Advanced SQLAlchemy

PyCon 2008

March 13th

Advanced SQLAlchemy

- SQL Expressions and Relational Selects
- ORM Query Techniques
- Object Mapping
- Inheritance
- Relations and Collections
- Session / flush() Trickery

Presenters

- Michael Bayer
- Jason Kirtland
- Jonathan Ellis

- Code samples accompany this presentation
- Each section has its own scriptname.py
-  is a reference to a section of the sample code

SQL Expressions

sqlexpression.py

1

Setup

```
from sqlalchemy import *

meta = MetaData(create_engine('sqlite://', echo=True))

users = Table('users', meta,
    Column('id', Integer, primary_key=True),
    Column('name', String(50)))

addresses = Table('addresses', meta,
    Column('id', Integer, primary_key=True),
    Column('email', String(50)),
    Column('user_id', Integer,
           ForeignKey('users.id')))

meta.create_all()
```

Some Data


```
ins = users.insert()

ins.execute([
    {'name':'jack'},
    {'name':'ed'},
    {'name':'wendy'},
    {'name':'mary'}
])

ins = addresses.insert()
ins.execute([
    {'user_id':1, 'email':'jack@jack.com'},
    {'user_id':2, 'email':'ed@yahoo.com'},
    {'user_id':2, 'email':'ed@msn.com'},
    {'user_id':3, 'email':'wendy@nyt.com'},
])
```

Generative Selects


```
sel = select([users.c.id])

sel = sel.select_from(
      users.join(addresses))

sel = sel.where(
    or_(users.c.name=='ed',
      addresses.c.email.like('ed%')))

sel = sel.column(addresses.c.email)

sel = sel.order_by(asc(users.c.name))

for row in sel.execute():
   print row
```

Generative Updates, Deletes


```
# update
upd = users.update()

upd = upd.where(users.c.name=='wendy')

upd = upd.values({'name':'wendy@nypost.com'})

upd.execute()

# delete
del_ = users.delete()

del_ = del_.where(users.c.name == 'wendy')

print del_
```

Exercise Q

```
Write these SQL statements as generative SQL
expressions:

1. SELECT id, name FROM users WHERE users.id > 5

2. UPDATE users SET name='jack smith' WHERE users.id=10

3. SELECT users.*, addresses.* FROM users JOIN addresses
ON users.id=addresses.user_id ORDER BY addresses.email
DESC

4. DELETE FROM addresses WHERE addresses.user_id=(SELECT
id FROM users WHERE name='jack')
```

Answers


```
1. select([users]).\
    where(users.c.id>5)

2. users.update().where(users.c.id==10).\
    values({'name':'jack smith'})

3. select([users, addresses]).\
    select_from(users.join(addresses)).\
    order_by(desc(addresses.c.email))

4. users.delete().where(
    addresses.c.user_id==
         select([users.c.id]).
         where(users.c.name=='jack')
    )
```

Subqueries

- Subqueries come in two flavors
- column clause and WHERE clause subqueries, which return a scalar result, i.e. one column, one row
- FROM clause subqueries, which represent a set of rows, and act like a "virtual" table
- column and WHERE clause subqueries are often "correlated" to the enclosing select

Subquery Examples

```
# column clause subquery
SELECT id, name,
   (select id FROM users where name='jack') as
jacks_id FROM users
```

```
# where clause subquery
SELECT id, name FROM users
  WHERE id=(select id FROM users where
    name='jack')

# from clause subquery
SELECT id, name FROM users JOIN
(select id FROM users where name like '%e%')
  AS ualias ON users.id=ualias.id
```

Scalar Subqueries

- Have one column in the "columns" clause
- Should return exactly one row
- Are declared "scalar" using either as_scalar() or label(); changes expression semantics

```
>>> sel = select([users.c.id]).\
...       where(users.c.name=='jack')

>>> sel = sel.as_scalar()

>>> print sel + 3
(SELECT users.id FROM users
WHERE users.name = :users_name_1) + :param_1
```

Scalar Subqueries


```
# scalar query
sel = select([users.c.id]).\
    where(users.c.name=='jack').as_scalar()

# column clause subqueries
select([users, sel])

select([sel, sel + 5, sel + 7],
    bind=meta.bind)

# WHERE clause subqueries
select([users]).where(users.c.id==sel)

select([users]).where(users.c.id==sel + 1)
```

Row-Based Subqueries

- Return one or more columns
- Return zero or more rows
- are part of the FROM clause
- will want to be an alias() 99% of the time

```
sel = select([users]).\
    where(users.c.name.like('%e%')).alias()

select([users]).select_from(
        users.join(sel, users.c.id==sel.c.id)
    )
```

Correlated Subqueries

- reference a table in an enclosing SELECT statement (usually in the WHERE clause)
- Correlate is indicated in SQL by the table not being referenced in the inner FROM clause
- SQLAlchemy select() object does this automatically
- ...unless you control it using select.correlate (<sometable>)

Correlated Subquery Examples

```
# column clause correlated subquery
SELECT id, name,
  (select count(*) FROM addresses WHERE
user_id=users.id)
FROM users
```

```
# where clause correlated subquery
SELECT id, name FROM users
WHERE name=(select min(email_address) FROM addresses
WHERE user_id=users.id)
```

Implicit Correlation


```
# create subquery - selects the 'min'
# email address for each user
adr = select([func.min(addresses.c.email)]).\
    where(addresses.c.user_id==users.c.id).\
    as_scalar()

# use it as a column-level subquery
sel = select([users, adr])

for row in sel.execute():
    print row
```

Explicit Correlation


```
# same subquery, except we say
# "correlate(users)" so that only "users"
# correlates
adr = select([func.min(addresses.c.email)]).\
    where(addresses.c.user_id==users.c.id).\
    correlate(users).as_scalar()

# select rows from users, each with their
# minimum email address string
sel = select([users, addresses]).\
    select_from(users.join(addresses)).\
    where(addresses.c.email==adr)

for row in sel.execute():
    print row
```

Subquery Exercises

1. Write this SQL as a SQLAlchemy expression.

```
SELECT id, name FROM users
WHERE id=(select user_id FROM addresses WHERE
email='foo@bar.com')
```

2. Given this statement:

```
SELECT id, user_id, email FROM addresses WHERE email IN
('jack@jack.com', 'ed@msn.com')
```

Convert the above to a SQLAlchemy expression, then embed it within an expression that joins the "users" table to the query in the FROM clause, and returns user's name and email address.

Answers


```
# 1.
select([users]).where(users.c.id==
    select([addresses.c.user_id]).
    where(addresses.c.email=='foo@bar.com').\
    as_scalar()
  )

# 2.
sel = select([addresses]).\
   where(addresses.c.email.in_(
     ['jack@jack.com', 'ed@msn.com']
   )).alias()

select([users.c.name,sel.c.email]).\
    select_from(
     users.join(sel, users.c.id==sel.c.user_id))
```

EXISTS Clause

- EXISTS is a SQL keyword which returns "true" if one or more rows is returned from a SELECT statement
- Is ideal for column/where subqueries, since it returns a scalar result in all cases
- Can replace the usage of "IN" against subquery expressions, runs with better efficiency
- Used by SA to check "does object A reference any rows of B with some condition"

EXISTS Examples

```
# return user rows which are referenced by a
# "foo@bar.com" address row
SELECT id, name FROM users WHERE EXISTS
(SELECT 1 FROM addresses WHERE user_id=users.id AND
email='foo@bar.com')


# return user rows which are not referenced by
# any address rows
SELECT id, name FROM users WHERE NOT EXISTS
(SELECT 1 FROM addresses WHERE user_id=users.id)
```

exists() Expressions

```
select([users]).where(
    exists([1]).
    where(addresses.c.user_id==users.c.id).
    where(addresses.c.email=='foo@bar.com')
)

select([users]).where(
    ~exists([1]).
    where(addresses.c.user_id==users.c.id)
  )
```

Exercise Q

1. write a SQL expression using EXISTS to locate all user rows which are referenced by an address row that contains the email address of

'jack@yahoo.com' *or* 'jack@msn.com'

2. write a SQL expression using EXISTS to locate all user rows which are referenced by an address row with email address 'jack@yahoo.com' *and* by an address row with email address 'jack@msn.com'

3. rewrite #1 and #2 as SQLAlchemy expressions

4. write SQL to find all users with 'jack@yahoo.com' but do *not* have 'jack@msn.com'.

Answers


```
# 1.
SELECT id, name FROM users WHERE EXISTS (SELECT 1 FROM addresses
  WHERE user_id=users.id AND email IN ('jack@yahoo.com', 'jack@msn.com'))

# 2.
SELECT id, name FROM users WHERE EXISTS (SELECT 1 FROM addresses
  WHERE user_id=users.id AND email='jack@yahoo.com')
  AND EXISTS (SELECT 1 FROM addresses WHERE user_id=users.id AND
  email='jack@msn.com')

# 3.
select([users]).where(
    exists([1]).where(addresses.c.user_id==users.c.id).\
    where(or_(addresses.c.email.in_(['jack@yahoo.com', 'jack@msn.com'])))
    )

select([users]).where(
      exists([1]).where(addresses.c.user_id==users.c.id).\
      where(addresses.c.user_id=='jack@yahoo.com')
    ).where(
      exists([1]).where(addresses.c.user_id==users.c.id).\
      where(addresses.c.user_id=='jack@msn.com'))
```

?

