SQLAlchemy

EuroPython 2010

Welcome

Why SQLAlchemy?

Session Goals

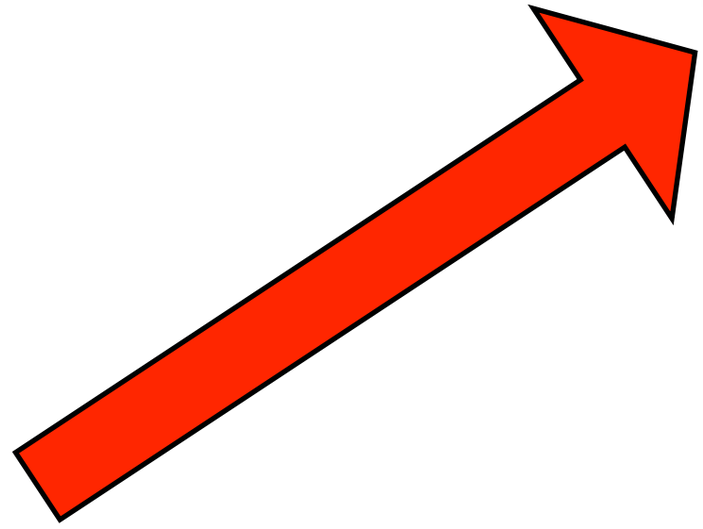
- Expose core concepts and code paths present in the toolkit
- Visit extension points and opportunities for customization

Format

- Illustration and discussion
- Case studies and executable code

Setup

```
$ python ready.py
```



Object Relational Mapping

SQL Expression Language

Database Connections & SQL Dialect Translation

1

part1.py

Connections and Dialects

Engines

```
>>> engine = engine_from_config(...)
```

```
>>> engine = create_engine('postgresql://...')
```

```
>>> Session.configure(bind=engine)
```

```
>>> session = create_session(engine)
```

```
>>> metadata.bind = engine
```

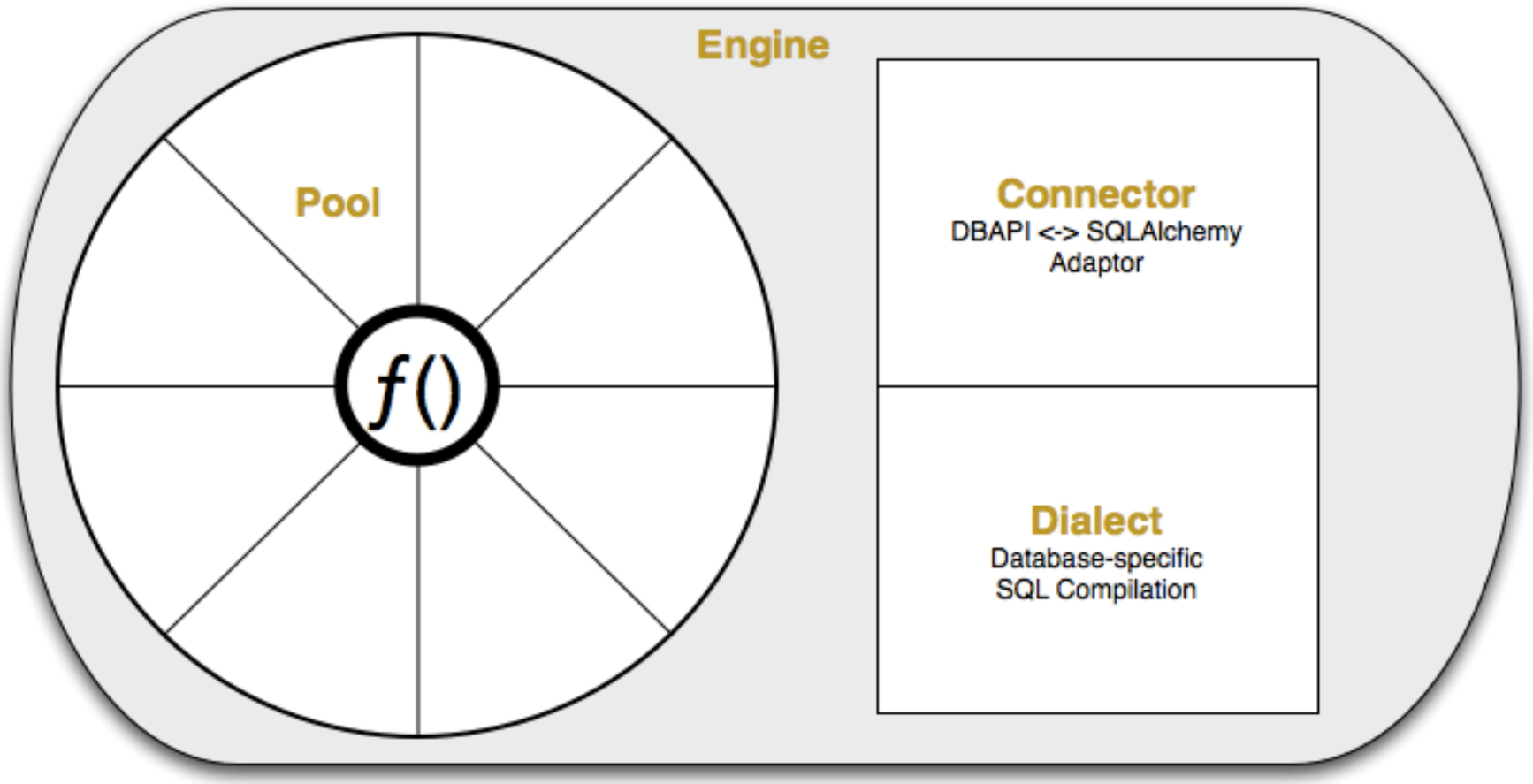
Engines

- Extensible: create your own
- To my knowledge no one has ever done this
- Why?

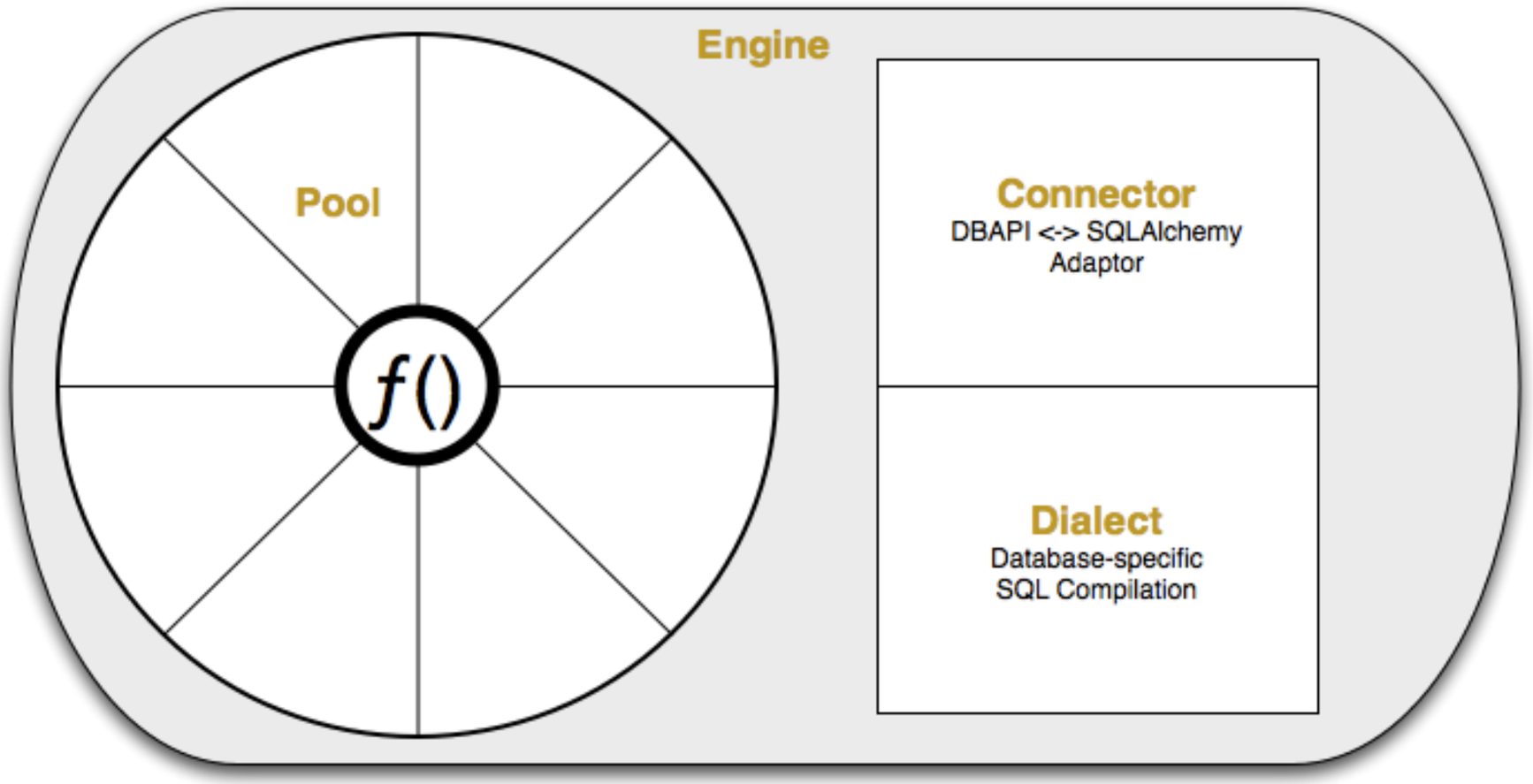
```
>>> engine = create_engine('sqlite:///:memory:')
>>> cx = engine.connect()
>>> results = cx.execute('SELECT 1')
```

```
>>> engine = create_engine('sqlite:///:memory:')
>>> cx = engine.connect()
>>> results = cx.execute('SELECT 1')
```

