Filtering and targeting data

INTRODUCTION TO DATABASES IN PYTHON

Jason Myers

Co-Author of Essential SQLAlchemy and Software Engineer

Where clauses

```
stmt = select([census])
stmt = stmt.where(census.columns.state == 'California')
results = connection.execute(stmt).fetchall()
for result in results:
    print(result.state, result.age)
```

```
California 0
California 1
California 2
California 3
California 4
California 5
...
```

Where clauses

- Restrict data returned by a query based on Boolean conditions
- Compare a column against a value or another column
- Often use comparisons `==` , `<=` , `>=` , or `!=`

Expressions

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

Expressions

```
stmt = select([census])
stmt = stmt.where(census.columns.state.startswith('New'))
for result in connection.execute(stmt):
    print(result.state, result.pop2000)
```

```
New Jersey 56983
New Jersey 56686
New Jersey 57011
...
```

Conjunctions

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

Conjunctions

```
from sqlalchemy import or_
stmt = select([census])
stmt = stmt.where(
    or_(census.columns.state == 'California',
        census.columns.state == 'New York'
    )
)
for result in connection.execute(stmt):
    print(result.state, result.sex)
```

```
New York M
...
California F
```

Let's practice!

INTRODUCTION TO DATABASES IN PYTHON

Ordering query results

INTRODUCTION TO DATABASES IN PYTHON

Jason Myers

Co-Author of Essential SQLAlchemy and Software Engineer

Order by clauses

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

Order by ascending

```
print(results[:10])
```

```
[('Illinois',), ...]
```

```
stmt = select([census.columns.state])
stmt = stmt.order_by(census.columns.state)
results = connection.execute(stmt).fetchall()
print(results[:10])
```

```
[('Alabama',), ...]
```

Order by descending

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

Order by multiple

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

Order by multiple

```
print(results)
```

```
('Alabama', 'M')
```

```
stmt = select([census.columns.state, census.columns.sex])
stmt = stmt.order_by(census.columns.state, census.columns.sex)
results = connection.execute(stmt).first()
print(results)
```

```
('Alabama', 'F')
('Alabama', 'F')
...
('Alabama', 'M')
```

Let's practice!

INTRODUCTION TO DATABASES IN PYTHON

Counting, summing, and grouping data

INTRODUCTION TO DATABASES IN PYTHON

Jason Myers

Co-Author of Essential SQLAlchemy and Software Engineer

datacamp

SQL functions

- E.g. `COUNT`, `SUM`
- `from sqlalchemy import func`
- More efficient than processing in Python
- Aggregate data

Sum example

```
from sqlalchemy import func
stmt = select([func.sum(census.columns.pop2008)])
results = connection.execute(stmt).scalar()
print(results)
```

```
302876613
```

Group by

- Allows us to group row by common values

Group by

```
stmt = select([census.columns.sex,
  func.sum(census.columns.pop2008)])
stmt = stmt.group_by(census.columns.sex)
results = connection.execute(stmt).fetchall()
print(results)
```

```
[('F', 153959198), ('M', 148917415)]
```

Group by

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

Group by multiple

```
stmt = select([census.columns.sex,
        census.columns.age,
        func.sum(census.columns.pop2008)
    ])
stmt = stmt.group_by(census.columns.sex,
        census.columns.age)
results = connection.execute(stmt).fetchall()
print(results)
```

```
[('F', 0, 2105442), ('F', 1, 2087705), ('F', 2, 2037280),
('F', 3, 2012742), ('F', 4, 2014825), ('F', 5, 1991082),
('F', 6, 1977923), ('F', 7, 2005470), ('F', 8, 1925725), ...
```

Handling ResultSets from functions

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

Using label()

```
print(results[0].keys())
```

```
['sex', u'sum_1']
```

```
stmt = select([census.columns.sex,
        func.sum(census.columns.pop2008).label('pop2008_sum')
    ])
stmt = stmt.group_by(census.columns.sex)
results = connection.execute(stmt).fetchall()
print(results[0].keys())
```

```
['sex', 'pop2008_sum']
```

Let's practice!

INTRODUCTION TO DATABASES IN PYTHON

SQLAlchemy and pandas for visualization

INTRODUCTION TO DATABASES IN PYTHON

Jason Myers

Co-Author of Essential SQLAlchemy and Software Engineer

datacamp

SQLAlchemy and pandas

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

DataFrame example

```
import pandas as pd
df = pd.DataFrame(results)
df.columns = results[0].keys()
print(df)
```

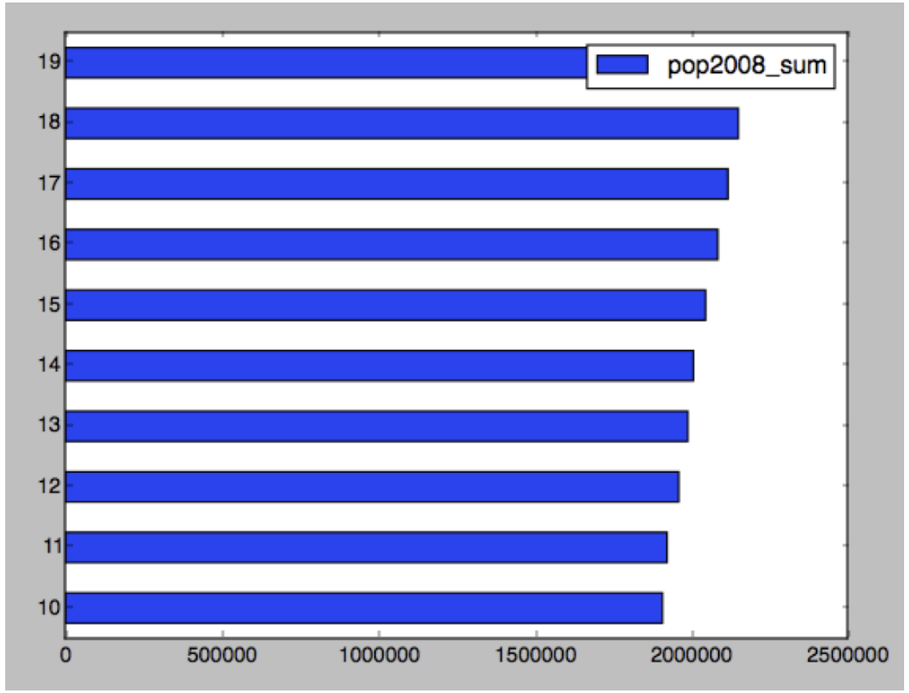
```
    sex  pop2008_sum
0     F      2105442
1     F      2087705
2     F      2037280
3     F      2012742
4     F      2014825
5     F      1991082
```

Graphing

- We can graph just like we would normally

Graphing example

```
import matplotlib.pyplot as plt
df[10:20].plot.barh()
plt.show()
```



Let's practice!

INTRODUCTION TO DATABASES IN PYTHON