ORM Query Techniques

ormquery.py

1

Model Setup

```
from sqlalchemy.orm import *

Session = scoped_session(sessionmaker(
  transactional=True,autoflush=False))

class Base(object):
    def __init__(self, **kwargs):
        for key, value in kwargs.iteritems():
            setattr(self, key, value)
    def __repr__(self):
        cls = self.__class__
        return "%s(%s)" % (
            cls.__name__,
            ", ".join(["%s=%r" % (c, getattr(self, c))
                       for c in cls.c.keys()]))

    query = Session.query_property()
```

ORM Configuration

2,3

```
class User(Base):
    pass

class Address(Base):
    pass

mapper(User, users, properties={
   'addresses':relation(Address, backref='user',
    cascade='all, delete-orphan')
})

mapper(Address, addresses)

print User.query.all()
```

Exercise Q

```
mapper(User, users, properties={
    'addresses':relation(Address, backref='user',
     cascade='all, delete-orphan')})

mapper(Address, addresses)
```

1. what is the "primaryjoin" condition for the "addresses" relation ?
2. what is the "foreign_keys" collection for the "addresses" relation ?
3. what determines if "addresses" is many to one, or one to many ?
4. same for the "user" backref ?

Answers

1. users.c.id==addresses.c.user_id
2. [addresses.c.user_id]
3. "addresses" is one-to-many because the foreign key column of the join condition (addresses.user_id) is on the right, or "remote" side of the relation.
4. "users" is many-to-one because the foreign key column (again, addresses.user_id) is on the left, or "local" side of the relation.

Using "any()" and "has()"

4-7

```
# parents where collection member meets criterion
User.query.filter(
  User.addresses.any(Address.email.like('%ed%')))

# parents where no collection member meets criterion
User.query.filter(
  ~User.addresses.any(Address.email.like('%ed%')))

# parents with empty collection
User.query.filter(~User.addresses.any())

# parents where referenced member meets criterion
Address.query.filter(Address.user.has(User.name=='ed'))

# parents where referenced member does not meet
# criterion
Address.query.filter(~Address.user.has(User.name=='ed'))
```

Using Instances in Criterion


```
# get a user, and an address
u1 = User.query.get(2)
a1 = Address.query.get(1)

# users who contain address 'a1'
User.query.filter(User.addresses.contains(a1))

# addresses who reference user 'u1'
Address.query.filter(Address.user == u1)

# addresses with "parent" object 'u1'
Address.query.with_parent(u1).all()

# more specific usage of with_parent
Address.query.with_parent(u1, "addresses").all()
```

Exercise Q

1. Do these all return the same results in all cases ?

```
User.query.join('addresses').\
    filter(Address.email.like('%y%'))
```

```
User.query.filter(
   User.addresses.any(Address.email.like('%y%')))

User.query.filter(
   exists([Address.id]).
   where(Address.user_id==User.id).
   where(Address.email.like('%y%')))

User.query.filter(User.id==select([Address.user_id]).\
    where(Address.email.like('%y%')))
```

Answer

1. The first three statements return the same result, the fourth may or may not. The subquery does not necessarily return a scalar result, which will either fail to return the right results or will raise an error, depending on the type of DB in use.

Self Referential Table


```
nodes = Table('nodes', meta,
    Column('id', Integer, primary_key=True),
    Column('data', String(50)),
    Column('parent_id', ForeignKey('nodes.id')))

nodes.create()
```

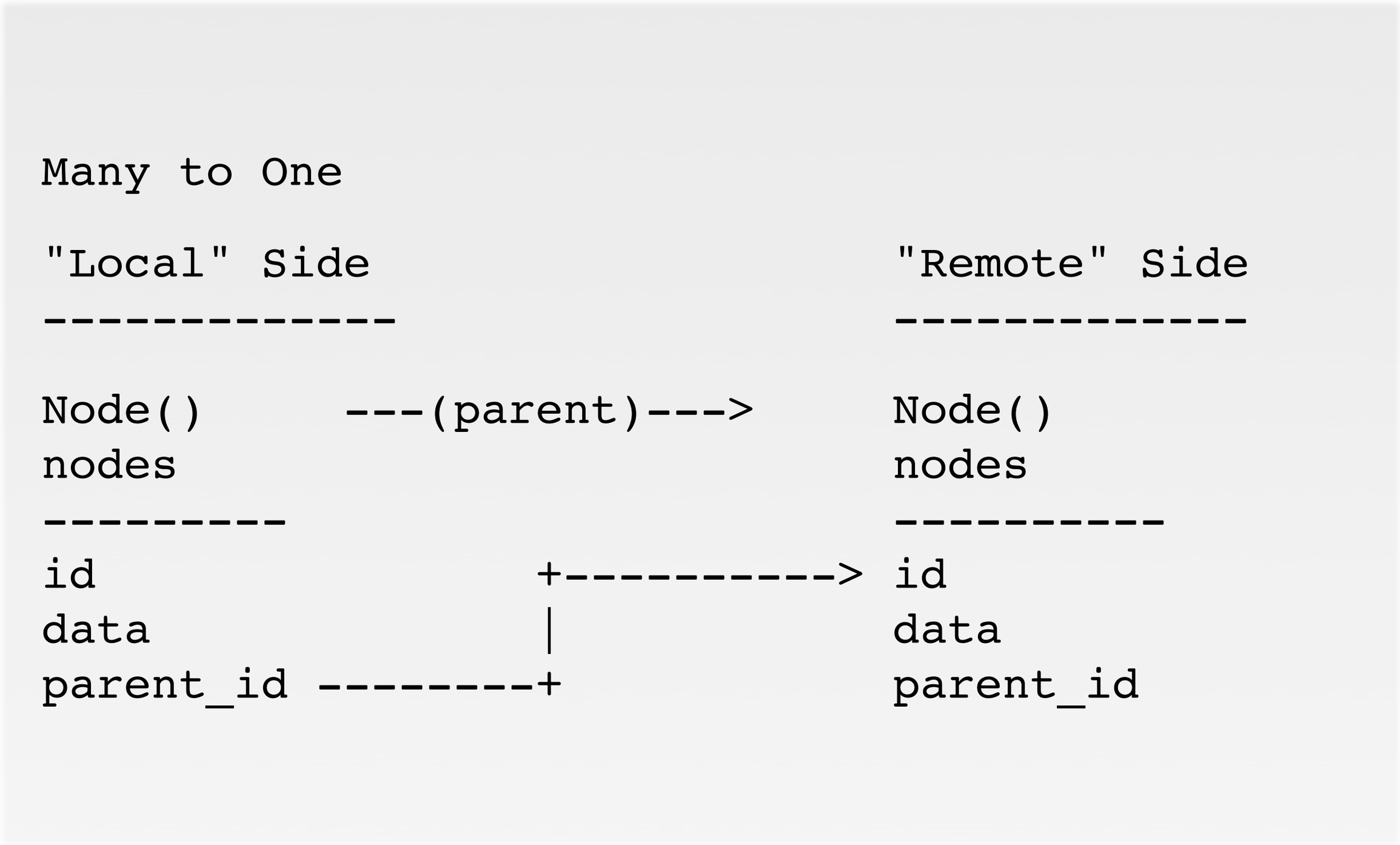
Defining "Remote Side"

```
One to Many

"Local" Side                          "Remote" Side
------------                          -------------

Node()      ---(children)--->         Node()
nodes                                 nodes
---------                             ----------
id     <------------+                 id
data                |                 data
parent_id           +----------       parent_id
```

Defining "Remote Side"



Self-Referential


```
class Node(Base):
    pass

# the "one to many" remote_side is the
# default, so we only need remote_side
# on the "many to one" side
mapper(Node, nodes, properties={
    'children':relation(Node,
     backref=backref('parent', remote_side=nodes.c.id))
})
```

Self-Referential


```
Session.clear()

n1 = Node(data='n1', children=[
    Node(data='n2'),
    Node(data='n3', children=[
        Node(data='n5'),
        Node(data='n6'),
        Node(data='n7'),
    ]),
    Node(data='n4', children=[
        Node(data='n8'),
        Node(data='n9'),
    ]),
  ])

Session.save(n1)
Session.flush()
```

Joining Self-Referentially

```
# SQL to select the node with the path n1/n4/n8

SELECT nodes.*
FROM
  nodes JOIN nodes AS parents ON
  nodes.parent_id=parents.id JOIN nodes AS
  grandparents ON
  parents.parent_id=grandparents.id
WHERE
   grandparents.data='n1' AND
   parents.data='n4' AND
   nodes.data='n8
```