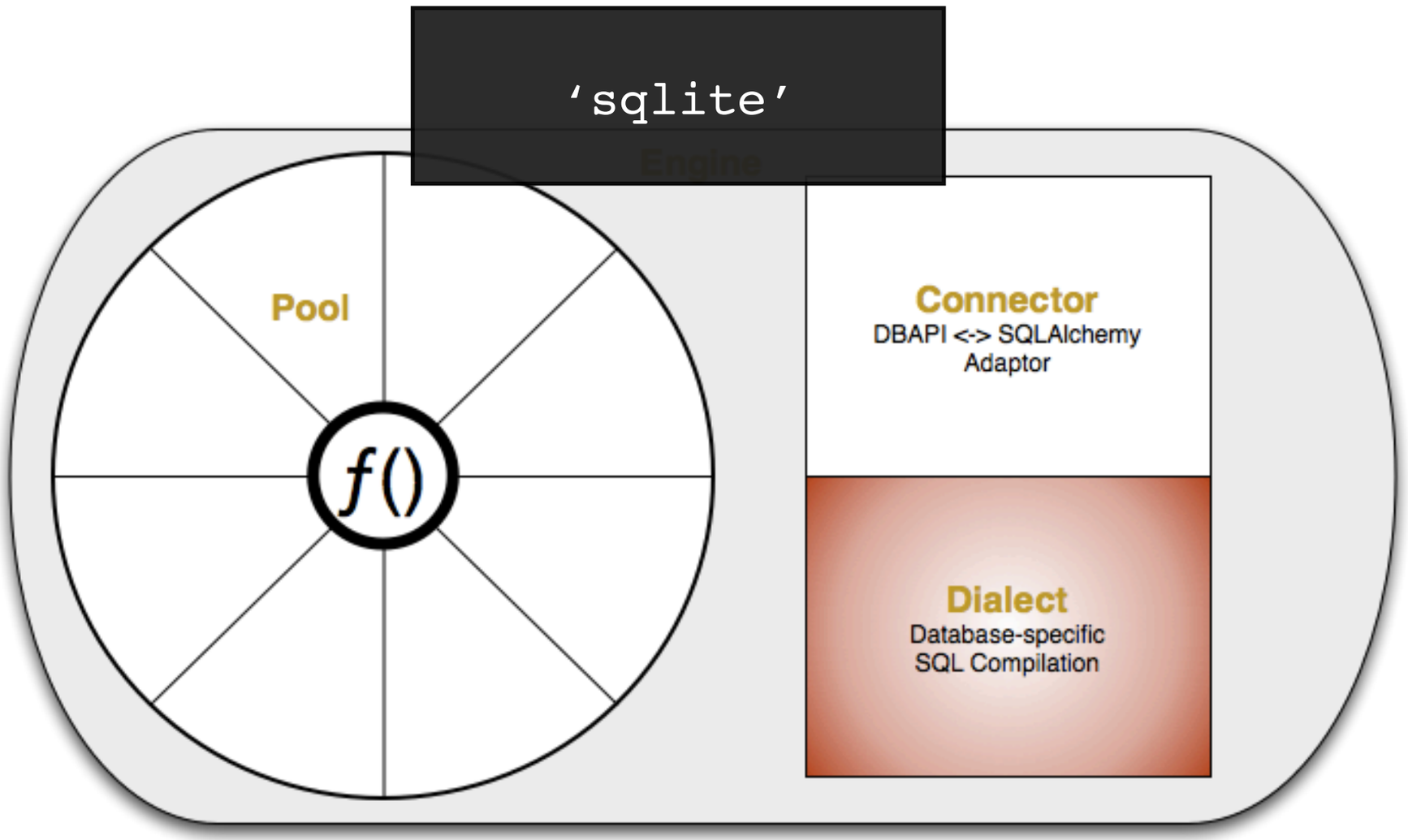
create_engine('sqlite:///:memory:')



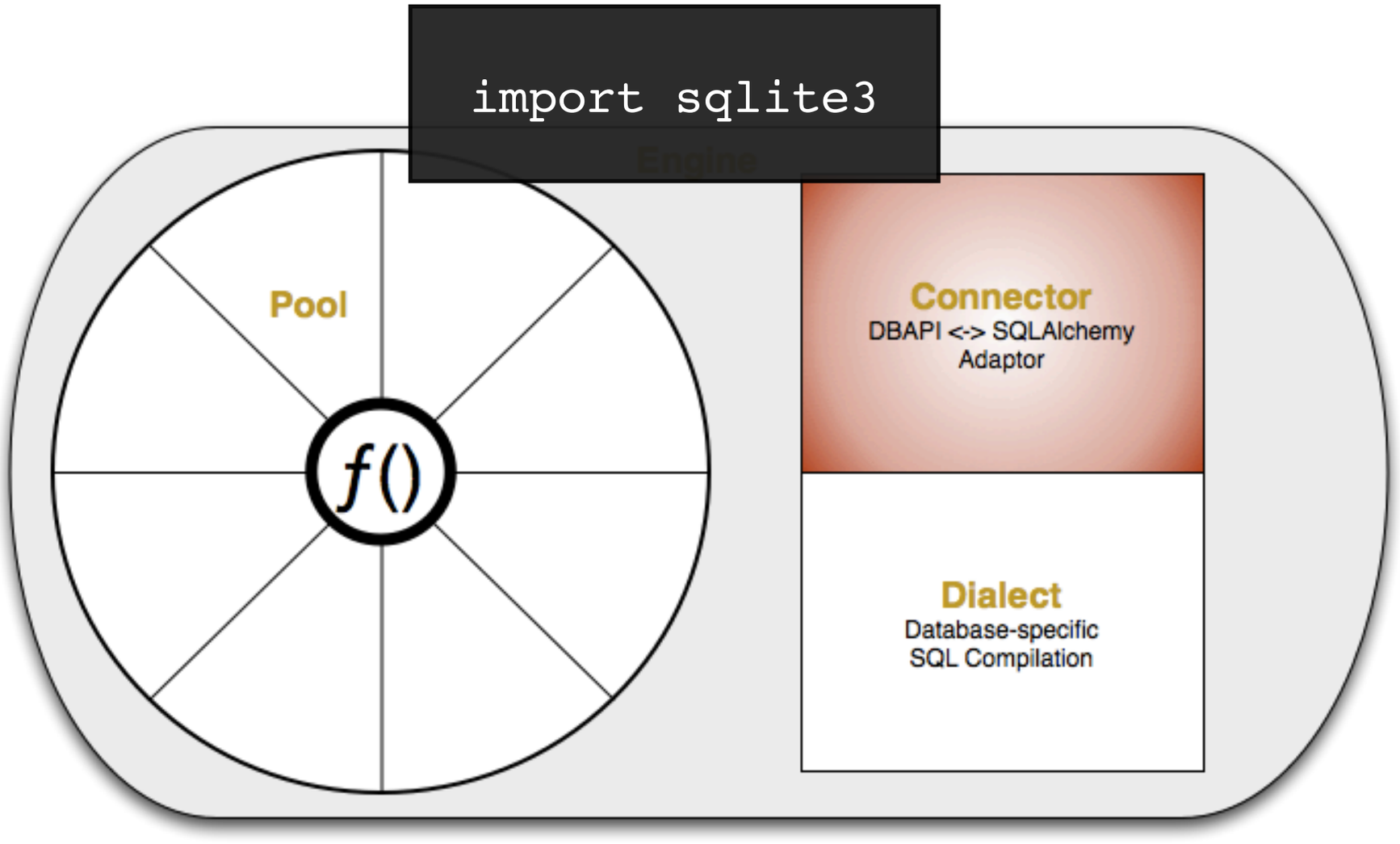
create_engine('sqlite:///:memory:')



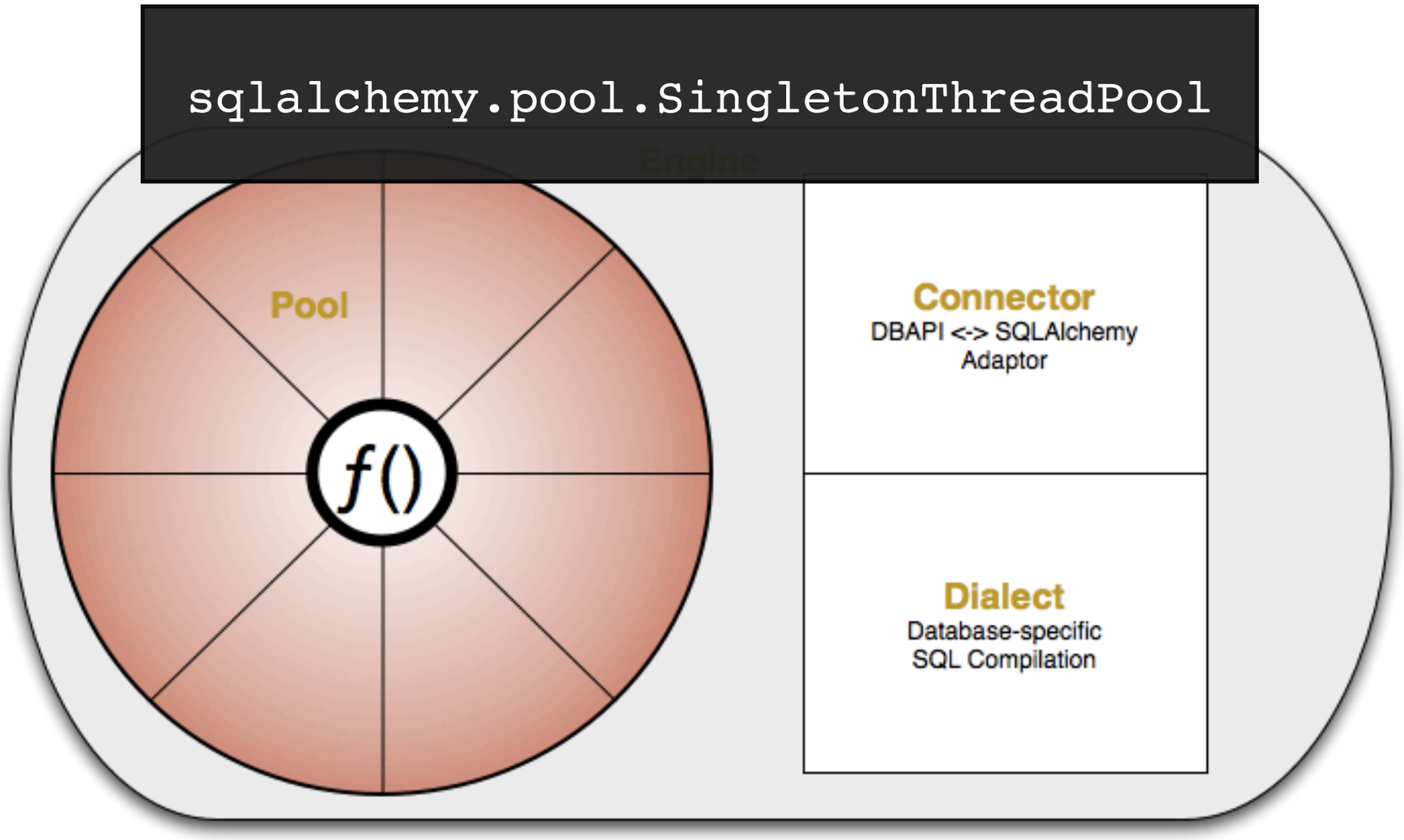
create_engine('sqlite:///:memory:')



create_engine('sqlite:///:memory:')

