
).filter(
    Office_Sale.officeID == officeid
).first()[0]

session.query(
    Office_Sale
).filter(
    Office_Sale.officeID == officeid
).update(
    {'officeTotalSale': curr + saleprice, 'month': month}
)

#update agent_commission
if saleprice < 100000:
    commission = saleprice * 0.1
elif 100000 <= saleprice < 200000:
    commission = saleprice * 0.075
elif 200000 <= saleprice < 500000:
```

```
                commission = saleprice * 0.06
            elif 500000 <= saleprice < 1000000:
                commission = saleprice * 0.05
            else:
                commission = saleprice * 0.04

            curr = session.query(
                Agent_Commission.commission
            ).filter(
                Agent_Commission.agentID == agentid
            ).first()[0]

            session.query(
                Agent_Commission
            ).filter(
                Agent_Commission.agentID == agentid
            ).update(
                {'commission': curr + commission, 'month': month}
            )

            #update the status of listing. Put this at the last entry of the transaction
            #because it should have front end correspondence (e.g. change status text)
            #so if the transaction failed, the front end should not respond and thus staf
            session.query(
                HouseListing
            ).filter(
                HouseListing.id == listingid
            ).update(
                {'sold': True}
            )

            # Success, commit everything
            session.commit()

        except:
            # if the transaction failed, roll back to the last status
            session.rollback()
            print("transaction failed. Redo the transaction again")
            raise

    Transaction(1)
```

In [13]:
```
#check if buyer table is created successfully
pd.read_sql(session.query(Buyer).statement, session.bind)
```

Out[13]:

| | id | name | email | phone |
|---|---|---|---|---|
| 0 | 1 | Trump | trump@gmail.com | 54321 |

In [14]:
```
#check if the office total sale is updated.
```

```
In [14]: pd.read_sql(session.query(Office_Sale).filter(Office_Sale.officeID == 1).statement, se
```

```
Out[14]:    officeID  officeTotalSale  month
         0         1            80000     12
```

```
In [15]: #check if the commission is updated
         pd.read_sql(session.query(Agent_Commission).filter(Agent_Commission.agentID == 1).sta
```

```
Out[15]:    agentID  commission  month
         0        1        8000     12
```

```
In [16]: #check if the housing list is marked as sold
         pd.read_sql(session.query(HouseListing).filter(HouseListing.id == 1).statement, sessi
```

```
Out[16]:    id  sellerID  agentID  officeID  nbedroom  nbathroom  saleprice  zipcode  \
         0   1         1        1         1         2          2      80000    99999

             listdate  sold
         0 2018-11-16  True
```

**Make Transactions so that all houses are sold**

```
In [17]: session = Session()
         for i in range(2, 10, 1):
             Transaction(i)
```

```
In [18]: pd.read_sql(session.query(HouseListing).statement, session.bind)
```

```
Out[18]:    id  sellerID  agentID  officeID  nbedroom  nbathroom  saleprice  zipcode  \
         0   1         1        1         1         2          2      80000    99999
         1   2         1        1         2         2          1     100000    88888
         2   3         1        2         2         5          1     200000    88887
         3   4         1        3         2         1          1     250000    88887
         4   5         1        3         2         2          3     400000    88887
         5   6         1        4         1         3          2     500000    99998
         6   7         1        4         3         3          2    1200000    77777
         7   8         1        5         3         1          1     600000    77776
         8   9         1        6         3         1          1     600000    77776

             listdate  sold
         0 2018-11-16  True
         1 2018-11-07  True
         2 2018-11-27  True
         3 2018-11-14  True
         4 2018-11-08  True
         5 2018-11-13  True
         6 2018-11-29  True
         7 2018-11-05  True
         8 2018-11-07  True
```

### 0.0.6 Test Queries. Assume that month refers to 12.

**Find the top 5 offices with the most sales for that month. (I only have three offices in total so all of them will be returned)**

```
In [21]: session = Session()

         query = (session.query(Office_Sale)
                  .filter_by(month=12)
                  .order_by(Office_Sale.officeTotalSale.desc())
                  .limit(5)
                 )

         pd.read_sql(query.statement, session.bind)
```

Out[21]:

| | officeID | officeTotalSale | month |
|---|---|---|---|
| 0 | 3 | 2400000 | 12 |
| 1 | 2 | 950000 | 12 |
| 2 | 1 | 580000 | 12 |

**Find the top 5 estate agents who have sold the most (include their contact details and their sales details so that it is easy contact them and congratulate them).**