Joining Self-Referentially


```
# SQL Expression

parents = nodes.alias('parents')
grandparents = nodes.alias('grandparents')

nodes.select().\
    select_from(
      nodes.join(parents,
nodes.c.parent_id==parents.c.id).\
      join(grandparents,
parents.c.parent_id==grandparents.c.id)
    ).\
    where(
      and_(
        grandparents.c.data=='n1',
        parents.c.data=='n4', nodes.c.data=='n8'
      )
    )
```

Joining Self-Referentially


```
# ORM Query (using join() to create aliases)

n8 = Node.query.filter(Node.data=='n8').\
    join('parent', aliased=True).\
    filter(Node.data=='n4').\
    join('parent', aliased=True, from_joinpoint=True).\
    filter(Node.data=='n1').one()
```

Practice Q

1. write an ORM query to select all leaf nodes (hint: any() need not take any arguments)
2. write an ORM query to select the children of both 'n3' and 'n4'. 'n3' and 'n4' can be matched by their name alone

Answers

21,22

1. Node.query.filter(~Node.children.any())

2. Node.query.filter(
       Node.parent.has(data.in_(['n3','n4'])))

Adding Columns


```
# SQL to get users and how many addresses for
# each, using correlated subquery
SELECT users.*, (SELECT count(id) FROM addresses WHERE
addresses.user_id=users.id) FROM users
```

```
# ORM version - returns tuples
q = User.query.\
    add_column(
      select([func.count(Address.id)]).\
      where(User.id==Address.user_id).as_scalar()
    )

for user, count in q:
    print user, count
```

Join to an Aggregate Subquery

```
# select all users and the count of their
# addresses (for users who have addresses)

SELECT users.*, anon.count FROM
users LEFT OUTER JOIN
(select user_id, count(id) AS count FROM addresses GROUP
BY addresses.user_id)
AS anon ON users.id=anon.user_id
```

ORM Aggregate Subquery


```
sel = select([
       Address.user_id,
       func.count(Address.id).label('count')
      ]).group_by(Address.user_id).alias()

# ORM version #1, use select_from()
User.query.select_from(
    users.outerjoin(sel,User.id==sel.c.user_id)
  ).add_column(sel.c.count)

# ORM version #2, use outerjoin()
User.query.outerjoin(('addresses', sel)).\
    add_column(sel.c.count)
```

Native SQL Queries


```
# pure SQL.  disambiguate common column names
# using <tablename>_<columnname>

sql = """
    select
      users.id as users_id,
      users.name,
      addresses.id as addresses_id,
      addresses.user_id,
      addresses.email
    from users
    left outer join addresses on
      users.id=addresses.user_id
    order by users.name
"""
```

Native SQL Queries

26-28

```
# load from SQL statement
User.query.from_statement(sql).\
      add_entity(Address)

# load from result set
User.query.add_entity(Address).instances(
   meta.bind.execute(sql))

# direct "address" columns into an eager load
User.query.options(
   contains_eager('addresses')).\
   from_statement(sql)
```

Exercise Q

```
Given this SQL:
SELECT
    users.id AS users_id, users.name AS users_name,
    addresses.id AS addresses_id,
    addresses.email AS addresses_email,
    addresses.user_id AS addresses_user_id,
    adrc.id AS adrc_id, adrc.email AS adrc_email,
    adrc.user_id AS adrc_user_id
FROM
  users JOIN addresses ON users.id=addresses.user_id
LEFT OUTER JOIN addresses AS adrc ON users.id=adrc.user_id
WHERE addresses.email like '%y%'
```

- Write an ORM query to return tuples of User and Address objects using the query above; the query should also attach Address objects from "adr_collection" to the appropriate User object's "addresses" collection (hint: contains_eager() has an "alias" keyword argument).

Answer


```
1. sql = """SELECT
        users.id AS users_id,
        users.name AS users_name,
        addresses.id AS addresses_id,
        addresses.email AS addresses_email,
        addresses.user_id AS addresses_user_id,
        adrc.id AS adrc_id,
        adrc.email AS adrc_email,
        adrc.user_id AS adrc_user_id
    FROM users JOIN addresses ON
    users.id=addresses.user_id
    LEFT OUTER JOIN addresses AS adrc ON
    users.id=adrc.user_id
    WHERE addresses.email like '%y%'"""

User.query.options(
contains_eager('addresses',alias='adrc')).\
     add_entity(Address).from_statement(sql)
```

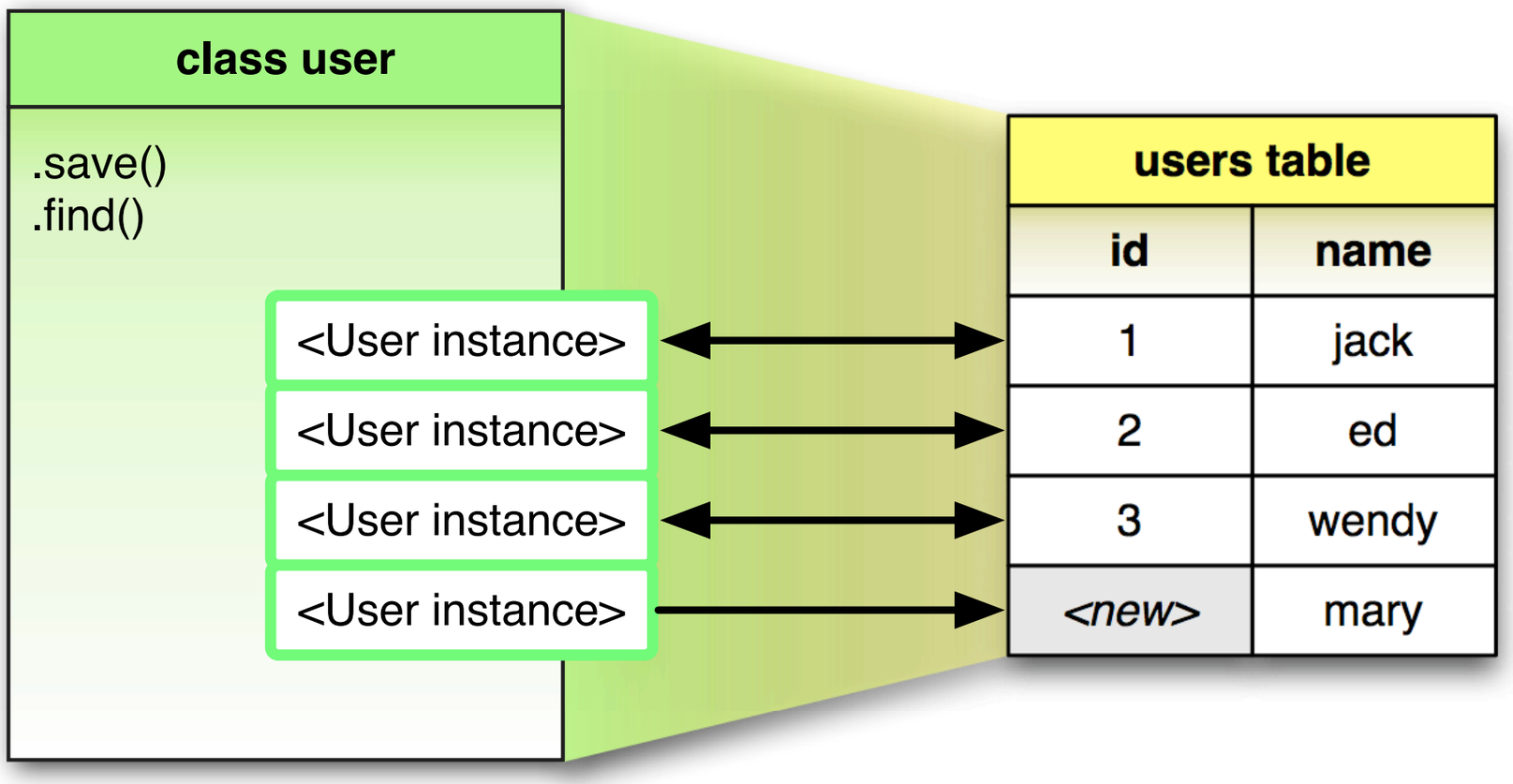
?