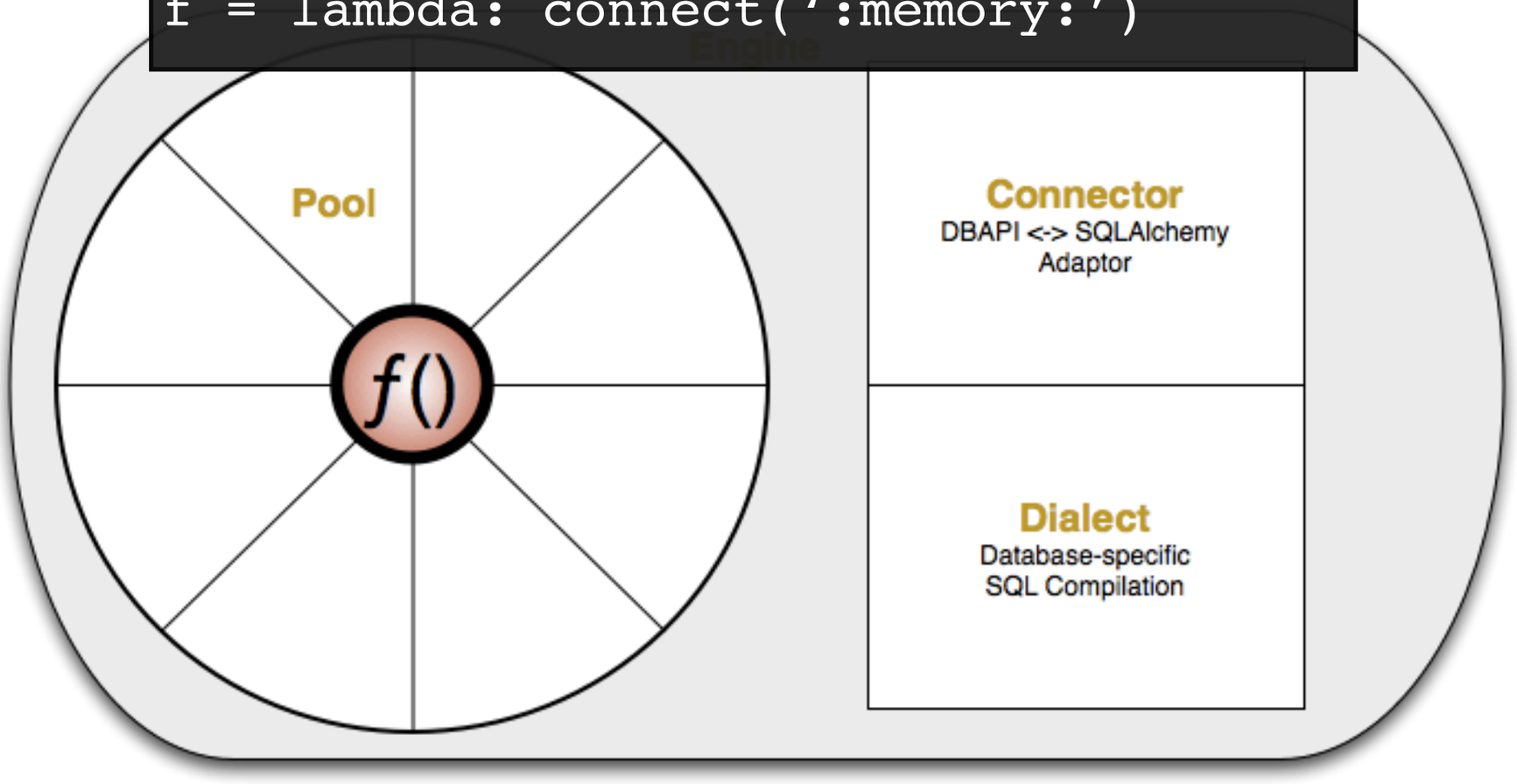


create_engine('sqlite:///:memory:')

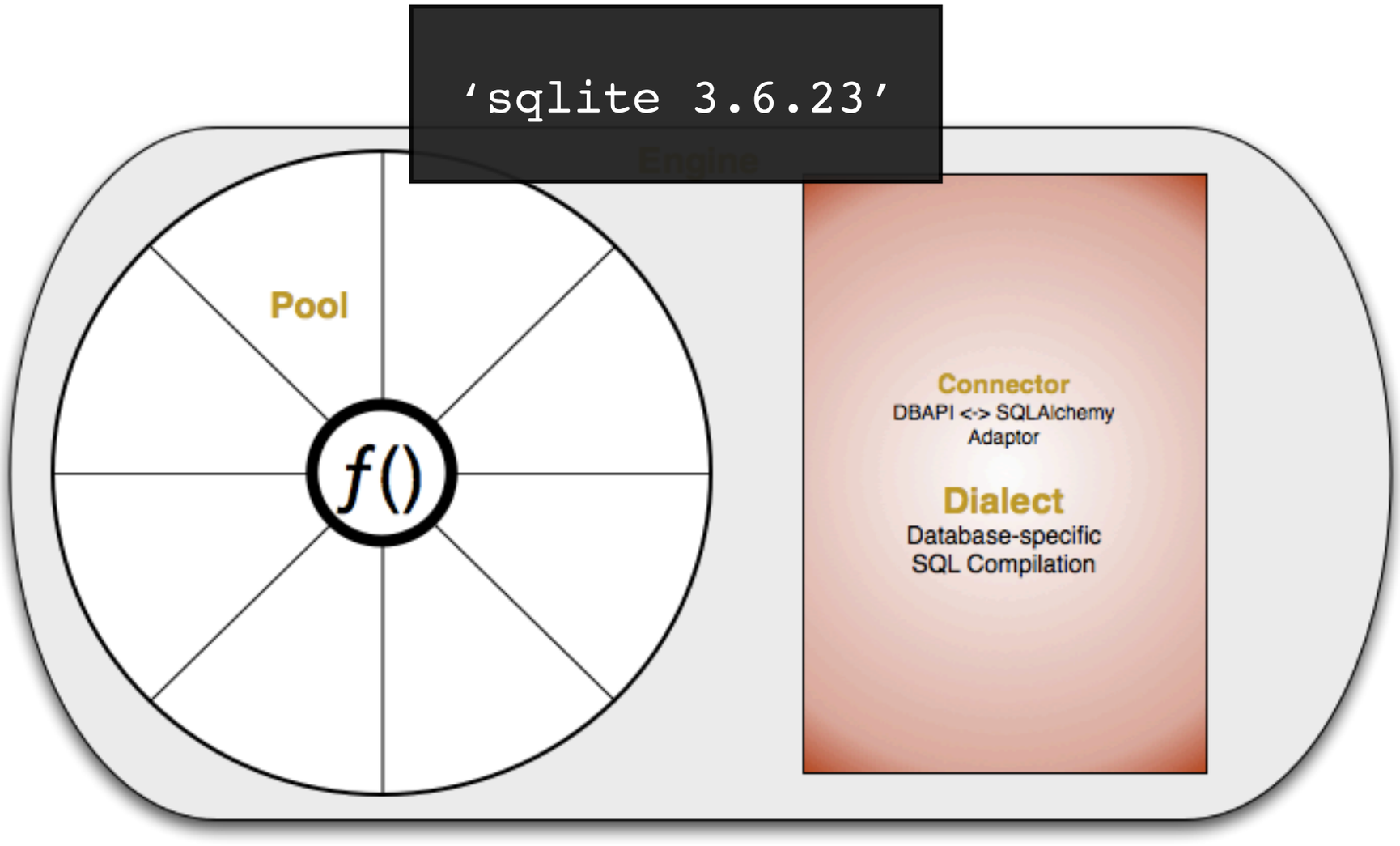


create_engine('sqlite:///:memory:')

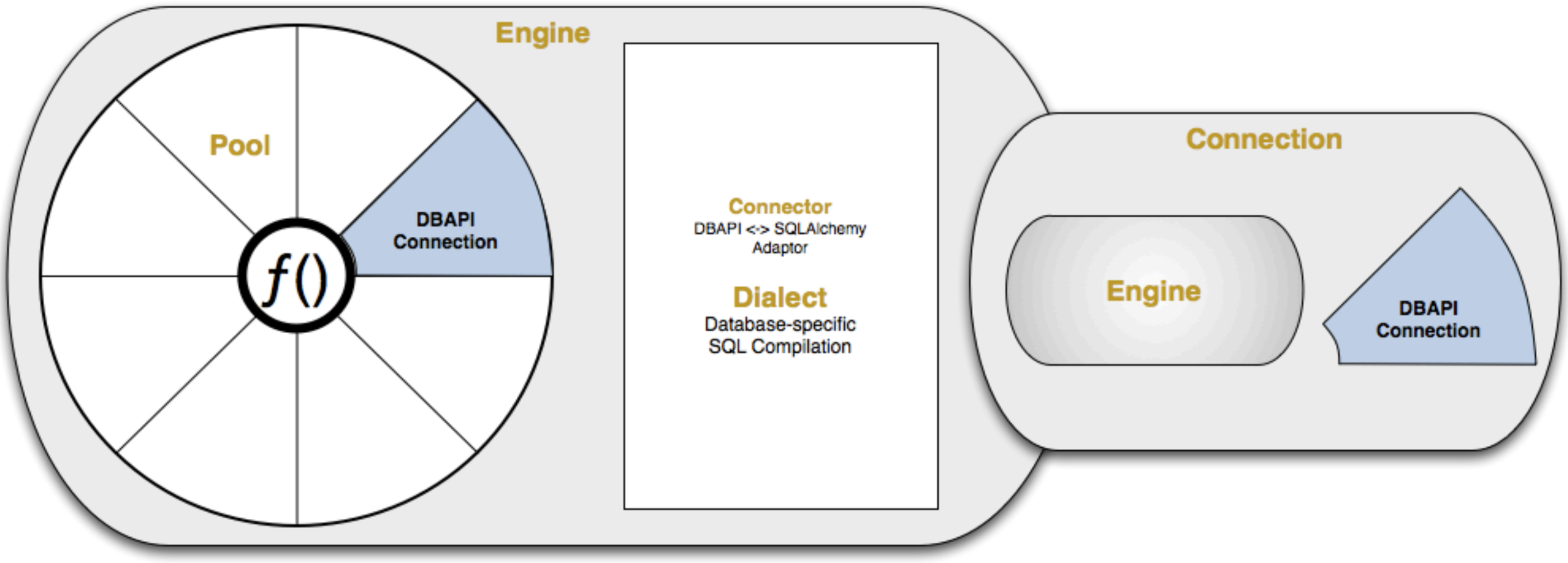
```
from sqlite3 import connect
f = lambda: connect(':memory:')
```



create_engine('sqlite:///:memory:')



```
>>> engine = create_engine('sqlite:///:memory:')

>>> cx = engine.connect()

>>> results = cx.execute('SELECT 1')
```



Pool

- Swappable: make your own pooling rules
- But the included pools are probably fine

```
>>> create_engine(poolclass=QueuePool)

>>> create_engine(pool=QueuePool(connect_fn, ...))
```

PoolListener

- Subscribe to pool and connection lifecycle events
- "Listener": subscriptions have limited ability to alter pool functionality. Listeners are not filters

PoolListener Events

- first_connect
- connect
- checkout
- checkin

PoolListener Events

- first_connect
- **connect**
- checkout
- checkin

set custom per-connection parameters with execute()

PoolListener Events

- first_connect
- connect
- checkout
- **checkin**

drop temporary tables

Case Study: disconnects

- If a `.execute()` operation encounters a disconnected DB-API connection, an exception is raised, the connection pool is emptied out & allowed to re-fill with fresh connections.

SQLAlchemy's Opinion

- "If a database connection is dropped, detect the disconnect on the first use of the connection. Application code should either back up and retry on a new connection, or raise an temporary error to the user."

Counter Opinion

- “The connection pool should always dispense valid connections.”

