INTRODUCTION TO DATABASES IN PYTHON

Calculating Values in a Query

Math Operators

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

Calculating Difference

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

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

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

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

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

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

Case Statement

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

Case Example

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

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

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

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

Cast Statement

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

Percentage Example

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

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

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

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

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

SQL Relationships

Relationships

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

Relationships

Census

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

State_Fact

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

Automatic Joins

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

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

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

Join

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

Select_from

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

Select_from Example

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

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

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

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

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

Joining Tables without Predefined Relationship

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

Select_from Example

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

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

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

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

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

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Working with Hierarchical Tables

Hierarchical Tables

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

Hierarchical Tables - Example

Employees

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

Hierarchical Tables - alias()

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

Querying Hierarchical Data

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

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

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

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

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

Group_by and Func

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

Querying Hierarchical Data

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

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

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

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

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

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!

INTRODUCTION TO DATABASES IN PYTHON

Handling Large ResultSets

Dealing with Large ResultSets

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

Fetching Many Rows

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

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

INTRODUCTION TO DATABASES IN PYTHON

Let’s practice!