Data Mapper Revisited

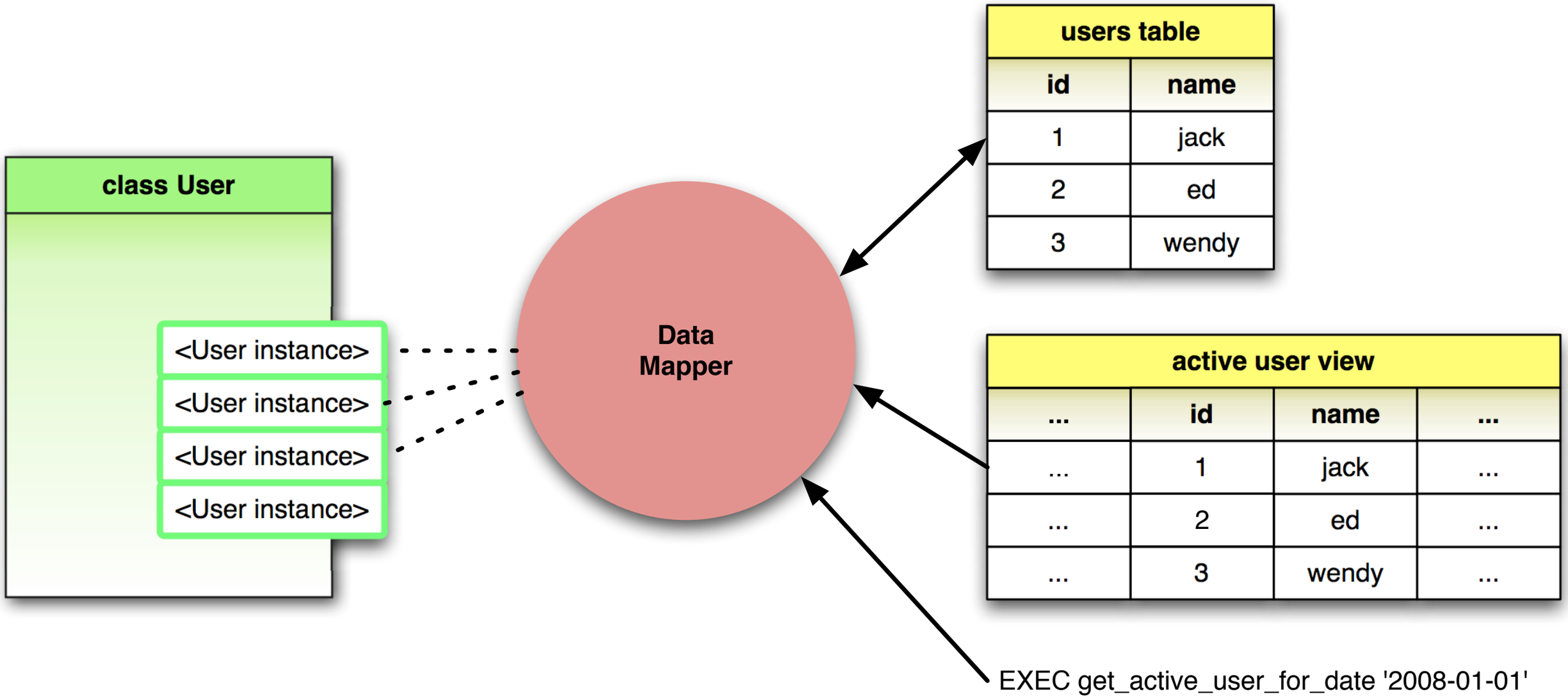
We've talked about SQL, let's talk about Python

- How Data Mapper impacts class design
- Do's and don'ts for mapped classes
- What SQLAlchemy does to your instances
- What SQLAlchemy does to your classes

“Active Record”



“Data Mapper”


class User
<User instance>
<User instance>
<User instance>
<User instance>
Data Mapper
users table
id
name
1
jack
2
ed
3
wendy
active user view
...
id
name
...
...
1
jack
...
...
2
ed
...
...
3
wendy
...
EXEC get_active_user_for_date '2008-01-01'

```
class User(object):
    pass
```

Requirements For a Mapped Class

- Inherit from `object`
- Your class will be monkeypatched

```
>>> class User(object):
...     pass
>>> list(vars(User))
['__dict__', '__module__', '__doc__']
```

```
>>> mapper(User, users)
>>> compile_mappers()
>>> list(vars(User))
['__dict__', '__module__', '__doc__',
 'id', 'name', 'c', '_class_state',
 '__init__']
```

```
'__init__': <function __init__>
```

- Nothing too exciting
- Decorates the class's existing __init__
- Triggers mapper compilation
- Hook for MapperExtensions

- A class can't be mapped twice*
- A class should always be mapped before use

* not counting secondary and entity name mappers

```
>>> u = User()
>>> u.name is None
True
```

```
>>> u.name is None
True
>>> User.name is None
False
>>> type(User.name)
<class 'InstrumentedAttribute'>
```

```
>>> class User(object):
...     pass
```

```
def __init__(self, name):
    self.name = name
```

Attribute Values

Aren't Always in __dict__

```
>>> u = User()
>>> 'name' in u.__dict__
False
>>> u.name
>>> 'name' in u.__dict__
True
```

Moral

- Don't bypass instrumentation
- Use `setattr()` instead of modifying `instance.__dict__`

Question

```
u = User()
```

- When will u.id be set?

Answer

```
>>> session = create_session()
>>> session.save(u)
>>> session.flush()
>>> u.id
1
```

How Instances Are Changed


```
>>> u = User()
>>> vars(u)
{'_state': <InstanceState object>}
```

How Instances Are Changed


```
>>> session.save(u)
>>> vars(u)
{'_state': <InstanceState>,
 '_entity_name': None,
 '_sa_session_id': 20251120}
```

How Instances Are Changed

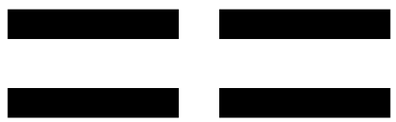

```
>>> session.flush()
>>> vars(u)
{'_state': <InstanceState>,
 '_entity_name': None,
 '_sa_session_id': 20251120,
 '_instance_key':
   (<class 'slides.User'>, (2,), None)
 'id': 2,
 'name': None}
```

?

That's it?

pickles

`__getstate__` / `__setstate__`



__eq__

__eq__

```
class Address(object):
    def __eq__(self, other):
        return self.email == other.email

session.query(User).filter(
  User.addresses.contains(this_address))
```

Backrefs

```
>>> u = User()
>>> a = Address()
>>> a.user = u
>>> u.addresses
[<Address object>]
```

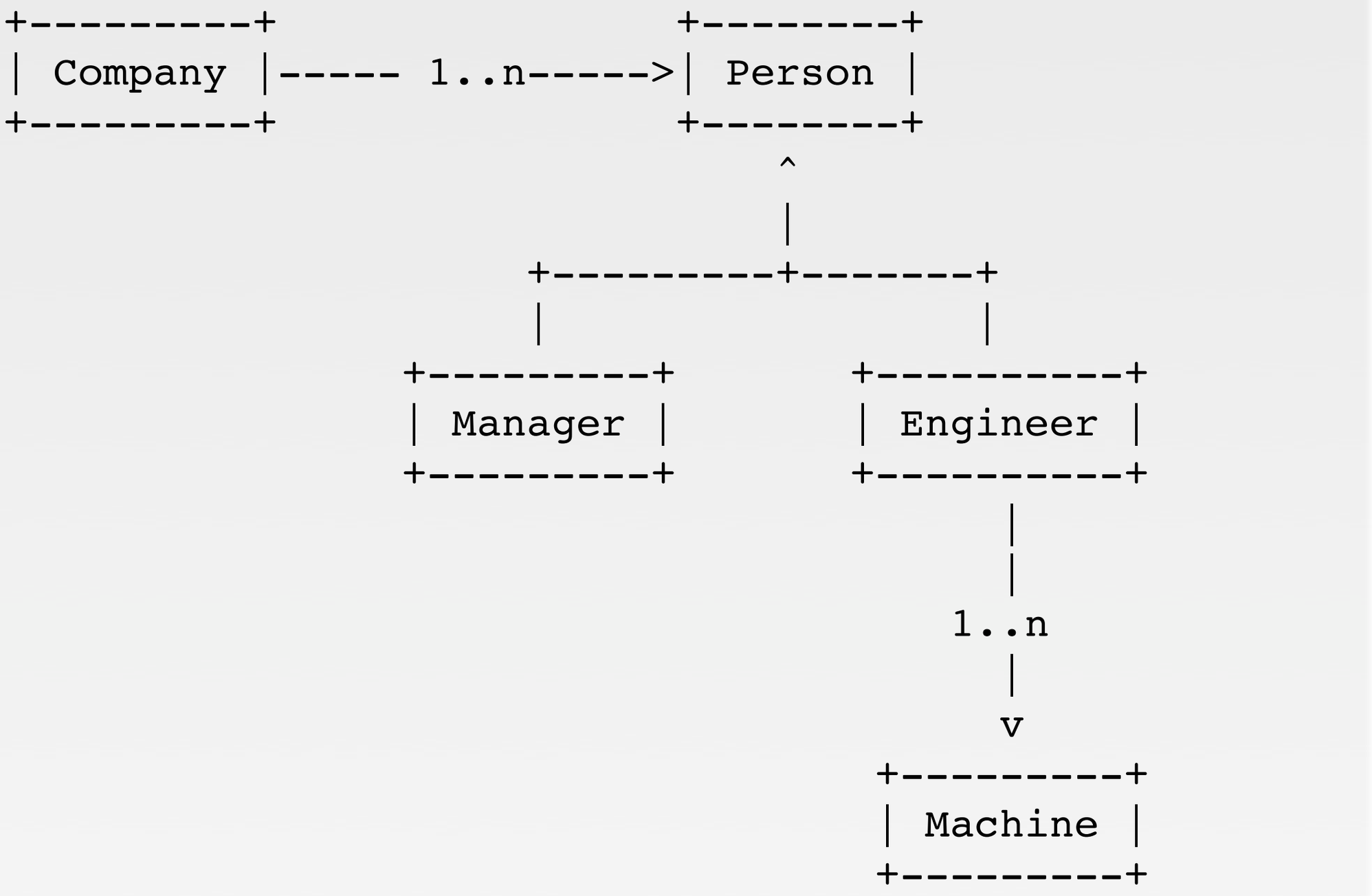
?