Approach

- Examine connection health as it is removed from the pool with a `checkout` listener
- If it is dead, raise `sqlalchemy.DisconnectionError`
- The pool will replace the dead connection with a fresh one

```
from sqlalchemy import exc


class LookLively(object):
    """Ensures that MySQL connections checked out of the pool are alive."""

    def checkout(self, dbapi_con, con_record, con_proxy):
        try:
            try:
                dbapi_con.ping(False)
            except TypeError:
                dbapi_con.ping()
        except dbapi_con.OperationalError, ex:
            if ex.args[0] in (2006, 2013, 2014, 2045, 2055):
                raise exc.DisconnectionError()
            else:
                raise


if __name__ == '__main__':
    from sqlalchemy import create_engine
    e = create_engine('mysql:///test', listeners=[LookLively()])
```

Connections

- ...provide a generic interface, and the Dialect provides runtime behavior appropriate for the database
- ...hold on to a DB-API connection until the Connection is `.close()`d or is garbage collected
- ...provide hooks for arbitrary metadata storage & execution interception

Pooled DB-API Connections

- The Pool creates a bookkeeping dictionary along with each DB-API connection it creates
- The dictionary lasts for the lifetime of the DB-API connection
- The dictionary is for your use and is called '`info`'

Pooled DB-API Connections

- The pool-managed storage is much easier than associating metadata with DB-API yourself
- Many DB-API implementations will re-use old connection object instances for new connections- same `id()`!

Connection .info

- Connection provides easy access to the low-level Pool-managed .info dictionary

```
>>> cx = engine.connect()
>>> cx.info['connected_at'] = time.time()
```

Case Study: audit log

- You'd like to record which user updated objects in your data model

Case Study: audit log

- Passing "current user" to all database-using functions in your app is not practical
- Your users are application-side, not actual database users or roles, so you can't use a database trigger

Approach

- Store “current user” in `Connection.info`
- Allow a column default to fill in the audit information anytime it is not explicitly provided by code

```
def updated_by(context):
    return context.connection.info.get('updated_by')

records = Table('records', metadata,
                Column('record_id', Integer, primary_key=True),
                Column('updated_by', Integer,
                       default=updated_by,
                       onupdate=updated_by),
                Column('data', String))
```

```
>>> cx = engine.connect()
>>> cx.info['updated_by'] = 123
>>> cx.execute(records.insert(), data='inserted')
<sqlalchemy.engine.base.ResultProxy object at 0x101724510>
>>> print cx.execute(records.select()).fetchall()
[(1, 123, u'inserted')]

>>> cx.info['updated_by'] = 456
>>> cx.execute(records.update().where(records.c.data == 'inserted'),
...                 data='updated')
<sqlalchemy.engine.base.ResultProxy object at 0x101724950>
>>> print cx.execute(records.select()).fetchall()
[(1, 456, u'updated')]
>>> cx.close()
```

Cleanup?

- Data in `.info` persists for the lifetime of the DB-API connection
- A pool listener is relatively fool-proof cleanup approach

```
def updated_by(context):
    return context.connection.info.get('updated_by')

def cleanup(dbapi_con, con_record):
    con_record.info.pop('updated_by', None)

records = Table('records', metadata,
                Column('record_id', Integer, primary_key=True),
                Column('updated_by', Integer,
                       default=updated_by,
                       onupdate=updated_by),
                Column('data', String))
```

```
>>> engine.pool.add_listener({'checkin': cleanup})
>>> cx = engine.connect()
>>> cx.info['updated_by'] = 789
>>> cx.execute(records.insert(), data='new row')
<sqlalchemy.engine.base.ResultProxy object at 0x101724dd0>
>>> cx.close()


>>> cx = engine.connect()
>>> print cx.info
{'connected_at': 1279397690.183189}
>>> cx.close()
```

ConnectionProxy

- Connections support an interception interface, allowing custom actions & behavior modification

ConnectionProxy

- begin()
  commit()
  rollback()
- execute()
- cursor_execute()
- ...

ConnectionProxy

- begin()
  commit()
  rollback()
- **execute()**
- cursor_execute()
- ...

count statements & execution times

Case Study: unit tests

- Deploy on PostgreSQL and test on SQLite?

Approach

- Don't drop & recreate all tables after each test
- Use a connection proxy to observe which tables were changed, and truncate them after tests finish

```
import re
from sqlalchemy import interfaces
import sqlalchemy.sql.expression as expr


class RDBMSChangeWatcher(interfaces.ConnectionProxy):
    safe_re = re.compile(
        r'\s*(?:CREATE|DROP|PRAGMA|SET|BEGIN|COMMIT|ROLLBACK)\b',
        re.I)

    def __init__(self):
        self.dirty = set()
        self.reset_all = False

    def execute(self, conn, execute, clauseelement, *multiparams, **params):
        action = type(clauseelement)

        if action == expr.Select:
            pass
        elif action in (expr.Insert, expr.Update, expr.Delete):
            self.dirty.add(clauseelement.table)
        elif action in (str, unicode):
            # Executing custom sql.  Could parse it, instead just reseting
            # everything.
            if not self.safe_re.match(clauseelement):
                self.reset_all = True
        else:
            self.reset_all = True
```

```
class RDBMSChangeWatcher(interfaces.ConnectionProxy):
    # ...
    def cleanup(self, metadata, connection):
        if not (self.dirty or self.reset_all):
            return

        transaction = connection.begin()
        try:
            if self.reset_all:
                for table in reversed(metadata.sorted_tables):
                    connection.execute(table.delete())
            else:
                for table in reversed(metadata.sorted_tables):
                    if table in self.dirty:
                        connection.execute(table.delete())
            self.clear()
        finally:
            transaction.commit()

    def clear(self):
        self.dirty.clear()
        self.reset_all = False
```

```
>>> watcher = RDBMSChangeWatcher()
>>> engine = create_engine('sqlite:///', proxy=watcher)
>>> records.create(engine)
>>> print engine.execute(records.select()).fetchall()
[]
>>> print 'dirty', [t.name for t in watcher.dirty]
dirty []

>>> engine.execute(records.insert(), data='first row')
<sqlalchemy.engine.base.ResultProxy object at 0x101735190>
>>> print 'inserted', engine.execute(records.select()).fetchall()
inserted [(1, None, u'first row')]
>>> print 'dirty', [t.name for t in watcher.dirty]
dirty ['records']

>>> watcher.cleanup(metadata, engine.connect())
>>> print 'post-cleanup', engine.execute(records.select()).fetchall()
post-cleanup []
```

Dialects

- Translate generic SQLAlchemy constructs into vendor SQL & accommodate database driver quirks
- Can be developed & distributed separately via `pkg_resources` entry points (setuptools, distribute).

- Everything goes through `engine.connect()`
- There is always a Pool, even if a pool of one
- Dialects do all of the heavy lifting

Up Next: DDL and SQL Expression Language

Questions?

part2.py

1

DDL and SQL Expression

SQL Expression Layer

- Tables provide Columns, data types and can emit DDL (`CREATE TABLE`)
- Expressions create queries and DML (`SELECT`) using Columns and Tables

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String))
```

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String))
```

- MetaData provides a container for related tables
- "related" is flexible, however foreign keys can not point into another MetaData

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String))
```

- Tables hold a collection of Columns, constraints and metadata

- Much of the metadata tables can hold is not used outside of `CREATE TABLE` operations: for example, indexes, unique constraints

- Tables also have `.info` dictionaries for your use

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String))
```

- Columns are named and have a Type
- Columns may hold additional information for use during `CREATE TABLE`
- Columns have `.info`

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String))
```

- `sqlalchemy.types` provides best-fit and exact data types
  - Integer vs INT
- Type implementations in Dialects provide translation to and from native DB-API data formats

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String)
              [redacted]
              )
```

- Extension points include:
  - Types
  - SchemaItems
  - DDL events

Extending the Non-Extensible

Case study: timestamps

- You require “last updated at” timestamps on all tables

Case study: timestamps

- Subclassing Table is not a good option
- Creating Columns in advance will not work

```
from sqlalchemy import DateTime


LAST_UPDATED = Column('updated_at', DateTime, nullable=False)

table_1 = Table('table_1', metadata,
                Column('id', Integer, primary_key=True),
                LAST_UPDATED)

table_2 = Table('table_2', metadata,
                Column('id', Integer, primary_key=True),
                LAST_UPDATED)
```

```
Table('table_1', metadata,
      Column('id', Integer, primary_key=True))
```

```
>>> col = Column.__init__('id', Integer, ...)
>>> tbl = Table.__init__('table_1', metadata, col)
>>> col._set_parent(tbl)
```

Approach

- Use a factory function that wraps Table and produces new Column objects on each invocation

6,7

```
def timestamped_table(*args, **kw):
    final_args = list(args) + [
        Column('updated_at', DateTime, nullable=False)
        ]
    return Table(*final_args, **kw)
```

- Use this approach for Column and other SchemaItems
- Wrapper functions are the recommended way to extend Mapper as well. More on that later

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String)
              [redacted]
              )
```

- Extension points include:
  - Types
  - SchemaItems
  - DDL events

Extending Types

- `sqlalchemy.types.TypeDecorator`
  - Map a known database type to a new python type
- `sqlalchemy.types.UserDefinedType`
  - Map an unknown database type to python

- For either implementation, you provide a function to translate DB-API result data to Python, and a function to translate Python to DB-API parameter data

Case Study: timezones

- You're storing timezone information & it would be more convenient for your code if you work only with python `datetime.tzinfo` objects.

Approach

- Write a TypeDecorator that decorates a character column with conversion to and from `pytz` objects

```
from sqlalchemy.types import TypeDecorator
from pytz import timezone


class Timezone(TypeDecorator):
    impl = String

    def process_bind_param(self, value, dialect):
        if isinstance(value, (basestring, type(None))):
            return value
        return value.zone

def process_result_value(self, value, dialect):
    if value is None:
        return value
    try:
        return timezone(value)
    except NameError:
        return value
```

```
events = Table('events', metadata,
               Column('id', Integer, primary_key=True),
               Column('occurred_at', DateTime),
               Column('occurred_tz', Timezone))
```

- Associating unique column values with Python singleton instances can be a very powerful pattern
- Enumerated constants, lightweight static association tables, geo data

UserDefinedTypes

- Best for database types you would use in a `CREATE TABLE` statement
- Great for vendor extensions
- `HSTORE`, `PERIOD`, ...

```
from sqlalchemy import MetaData, Table, Column, String, Integer

metadata = MetaData()

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('email', String)
              [redacted]
              )
```

- Extension points include:
  - Types
  - SchemaItems
  - DDL events

More `.info`

- Tables and Columns have `.info` storage as well
- Completely user defined

- Table and Column `.info` has been known to be used to store SQL comments for self-describing data dictionaries, form labels and other metadata for GUI use...

DDL Events

- MetaData and Table emit events during `CREATE` and `DROP` operations

```
>>> table.append_ddl_listener(event, listener)
```

  - before-create, after-create
  - before-drop, after-drop

```
>>> def listener(event, target, connection): ...
```

- DDL() makes it simple to customize generation for database-specific needs
  - A simple wrapper that builds on DDL events

```
>>> stmt = DDL('ALTER TABLE foo SET WITH OIDS',
...            on='postgresql')
>>> stmt.execute_at('after-create', table_foo)
```

- DDL statements can be reused and applied to multiple tables with templating

```
>>> stmt = DDL('CREATE TRIGGER %(table)s_ins '
...            'BEFORE INSERT ON %(table)s '
...            'EXECUTE ...')