```
In [25]: session = Session()

         query = (session.query(Agent_Commission)
                 .join(Agent)
                 .filter(Agent.id==Agent_Commission.agentID, Agent_Commission.month==12)
                 .order_by(Agent_Commission.commission.desc())
                 .limit(5)
                )

         pd.read_sql(query.statement, session.bind)
```

Out[25]:

| | agentID | commission | month |
|---|---|---|---|
| 0 | 4 | 73000 | 12 |
| 1 | 3 | 39000 | 12 |
| 2 | 5 | 30000 | 12 |
| 3 | 6 | 30000 | 12 |
| 4 | 1 | 15500 | 12 |

**Calculate the commission that each estate agent must receive and store the results in a separate table.**

```
In [27]: pd.read_sql(session.query(Agent_Commission)
                     .filter_by(month=12).statement, session.bind)
```

Out[27]:

| | agentID | commission | month |
|---|---|---|---|
| 0 | 1 | 15500 | 12 |

```
1        2        12000      12
2        3        39000      12
3        4        73000      12
4        5        30000      12
5        6        30000      12
```

**For all houses that were sold that month, calculate the average number of days that the house was on the market.** I cannot figure out how to subtract two date columns. I tried using func.datediff, but it keeps giving me syntax error.

```
In [76]: session = Session()

         query = (session.query(HouseListing.listdate, SoldRecord.saledate)
                 .join(SoldRecord)
                 .filter(HouseListing.id==SoldRecord.houseListingID,
                         extract('month', SoldRecord.saledate) == 12)
                )

         # query = (session.query(func.datediff(text('day'), SoldRecord.saledate, HouseListing
         #                        extract('month', SoldRecord.saledate).label('month'))
         #          .join(HouseListing)
         #          .filter(SoldRecord.houseListingID == HouseListing.id,
         #                  extract('month', SoldRecord.saledate) == 12)
         #          )

         # query = (session.query(func.avg((SoldRecord.saledate - HouseListing.listdate).days)
         #                        extract('month', SoldRecord.saledate).label('month'))
         #          .join(HouseListing)
         #          .filter(SoldRecord.houseListingID == HouseListing.id,
         #                  extract('month', SoldRecord.saledate) == 12)
         #          )

         average = []
         for i in query:
             delta = i[1] - i[0]
             average.append(delta.days)

         print("average days for sale", np.mean(average))
         pd.read_sql(query.statement, session.bind)

average days for sale 27.4444444444


Out[76]:     listdate   saledate
         0 2018-11-16 2018-12-09
         1 2018-11-07 2018-12-22
         2 2018-11-27 2018-12-16
         3 2018-11-14 2018-12-08
```

```
4 2018-11-08 2018-12-05
5 2018-11-13 2018-12-09
6 2018-11-29 2018-12-10
7 2018-11-05 2018-12-07
8 2018-11-07 2018-12-17
```

**For all houses that were sold that month, calculate the average selling price**

```
In [46]: session = Session()

         query = (session.query(func.avg(HouseListing.saleprice).label('average'),
                                extract('month', SoldRecord.saledate).label('month'))
                  .join(SoldRecord)
                  .filter(HouseListing.id==SoldRecord.houseListingID,
                          extract('month', SoldRecord.saledate) == 12)
                  )

         pd.read_sql(query.statement, session.bind)
```

Out[46]:

| | average | month |
|---|---|---|
| 0 | 436666.666667 | 12 |

**Find the zip codes with the top 5 average sales prices**

```
In [96]: session = Session()

         query = (session.query(HouseListing.zipcode, func.avg(HouseListing.saleprice).label('a
                  .join(SoldRecord)
                  .group_by(HouseListing.zipcode)
                  .filter(HouseListing.id==SoldRecord.houseListingID)
                  .order_by(func.avg(HouseListing.saleprice).desc())
                  .limit(5)
                  )

         pd.read_sql(query.statement, session.bind)
```

Out[96]:

| | zipcode | average sale price |
|---|---|---|
| 0 | 77777 | 1200000.000000 |
| 1 | 77776 | 600000.000000 |
| 2 | 99998 | 500000.000000 |
| 3 | 88887 | 283333.333333 |
| 4 | 88888 | 100000.000000 |

**Return the sum of total sale in the entire company**

```
In [72]: session = Session()

         query = (session.query(func.sum(Office_Sale.officeTotalSale).label('sum'))
                  )
```

```
pd.read_sql(query.statement, session.bind)
```

Out[72]:

| | sum |
|---|---|
| 0 | 3930000 |