Inheritance

Inheritance Patterns

- Three styles: single table, joined table, and concrete
- Joined table is probably the most common
- SQLAlchemy has three ways of loading joined-table data: "union", "select" and "deferred"
- "union" is less important these days as new Query functionalities allow finer-grained control

Information Layout



inheritance.py

1

Table Layout

```
people = Table('people', meta,
   Column('person_id', Integer, primary_key=True),
   Column('company_id', Integer,
     ForeignKey('companies.company_id')),
   Column('name', String(50)),
   Column('type', String(30)))

engineers = Table('engineers', meta,
   Column('person_id', Integer,
     ForeignKey('people.person_id'), primary_key=True),
   Column('primary_language', String(50)))

managers = Table('managers', meta,
  Column('person_id', Integer,
    ForeignKey('people.person_id'), primary_key=True),
  Column('golf_swing', String(30)))
```

Table Layout


```
companies = Table('companies', metadata,
    Column('company_id', Integer, primary_key=True),
    Column('name', String(50)))

machines = Table('machines', metadata,
  Column('machine_id', Integer, primary_key=True),
  Column('name', String(50)),
  Column('engineer_id', Integer,
    ForeignKey('engineers.person_id')))
```

Classes


```
class Company(Base):
    pass

class Person(Base):
    pass

class Engineer(Person):
    pass

class Manager(Person):
    pass

class Machine(Base):
    pass
```

Mappers


```
mapper(Company, companies, properties={
    'employees':relation(Person)})

mapper(Person, people, polymorphic_on=people.c.type)

mapper(Engineer, engineers, inherits=Person,
       polymorphic_identity='engineer',
       properties={'machines':relation(Machine)})

mapper(Manager, managers, inherits=Person,
    polymorphic_identity='manager')

mapper(Machine, machines)
```

Data


```
c1 = Company(name="MegaCorp, Inc.", employees=[
    Engineer(name="dilbert", primary_language="java",
            machines=[Machine(name='IBM ThinkPad'),
                Machine(name='IPhone'),]),
    Engineer(name="wally", engineer_name="wally",
            primary_language="c++",
            machines=[Machine(name="Commodore 64")]),
    Manager(name="dogbert", manager_name="dogbert",
       golf_swing="fore!")])

c2 = Company(name="Elbonia, Inc.", employees=[
    Engineer(name="vlad", primary_language="cobol",
            machines=[Machine(name="Commodore 64"),
                Machine(name="IBM 3270")])])
Session.save(c1)
Session.save(c2)
Session.flush()
```

Anatomy of a Polymorphic Load
polymorphic_fetch="select"

```
print Person.query.filter(Person.name=='dilbert').all()
-----------------------------------------------------

Query.compile() creates SQL statement selecting from "people" table

Query.instances() executes

Query.instances() receives row: (1, 1, u'dilbert', u'engineer')
  Person mapper receives row
    Engineer mapper receives row
    Engineer mapper constructs Engineer instance
  Person mapper returns

Query.instances() appends Engineer to result list

Query.instances() tells Engineer mapper to "post-fetch"

  Engineer mapper selects from "engineers" table
  Engineer mapper populates Engineer with "engineers" columns

Query.instances() returns results
```

Anatomy of a Polymorphic Load

with_polymorphic('*')

```
Person.query.with_polymorphic('*').\
  filter(Person.name=='dilbert').all()
-------------------------------------------------------
Query.compile() creates SQL statement selecting from "people LEFT
OUTER JOIN engineers LEFT OUTER JOIN managers"

Query.instances() executes

Query.instances() receives row:
   (1, 1, None, 1, None, u'regular engineer',u'dilbert', u'java',
None, u'dilbert',u'engineer')

 Person mapper receives row
  Engineer mapper receives row
  Engineer mapper constructs:

   Engineer(name=u'dilbert', type=u'engineer',status=u'regular
engineer',  engineer_name=u'dilbert', primary_language=u'java')

 Person mapper returns: Engineer
Query.instances() appends Engineer to result list
```

Polymorphic Querying


```
# issuing a query against Person returns Engineer
# and Manager objects (and Person, if those exist)
Person.query.all()

# To get just Engineers (and subclasses of Engineer),
# query from Engineer
Engineer.query.all()

# Filter criterion can be against the classes
# explicitly present:
Person.query.filter(Person.name=='dilbert').all()

Engineer.query.filter(Person.name=='dilbert').\
    filter(Engineer.primary_language=='java').all()

Person.query.with_polymorphic(Engineer).\
    filter(Engineer.primary_language='java').all()
```

Polymorphic Relations

13-14

```
# suppose we want to load Company objects,
# which contain a certain person.  The any()
# operator creates an EXISTS clause:
companies = Company.query.filter(
   Company.employees.any(Person.name=='dilbert')
 ).all()

# what if we want to filter on a subclass criterion ?
# use of_type().  the EXISTS will join against the
# subclass' table as well.
companies = Company.query.filter(
   Company.employees.of_type(Manager).\
     any(Manager.golf_swing='fore!')
 ).all()
```

Polymorphic Joins

15-17

```
# similar rules apply when we deal with joins.
# if we want to join Companies to Employees,
# we can use join():
Company.query.join('employees').\
    filter(Person.name=='dilbert').all()

# to join to Engineers via the "employees"
# relation, again the easiest way is of_type():
Company.query.\
   join(Company.employees.of_type(Engineer)).\
   filter(Engineer.primary_language='c++').all()

# same idea works when joining further:
Company.query.\
   join(
     [Company.employees.of_type(Engineer), 'machines']
   ).filter(Machine.name=='Commodore 64').all()
```

Exercise Q

1. Write an ORM query which returns all engineers who own a Machine with the name "IBM 3270"
2. Write an ORM query which returns the Company for each engineer that owns a machine with the name "IBM 3270"
3. Write an ORM query which returns companies that do not have any managers.
4. Write an ORM query which returns Engineers who use "c++" as their primary language, as well as all managers (hint: the Manager part of the criterion is People.type=='manager').

Answers


```
1. Engineer.query.filter(
     Engineer.machines.any(Machine.name=='IBM 3270')
   ).all()

2. Company.query.join(
     Company.employees.of_type(Engineer)
   ).filter(
     Engineer.machines.any(Machine.name=='IBM 3270')
   ).all()

3. Company.query.filter(
     ~Company.employees.of_type(Manager).any()
   ).all()

4. Person.query.with_polymorphic('*').filter(
     or_(Engineer.primary_language=='c++',
         Manager.type=='manager')
   ).all()
```

?

Relations and Collections

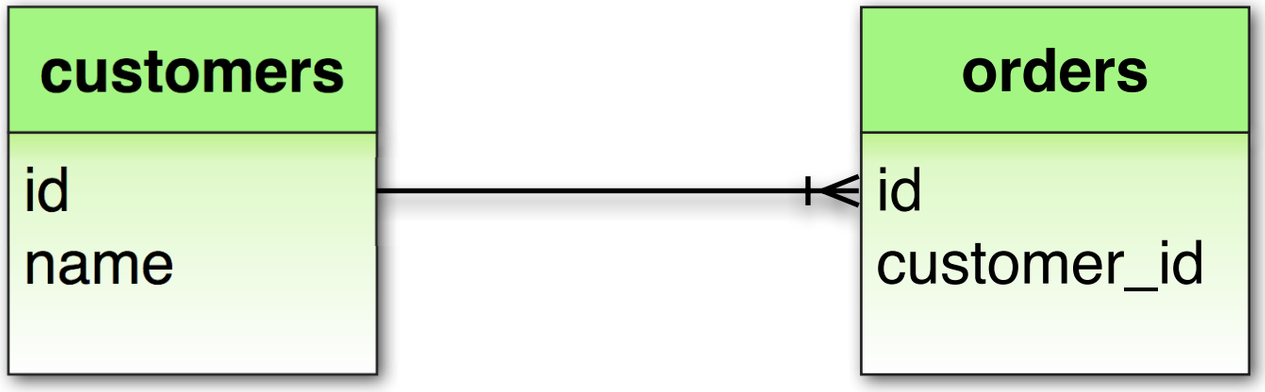
Relations and Collections

- Manipulating Relations
- Beyond Lists
- Associations and Properties of Relations
- Behavior of Relations

relations.py

1,2

One to Many



One to Many

```
>>> c = Customer(name='custie')
>>> o = Order()

>>> c.orders.append(o)
>>> c.orders
[<Order>]

>>> session.save(c)
>>> o in session
True
```

```
>>> class MyCollection(list):
...     pass
>>> mapper(Customer, customers,
      properties={
        'orders': relation(Order,
          backref='customer',
          collection_class=MyCollection)})

>>> c = Customer(name='custie')
>>> type(c.orders)

MyCollection
```

```
>>> o = Order()
>>> c.orders.append(o)
>>> session.save(c)
>>> o in session
True
```

```
>>> session.flush()
>>> c.orders.pop()

>>> session.flush()
```

```
class MyCollection(list):
    pass

class InstrumentedList(list):
    pass
```