>>> for table in metadata.tables.sorted_tables:
...     stmt.execute_at('after-create', table)
```

Case Study: fixtures

- Some tables in your database have a fixed or seldom-changing set of data
- You want to be able to create empty databases for testing, but that fixed data is required for your app to function
- Fixed data and the table schema should not be allowed to drift apart

Approach

- Define fixed data alongside the Table- or better, in the Table definition
- Use ‘after-create’ DDL event to issue `INSERT`s
- Make fixed data introspectable in python without need for a `SELECT`

```
Table('location', metadata,
      Column('x', Integer),
      Column('y', Integer),
      Fixture(('x', 'y'),
              (10, 10),
              (20, 20)),
      )
```

```
class Fixture(object):
    """Associate a fixed data set with a Table."""

    def __init__(self, column_names, *rows):
        self.column_names = tuple(column_names)
        self.rows = list(rows)

    def _set_parent(self, table):
        """Implements sqlalchemy.schema.SchemaItem._set_parent."""
        table.append_ddl_listener('after-create',
                                  self.load_fixture_data)
        table.info['fixture'] = self

def load_fixture_data(self, event, schema_item, connection):
    """Unconditionally load fixed data into a Table."""
    insert = schema_item.insert()
    data = (dict(zip(self.column_names, values))
            for values in self.rows)
    connection.execute(insert, *data)
```

```
>>> locations = Table('location', metadata,
...                   Column('x', Integer),
...                   Column('y', Integer),
...                   Fixture(('x', 'y'),
...                           (10, 10),
...                           (20, 20)),
...                   )

>>> cx = engine.connect()
>>> locations.create(cx)
>>> print cx.execute(locations.select()).fetchall()
[(10, 10), (20, 20)]
>>> print locations.info['fixture'].rows
[(10, 10), (20, 20)]
```

SQL Expressions

```
>>> select([users.c.id]).\
... where(users.c.email.startswith('jek@')
```

```
>>> print select([func.now()])
SELECT now() AS now_1
```

sqlalchemy.func

- Generator, renders out any parenthesized SQL function
- `func.now()`, `func.group_concat(..., ...)`

Complex Clauses

- `CASE ...`
- Use `text()`
- Subclass existing bases
- Or use the compiler extension

Case Study: utcnow

- You need to generate UTC timestamps in insert and update clauses
- Date handling functions vary among the databases you are targeting

Approach

- Use the compiler extension to create a timestamp function that generates native SQL for each database target

```python
from sqlalchemy.sql.expression import ColumnElement
from sqlalchemy.ext.compiler import compiles


class utcnow(ColumnElement):
    type = DateTime()


@compiles(utcnow, 'sqlite')
def compile_timestamp(element, compiler, **kw):
    return "datetime('now')"


@compiles(utcnow, 'postgresql')
def compile_timestamp(element, compiler, **kw):
    return "TIMESTAMP 'now' AT TIME ZONE 'utc'"

@compiles(utcnow)
def compile_timestamp(element, compiler, **kw):
    return "current_timestamp"
```

```
>>> from sqlalchemy.dialects.postgresql import dialect as postgres
>>> from sqlalchemy.dialects.sqlite import dialect as sqlite

>>> print utcnow().compile(dialect=sqlite())
datetime('now')
>>> print utcnow().compile(dialect=postgres())
TIMESTAMP 'now' AT TIME ZONE 'utc'
>>> print utcnow()
current_timestamp
```

Up Next: ORM

Questions?

part3.py

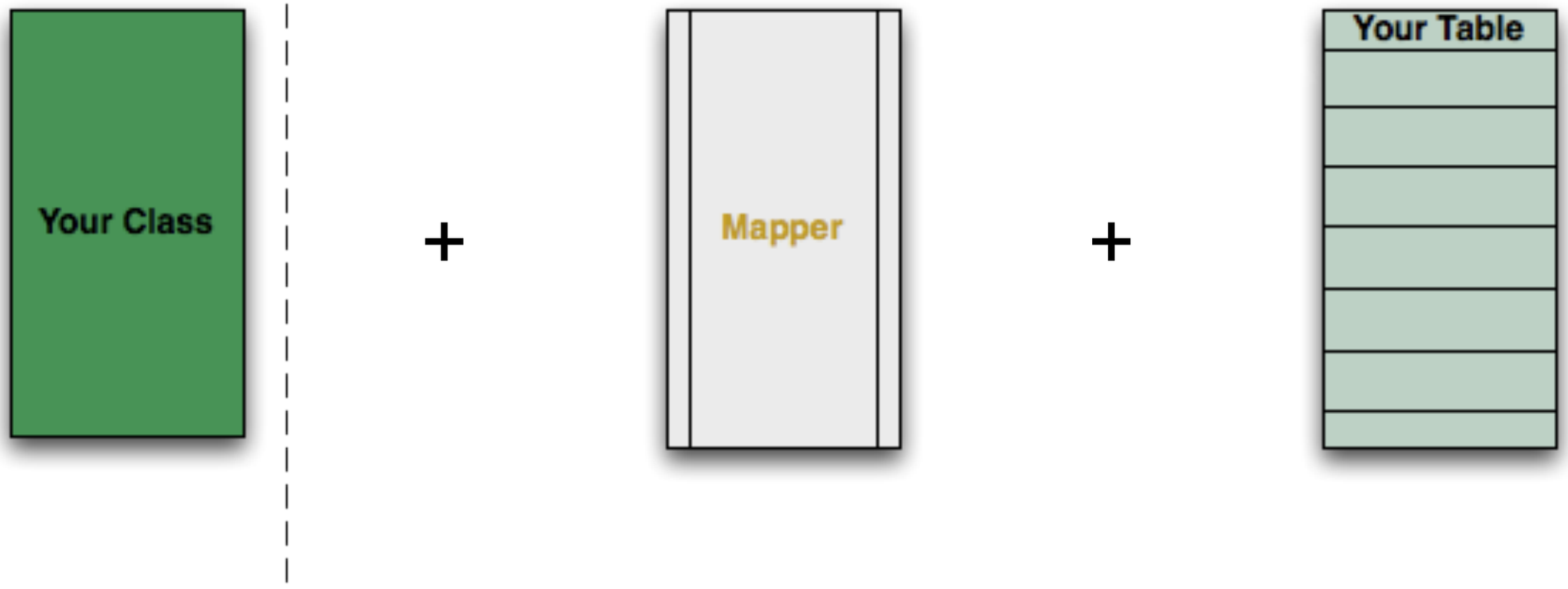
1,2

ORM

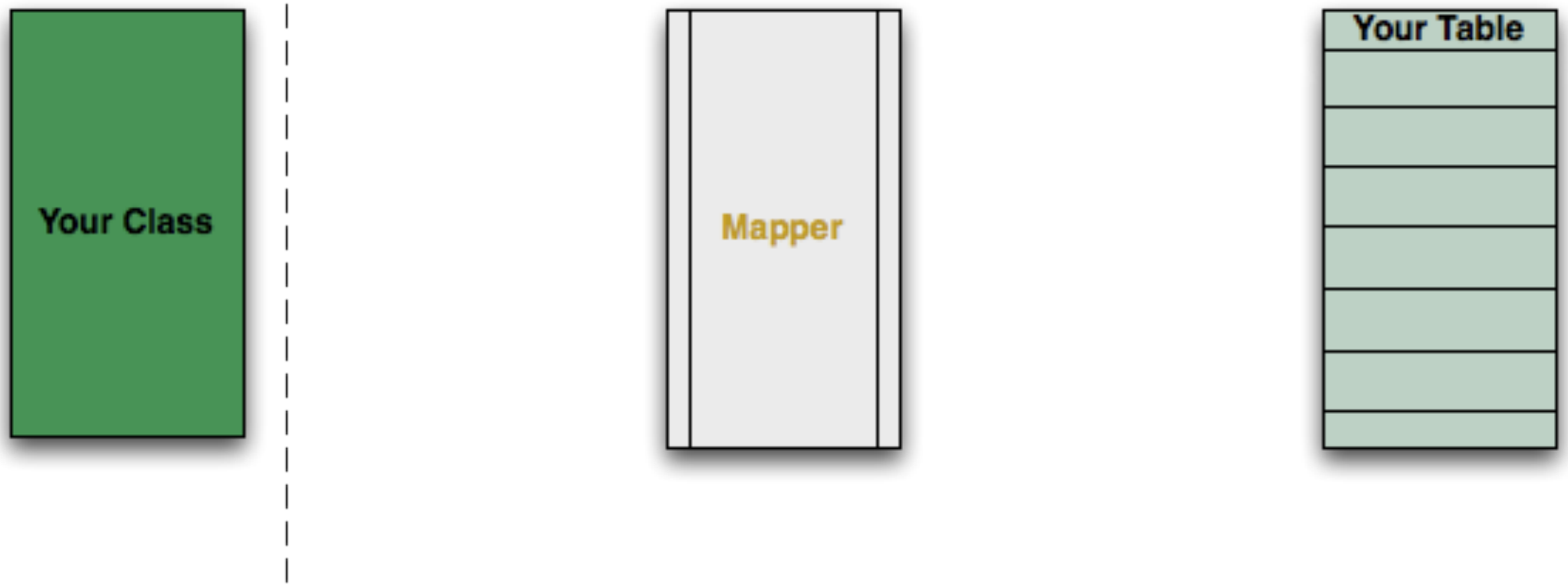
```
mapper(User, users_table, properties={
    'comments': relation(Comment, backref='posted_by'),
    })
```

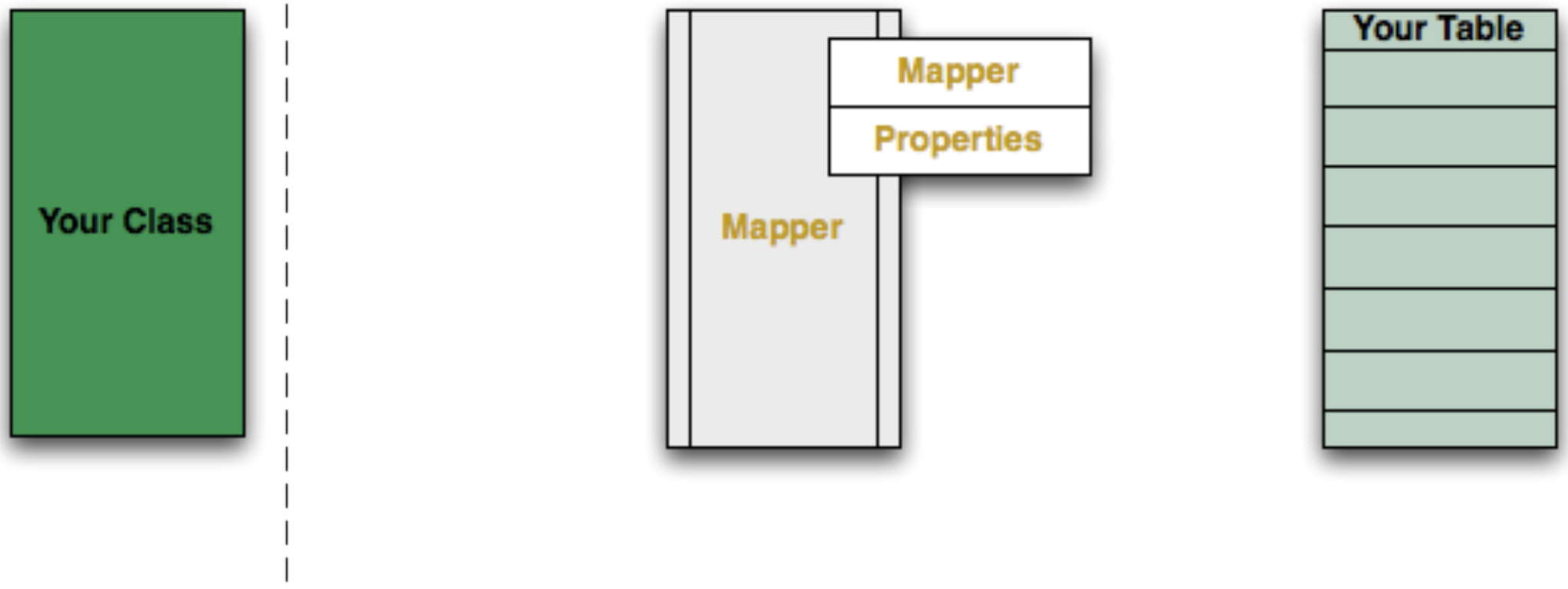
```
class User(Base):
    __table_name__ = 'users'

    comments = relation('Comment', backref='posted_by')
```

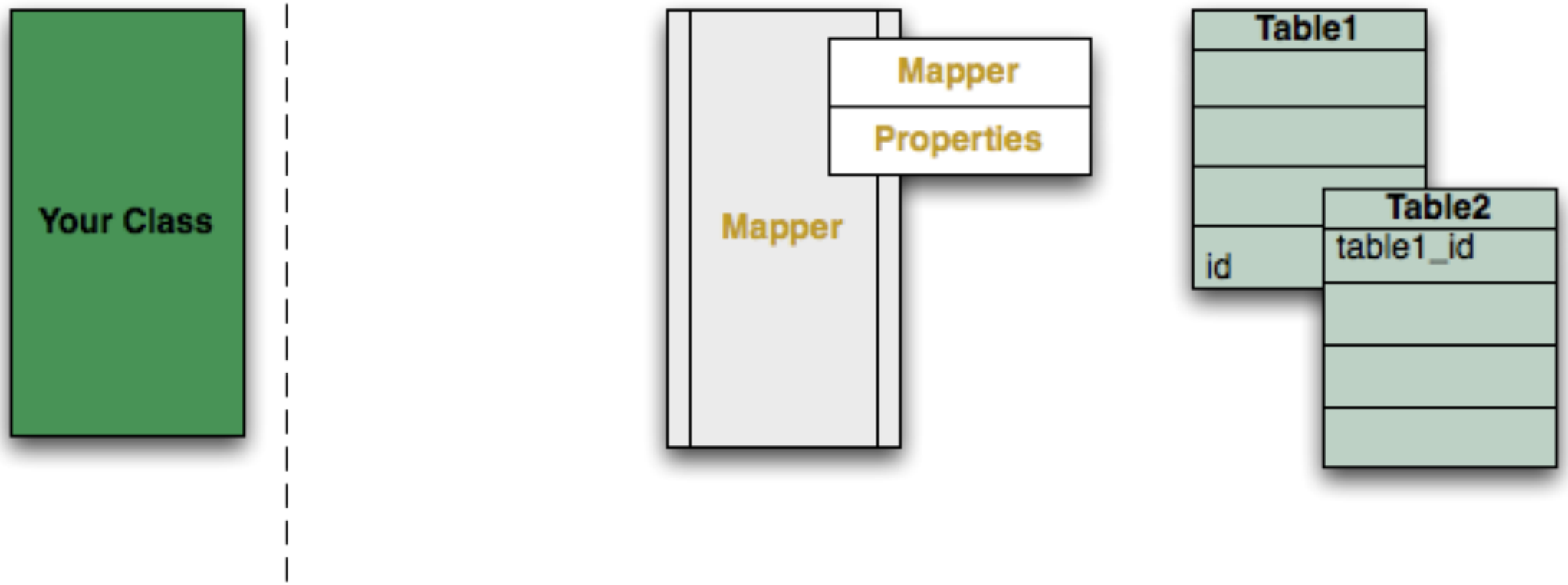
```
>>> Session = scoped_session(sessionmaker())
>>> session = Session()
>>> user = session.query(User).filter(
...     User.email.startswith('jek').one()
```

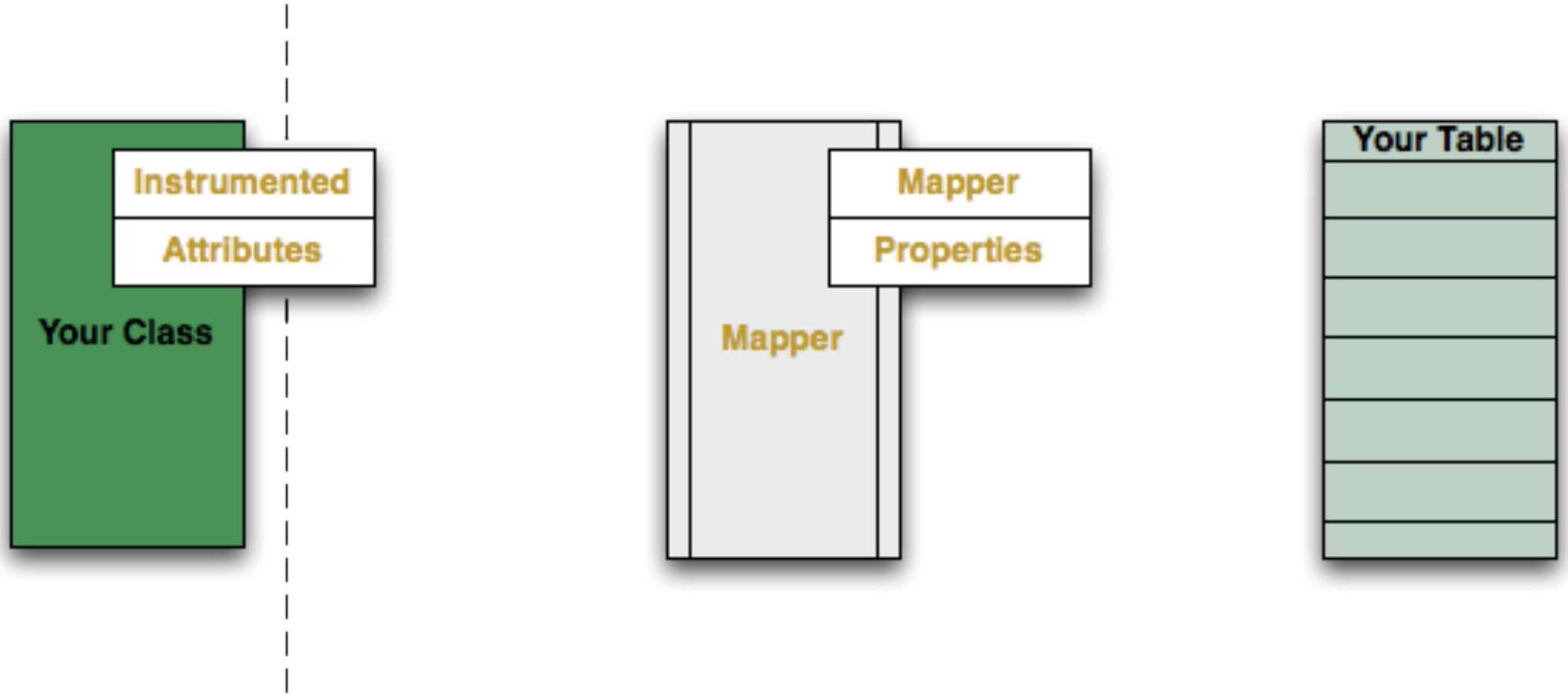


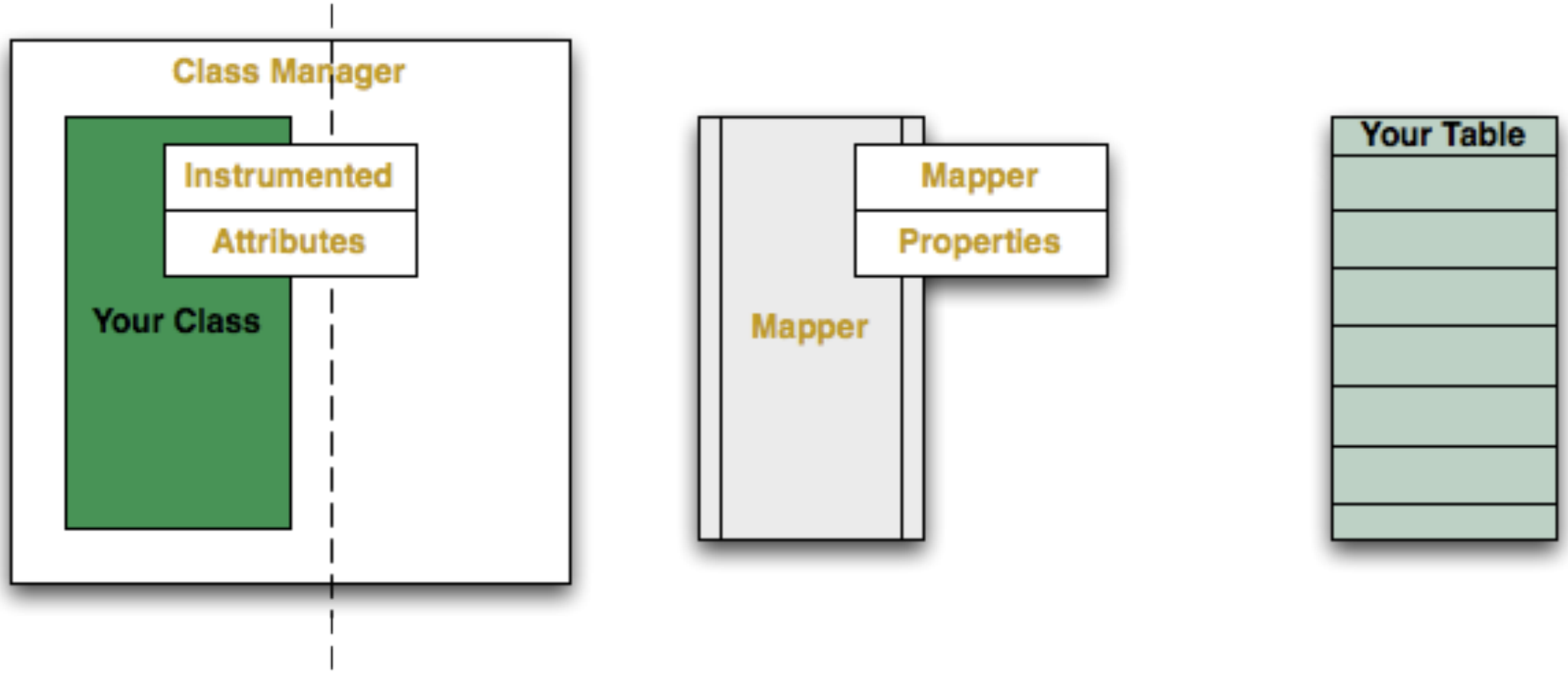
== magic

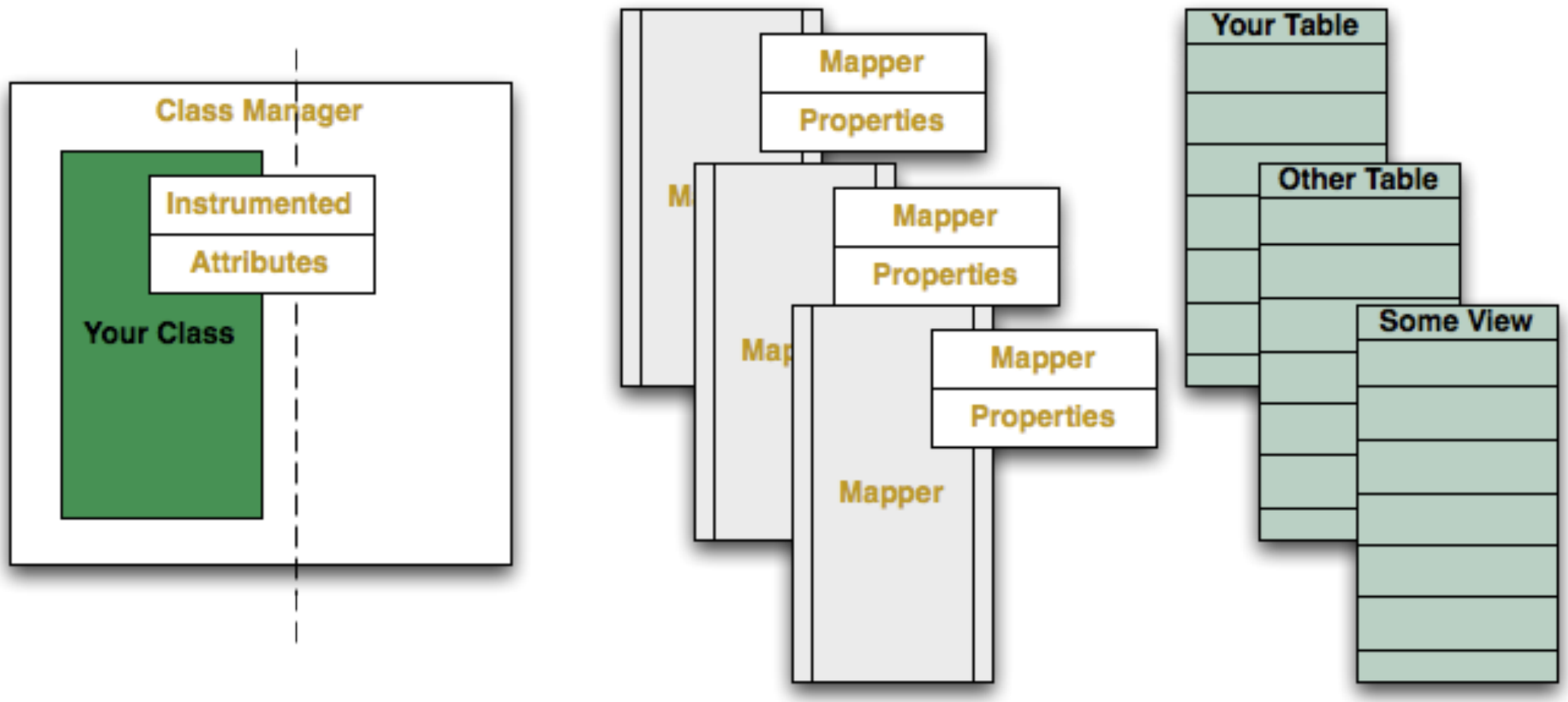

Your Class
Mapper
Your Table

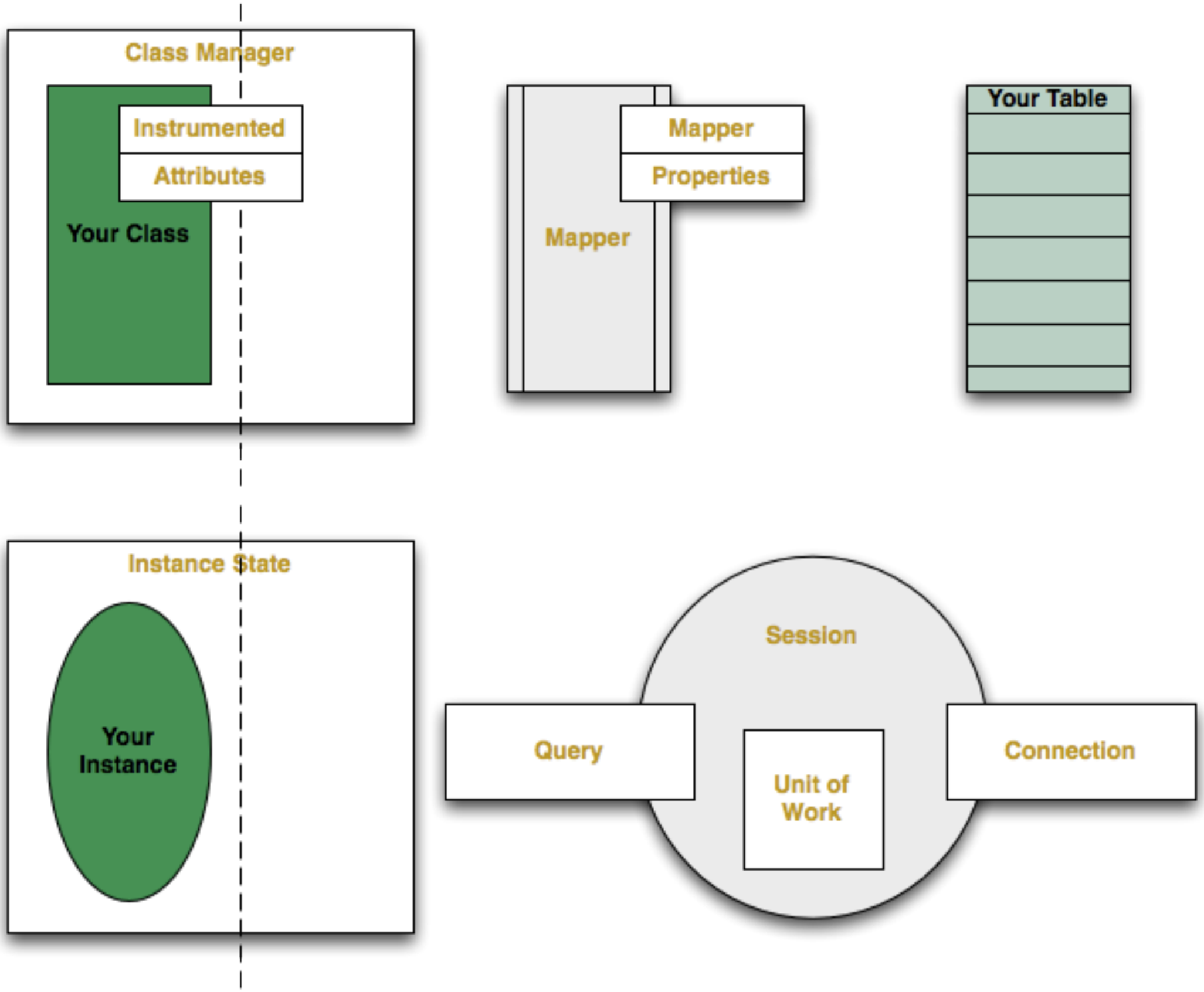

Your Class
Mapper
Properties
Mapper
Your Table

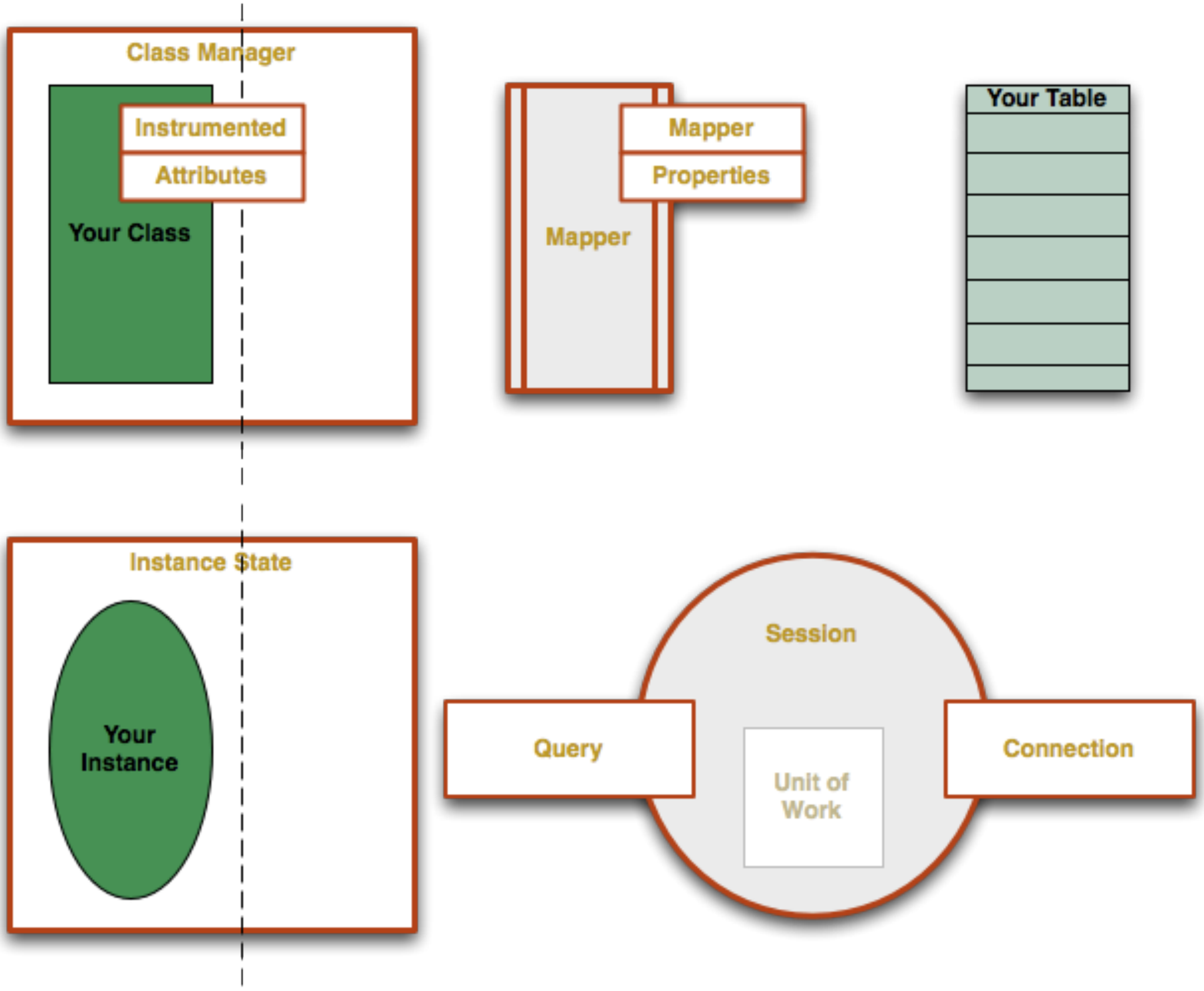

Your Class
Mapper
Properties
Mapper
Table1
Table2
table1_id
id


Instrumented
Attributes
Your Class
Mapper
Properties
Mapper
Your Table


Class Manager
Instrumented
Attributes
Your Class
Mapper
Properties
Mapper
Your Table


Class Manager
Instrumented
Attributes
Your Class
Mapper
Properties
Mapper
Properties
Mapper
Properties
Mapper
Your Table
Other Table
Some View


Class Manager
Instrumented
Attributes
Your Class
Mapper
Properties
Mapper
Your Table
Instance State
Your
Instance
Session
Query
Unit of
Work
Connection


Class Manager
Instrumented
Attributes
Your Class
Mapper
Properties
Mapper
Your Table
Instance State
Your
Instance
Session
Query
Unit of
Work
Connection

Main Extension Points

- MapperExtension
  - Instance construction
    - SQL CRUD- INSERT SELECT UPDATE...
- SessionExtension
  - Unit-of-work lifecycle and transaction

More Extension Points

- Relation `collection_class`
  - sets, dicts, custom collections
- MapperProperties and Comparators
  - Composite types
  - User.email == ‘x’
- Session and Query subclasses

Risky Extension Points

DANGER

- Instrumentation: InstrumentedAttributes, class and instance managers
- AttributeExtension

Sessions

Sessions