Exercise

```
mapper(Customer, customers,
       properties={
          'orders': relation(Order,
            backref='customer',
            lazy=False,
            collection_class=set)})
```

- Load a Customer from the session and add two new Orders to the .orders collection
- Load a Customer from the session and remove an Order from the .orders collection

Answers

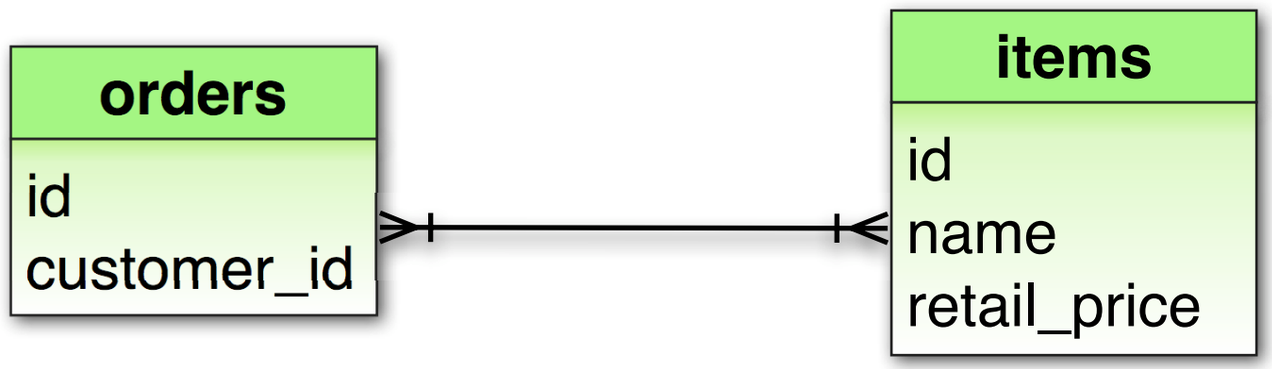
```
>>> c1.orders.add(Item())
>>> c1.orders.add(Item())

>>> c2.orders.pop()
```

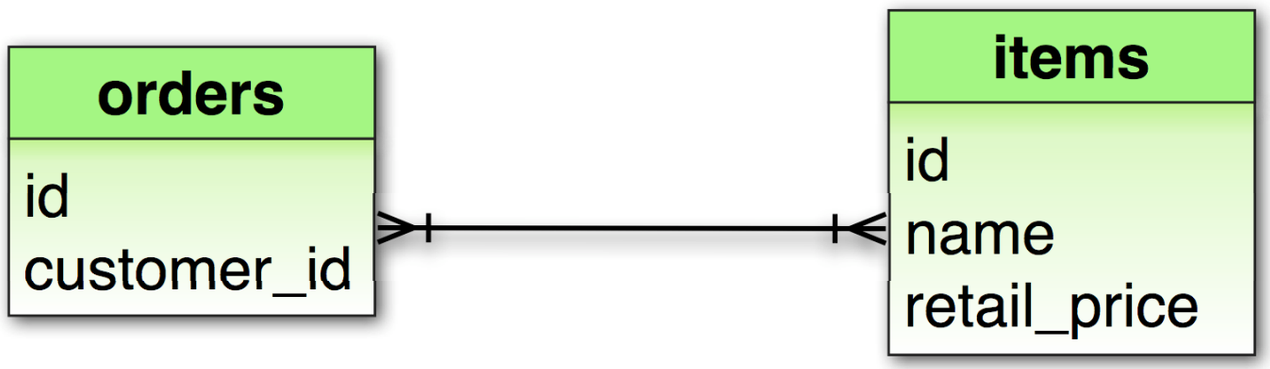
mappings

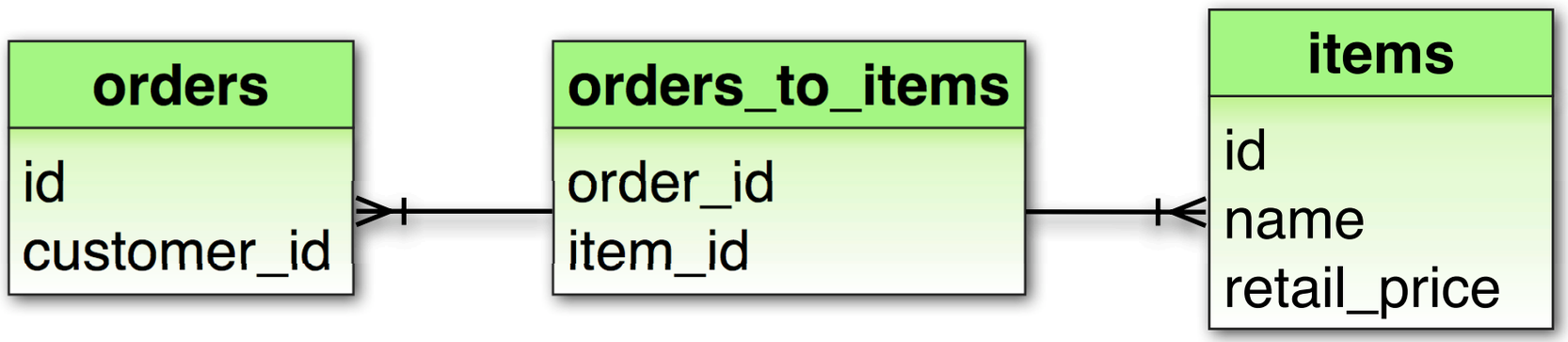
- (key, entity) pairs
- A keying function computes the key for each entity loaded from the database
- keyfunc(instance) must be invariant for the lifetime of a session, and the result must be hashable
- Tricky to use, but can be very useful

Many to Many



Many to Many


orders
id
customer_id
items
id
name
retail_price


orders
id
customer_id
orders_to_items
order_id
item_id
items
id
name
retail_price

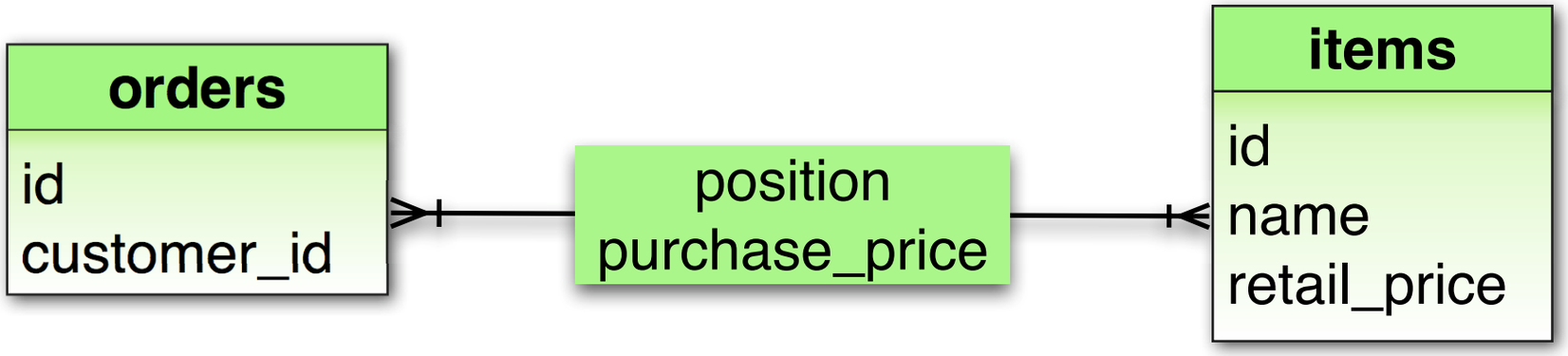
Mapped Association

secondary

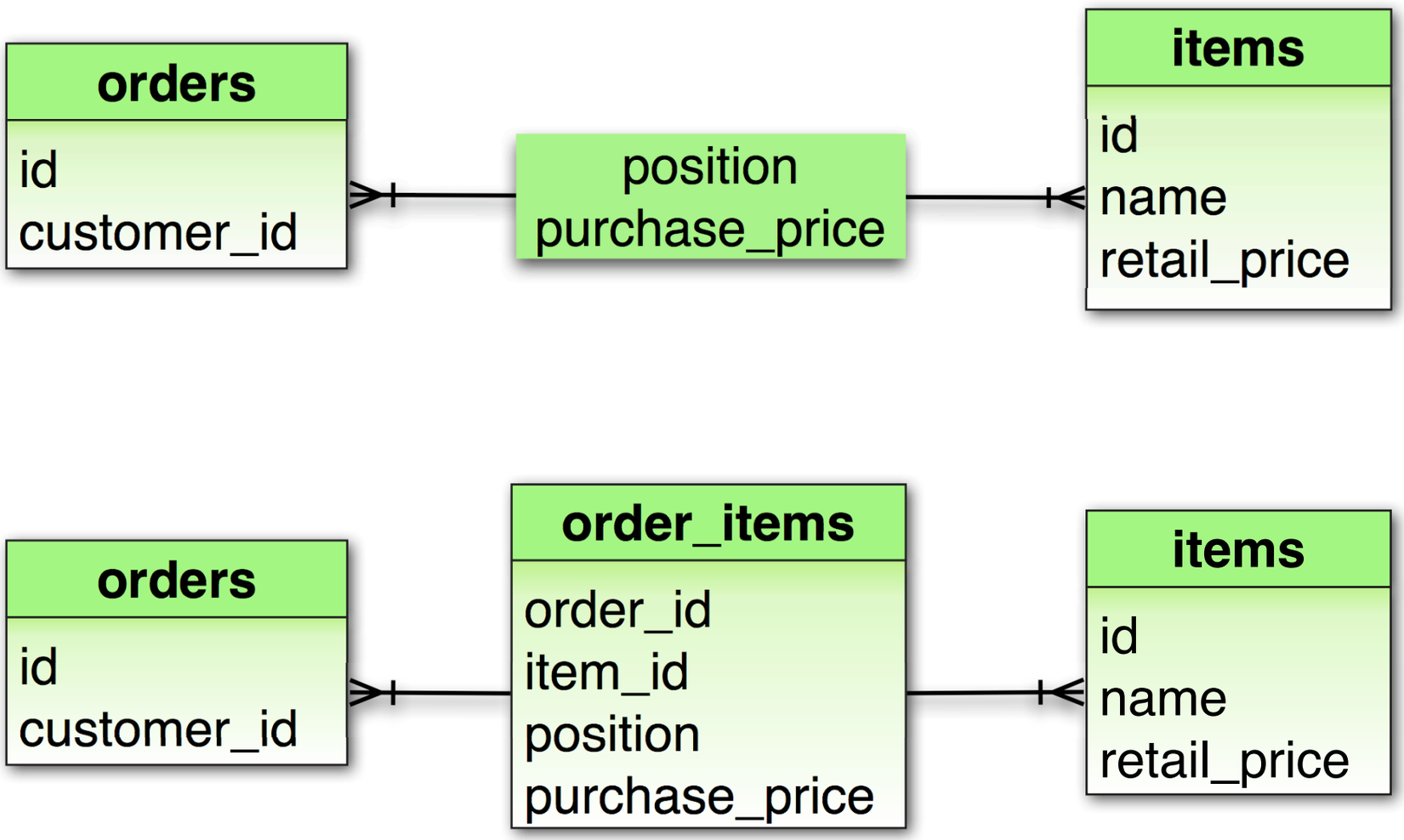

```
>>> mapper(Order, orders, properties={
        'items': relation(Item,
                          secondary=orders_to_items,
                          backref='orders')})
>>> mapper(Item, items)

>>> o = Order()
>>> ia = Item(name='item a')
>>> ib = Item(name='item b')
>>> o.items.extend([Item(), Item()])
```

Properties of a Relation



Properties of a Relation



Mapped Associations

properties of a relation

```
>>> mapper(Order, orders, properties={
      'orderitems': relation(OrderItem,
                             backref='order')})
>>> mapper(OrderItem, order_items,
        properties={'item': relation(Item)})
>>> mapper(Item, items)
```

Mapped Associations

properties of a relation


```
>>> order = session.query(Order).first()
>>> order.orderitems
[<OrderItem>]

>>> oi = order.orderitems[0]
>>> oi.order is order
True
>>> oi.item
<Item #1 u'plastic foo'>
>>> oi.purchase_price
Decimal('1')
```

Association Proxy

10,11

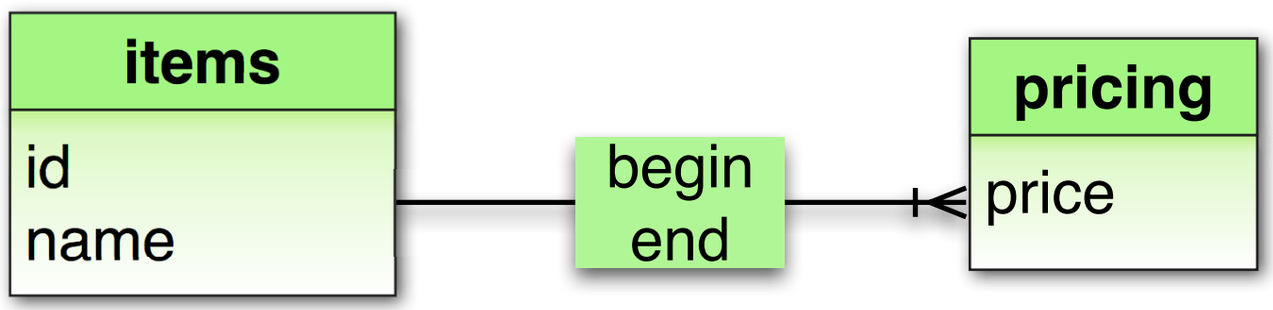
```
>>> [x.item for x in order.orderitems]
[<Item>]

class Order2(Order):
    items = association_proxy('orderitems', 'item')
```

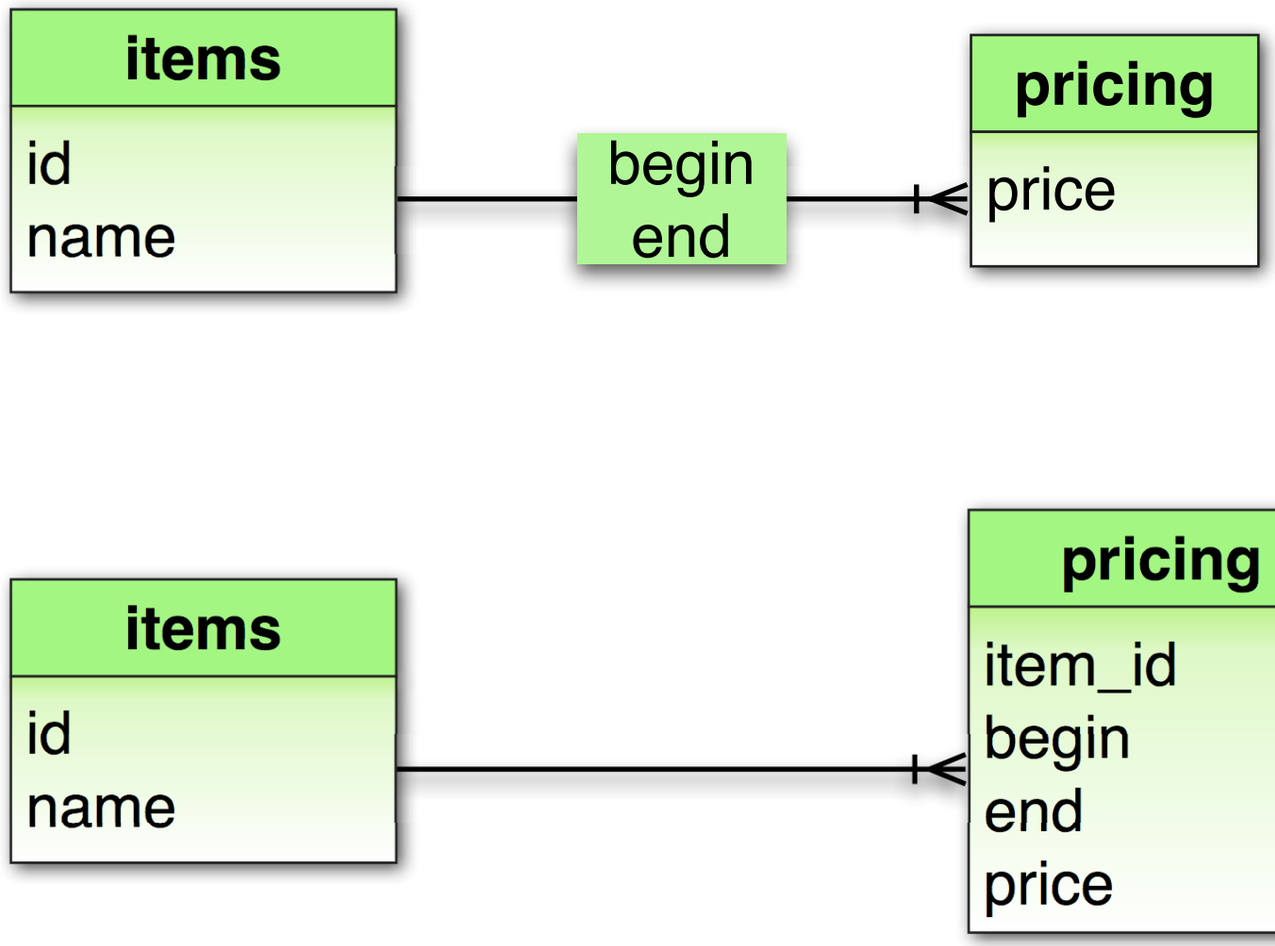
- Provides a Python-side view of a relation