```
>>> Session = scoped_session(sessionmaker())
>>> session = Session()
>>> session = sqlalchemy.orm.session.Session()
>>> session = sessionmaker()()
>>> session = create_session()
```

- There is only

`sqlalchemy.orm.session.Session`

```
def create_session(bind=None, **kwargs):
    kwargs.setdefault('autoflush', False)
    kwargs.setdefault('autocommit', True)
    kwargs.setdefault('expire_on_commit', False)
    return Session(bind=bind, **kwargs)
```

```
def sessionmaker(bind=None, **kwargs):
    def create_session():
        return Session(bind=bind, **kwargs)
    return create_session
```

```
def sessionmaker(bind=None, **kwargs):
    def create_session():
        return Session(bind=bind, **kwargs)
    return create_session

Session = sessionmaker()
session = Session()

session = sessionmaker()()
```

```
class scoped_session(object):
    def __init__(self, factory):
        self._factory = factory
        self._instance = None

    def __getattr__(self, attribute):
        if self._instance is None:
            self._instance = self._factory()
        return getattr(self._instance, attribute)

    def remove(self):
        self._instance = None


Session = scoped_session(sessionmaker())
```

Extend by Wrapping

```
def YourSession(*args, **kwargs):
    extensions = kwargs.setdefault('extension', [])
    extensions.append(MySessionExtension)
    session = Session(*args, **kwargs)
    session.info = {}
    return session
```

Extend by Subclassing

```
class YourSession(Session):
    def __init__(self, *args, **kwargs):
        extensions = kwargs.setdefault('extension', [])
        extensions.append(MySessionExtension)
        Session.__init__(self, *args, **kwargs)
        self.info = {}
```

Using

```
>>> sessionmaker(class_=YourSession)
```

SessionExtension

- Listen for Session lifecycle events
  - Transaction boundaries
  - Flush
- Like `PoolListener`, limited ability to change the course of events in-progress.

```python
class SessionExtension:

    def before_commit(self, session):
        ...

    def after_commit(self, session):
        ...

    def after_rollback(self, session):
        ...

    def before_flush(self, session, flush_context, instances):
        ...

    def after_flush(self, session, flush_context):
        ...

    def after_flush_postexec(self, session, flush_context):
        ...

    def after_begin(self, session, transaction, connection):
        ...

    def after_attach(self, session, instance):
        ...

    def after_bulk_update(self, session, query, query_context, result):
        ...

    def after_bulk_delete(self, session, query, query_context, result):
        ...
```

```python
class SessionExtension:

    def before_commit(self, session):
        ...

    def after_commit(self, session):
        ...

    def after_rollback(self, session):
        ...

    def before_flush(self, session, flush_context, instances):
        ...

    def after_flush(self, session, flush_context):
        ...

    def after_flush_postexec(self, session, flush_context):
        ...

    def after_begin(self, session, transaction, connection):
        ...

    def after_attach(self, session, instance):
        ...

    def after_bulk_update(self, session, query, query_context, result):
        ...

    def after_bulk_delete(self, session, query, query_context, result):
        ...
```

- Attribute history is available to SessionExtensions in certain phases

```
>>> from sqlalchemy.orm.attributes import get_history

>>> print get_history(user, 'email')
((), [u'jek@discorporate.us'], ())

>>> user.email = 'jek+spam@discorporate.us'
>>> print get_history(user, 'email')
(['jek+spam@discorporate.us'], (), [u'jek@discorporate.us'])
```

- Generally speaking, SessionExtensions can not have Session side-effects (excepting before_flush)

Session Defaults

- `autoflush=True`
  - Flush will be visited many, many times before `commit()`
  - If tracking changes to objects, expect to see the same object more than once

- `expire_on_commit=True`
  - Object state will be gone after `commit()`
  - Inspecting attributes in `after_commit` will raise an exception

Strategies

- Inspect `session.new` `.dirty` `.deleted` in `before_flush`
- Inspect in a mapper extension & communicate to session extension

Revisiting Column Defaults

- Extend Session with `.info`
- Code interacts with sessions
- Session extensions make information available to Connections during transactions

```python
from sqlalchemy.orm.session import Session


class CustomSession(Session):

    def __init__(self, **kw):
        extensions = kw.get('extension', [])
        extensions.append(ContextualDefaultPopulator())
        kw['extension'] = extensions
        super(CustomSession, self).__init__(**kw)
        self.info = {}
```

```python
from collections import defaultdict
from sqlalchemy.orm.interfaces import SessionExtension


class ContextualDefaultPopulator(SessionExtension):
    """Links Session-level info with low-level Connection info."""

    def __init__(self):
        self._connection_map = defaultdict(list)

    def after_begin(self, session, transaction, connection):
        self.register(session, connection)
        self._connection_map[id(session)].append(connection)

    def after_commit(self, session):
        for connection in self._connection_map[id(session)]:
            self.unregister(connection)
        del self._connection_map[id(session)]

    after_rollback = after_commit

    def register(self, session, connection):
        """Copy session.info data to connection.info."""
        if 'updated_by' in session.info:
            connection.info['updated_by'] = session.info['updated_by']

    def unregister(self, connection):
        """Remove data from connection.info"""
        if 'updated_by' in connection.info:
            del connection.info['updated_by']
```

```
>>> session_factory = sessionmaker(class_=CustomSession)
>>> session = session_factory()

>>> session.info['updated_by'] = 456
>>> record = Record('record 1')
>>> session.add(record)

>>> print 'updated_by', record.updated_by
updated_by None
>>> session.commit()
>>> print 'after commit: updated_by', record.updated_by
after commit: updated_by 456
```

```
class MapperExtension:

    def instrument_class(self, mapper, class_):
        ...
    def init_instance(self, mapper, class_, oldinit, instance, args, kwargs):
        ...
    def init_failed(self, mapper, class_, oldinit, instance, args, kwargs):
        ...
    def translate_row(self, mapper, context, row):
        ...
    def create_instance(self, mapper, selectcontext, row, class_):
        ...
    def append_result(self, mapper, selectcontext, row, instance, result, **flags):
        ...
    def populate_instance(self, mapper, selectcontext, row, instance, **flags):
        ...
    def reconstruct_instance(self, mapper, instance):
        ...
    def before_insert(self, mapper, connection, instance):
        ...
    def after_insert(self, mapper, connection, instance):
        ...
    def before_update(self, mapper, connection, instance):
        ...
    def after_update(self, mapper, connection, instance):
        ...
    def before_delete(self, mapper, connection, instance):
        ...
    def after_delete(self, mapper, connection, instance):
        ...
```

Case Study: auto-add

- You'd like instances to be added to the Session automatically at construction
- This is a difficult pattern to use in practice

Approach

- Use a MapperExtension to intercept object initialization
- Have access to the “current” session, for example via a scoped_session

```
from sqlalchemy.orm.interfaces import MapperExtension

class Location(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

class AutoAdd(MapperExtension):
    """Automatically add instances to *session*."""

    def init_instance(self, mapper, class_, oldinit,
                      instance, args, kwargs):
        session.add(instance)

    def init_failed(self, mapper, class_, oldinit,
                    instance, args, kwargs):
        session.expunge(instance)

mapper(Location, locations_table, extension=AutoAdd())

point = Location(1, 2)
assert point in session
```

Extending Queries

- Make a domain-specific query language

9,10

```
>>> class UserQuery(Query):
...     def with_email(self, email):
...         return self.filter_by(email=email)
```

Extending Queries

```
>>> sessionmaker(query_cls=MyQuery)
>>> class MySession(Session):
...     def query(self, *entities, **kwargs):
...         return MyQuery(entities, self,
...                        **kwargs)
```

Extending Queries

```
>>> MyClass.query = scoped_session.query_property(MyQuery)
```

Case Study: object managers

- Implement Django-style "object managers" to separate persistence code from business logic

Approach

- Give mapped classes a `.objects` responsible for taking instances in and out of persistent state (`.add`, `.delete`, `.query`)
- Allow extending a class's `.objects` with query DSL methods
- Associate `.objects` with a MapperExtension

```
class DatabaseManager(object):
    """A Django-like database manager."""

    _query_cls = None
    _query_passthrough = ['filter', 'filter_by', 'all', 'one', 'get']
    _session_passthrough = ['add', 'add_all', 'commit', 'delete', 'flush']

    def __init__(self, session=None):
        self.model = None
        self.session = session

    @property
    def query(self):
        """A Query of managed model class."""
        if self._query_cls:
            return self._query_cls(self.model, session=self.session())
        return self.session.query(self.model)

def bind(self, model, session):
    """Called to link the manager to the model and default session."""
    assert self.model is None
    self.model = model
    if self.session is None:
        self.session = session

def __getattr__(self, attribute):
    if attribute in self._query_passthrough:
        return getattr(self.query, attribute)
    if attribute in self._session_passthrough:
        return getattr(self.session, attribute)
    raise AttributeError(attribute)
```

```
from sqlalchemy.orm import EXT_CONTINUE


class DatabaseManagerExtension(MapperExtension):
    """Applies and binds DatabaseManagers to model classes."""

    def __init__(self, session, default_factory=DatabaseManager):
        self.session = session
        self.default_factory = default_factory

def instrument_class(self, mapper, cls):
    factory = getattr(cls, '_manager_factory',
                      self.default_factory)
    manager = factory()
    cls.objects = manager
    manager.bind(cls, self.session)
    return EXT_CONTINUE
```

```
class UserManager(DatabaseManager):
    _query_cls = UserQuery
    _query_passthrough = \
        DatabaseManager._query_passthrough + ['with_email']


class User(object):
    _manager_factory = UserManager

    def __init__(self, email=None):
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.email

mapper(User, users_table,
       extension=[DatabaseManagerExtension(session)])
```

```
>>> User.objects
<sliderepl.UserManager object at 0x1017ffd10>
>>> User.objects.all()
[<User u'jek@discorporate.us'>]


>>> user2 = User('sqlalchemy@googlegroups.com')
>>> User.objects.add(user2)
>>> User.objects.with_email
('sqlalchemy@googlegroups.com').first()
<User 'sqlalchemy@googlegroups.com'>
```

Questions?

Thank You!

jek@discorporate.us
jek on freenode