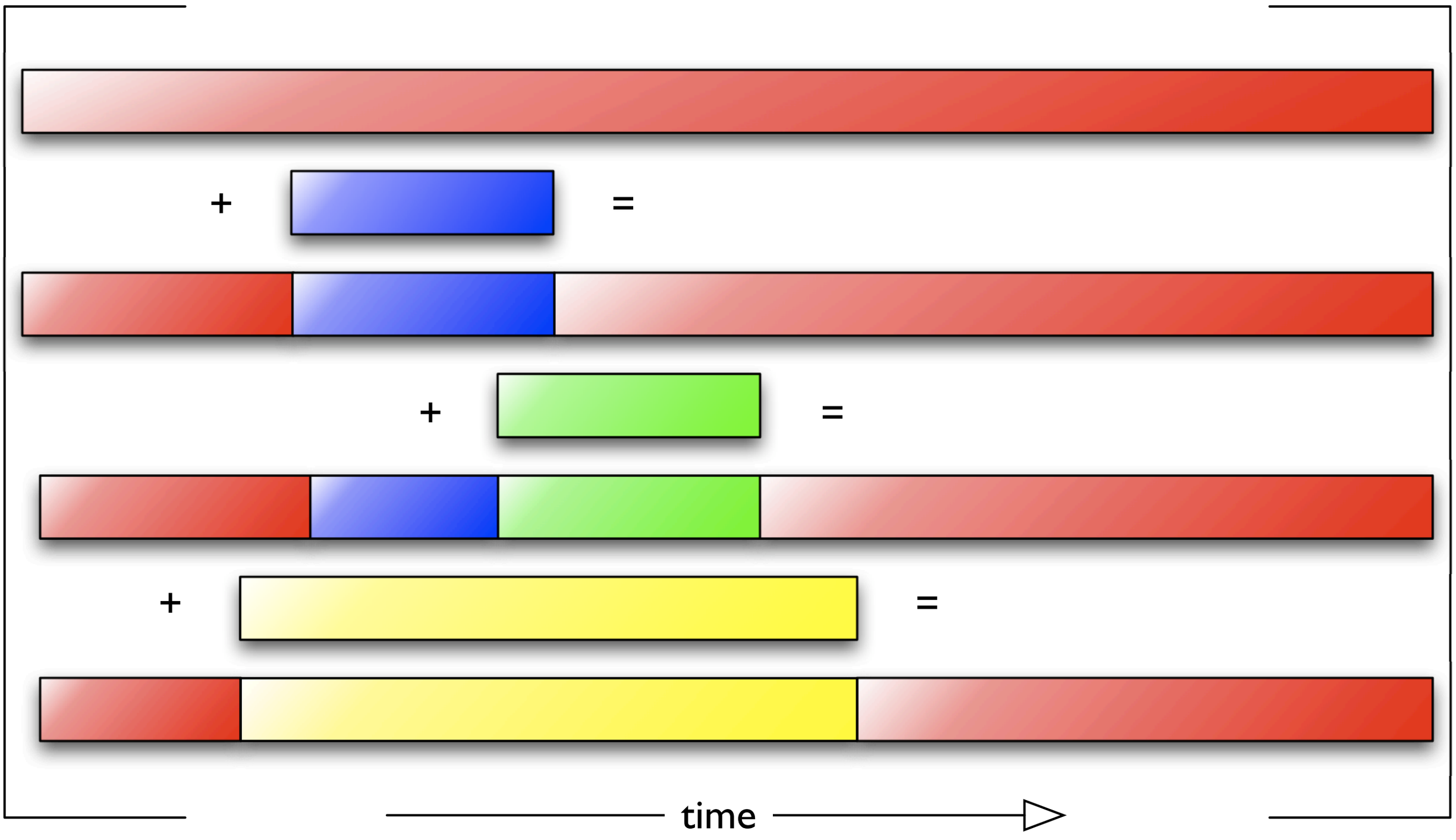
```
>>> order = session.query(Order).first()
>>> order.orderitems
[<OrderItem>]
>>> order.items
[<Item>]
```

Time Valued Attribute



Time Valued Attribute




+
=
+
=
+
=
time

A Mapping Challenge?

- Want an attribute on Item called .price_for
- item.price_for.today() returns a number
- item.price_for[date(2008, 1, 10)] returns a number
- item.price_for[date(2008, 1, 1), date(2008, 2, 1)] = 10

As a Collection

- Map ‘pricing’ table to a basic Pricing class
- Write a Python class that:
    - Can hold Pricing objects
        - Looks at their .begin and .end to find valid prices by date
        - Has a setter for begin, end and price; modifies, creates and deletes Pricings as required
- Add 3 line 1-line functions to adapt it to the SQLAlchemy ORM

```
>>> item.price_for.today()
Decimal('49.50')

>>> item.price_for[date(2008, 10, 10)]
Decimal('59.50')

# Special for October: only 1.00
>>> cheap = Decimal('1.00')
>>> item.price_for[date(2008, 10, 1),
                   date(2008, 11, 1)] = cheap
>>> item.price_for[date(2008, 10, 10)]
Decimal('1')
>>> item.price_for[date(2008, 9, 15)]
Decimal('59.50')
>>> item.price_for[date(2008, 11, 15)]
Decimal('59.50')
```

?

Session / flush()

Trickery

Embedding Expressions

```
u1 = User.query.get(2)
u1.name = User.name + ' smith'
Session.flush()

# SQL:

UPDATE users SET name=(users.name || ?)
WHERE users.id = ?
[' smith', 2]
```

Merge

- merge() loads a graph of objects into the session, corresponding to those within a given object graph; it then copies the state of the given objects onto the internal objects and returns them.

```
sess = create_session()

u1 = sess.query(User).get(2)
u1.addresses.append(Address
(email='ed@gmail.com'))
u1.name = 'ed'

u1 = Session.merge(u1)
Session.flush()
```

More on Merging

- Use merge() when you have a set of objects loaded from a variety of places, some of which may already be represented in the current session.
- merge() is also handy when restoring objects from a 2nd level cache back into the session.
- Transient instances passed to merge() will be copied into the Session as pending objects.
- The flag dont_load=True will disable loads, and will just copy objects instead. The given objects must have no pending changes (this will be improved eventually).

Merge is Not "Get or Create"

- merge() will create a pending instance from an instance that is transient or pending
- ...and will create a persistent instance from an instance that is persistent or detached.
- it will not create a persistent instance from a transient:

```
>>> u1 = User(id=2)
>>> u2 = Session.merge(u1)
>>> u2 is User.query.get(2)
False
```

Attribute Expiry / Refreshery


```
>>> u1 = User.query.get(2)
>>> u1.name = 'foobarfoobar'
>>> u1.addresses = [Address(email='lala')]
>>> Session.expire(u1)

>>> print u1.name, u1.addresses
ed [Address(id=2, email=u'ed@yahoo.com',
user_id=2), Address(id=3,
email=u'ed@msn.com', user_id=2), Address
(id=5, email=u'ed@gmail.com', user_id=2)]
```

Attribute Expiry / Refreshery

13,14

```
>>> u1 = User.query.get(2)
>>> u1.name = 'foobarfoobar'
>>> u1.addresses = [Address(email='lala')]

>>> Session.expire(u1, ['name'])

>>> print u1.name, u1.addresses
ed [Address(id=None, email='lala',
user_id=None)
```

Session Binding

```
Session.remove()

conn = meta.bind.connect()
Session(bind=conn)

Session.save(User(name='fred', addresses=[
   Address(email='fred@gmail.com')
]))

Session.commit()

Session.remove()
```

Joining External Transactions


```
Session.remove()

conn = meta.bind.connect()
trans = conn.begin()

Session(bind=conn)
u = User.query.filter_by(name='fred')
u.addresses.append(
   Address(email='xyz@python.org'))

Session.commit()

Session.remove()

trans.commit()
```

?

Thank You!

www.sqlalchemy.org

Advanced SQLAlchemy

March 13, 2008