Object Relational Mappers

Friend or Foe?

"All problems in computer science can be solved by another level of indirection"

"(except for the problem of too many layers of indirection)"

–David Wheeler

What Is an ORM?

- Object-oriented persistence abstraction
- A.K.A…
- Classes representing tables, instances representing rows
- But can be so much more

ORM Table Definition

- Table == Class
- Row == Instance
- Column == Object Field
- Foreign Key == Relationship Accessor

SQLAlchemy

DBIx::Class

```
class Firm < ActiveRecord::Base
  has_many   :clients
  has_one    :account
  belongs_to :conglomerate
end
```

HIBERNATE

SQLAlchemy

Talk Sponsorships

- The American Federation of ORM Vendors
- Local SF ORM User’s Union
- Koch Brothers
- Salman bin Abdulaziz Al Saud

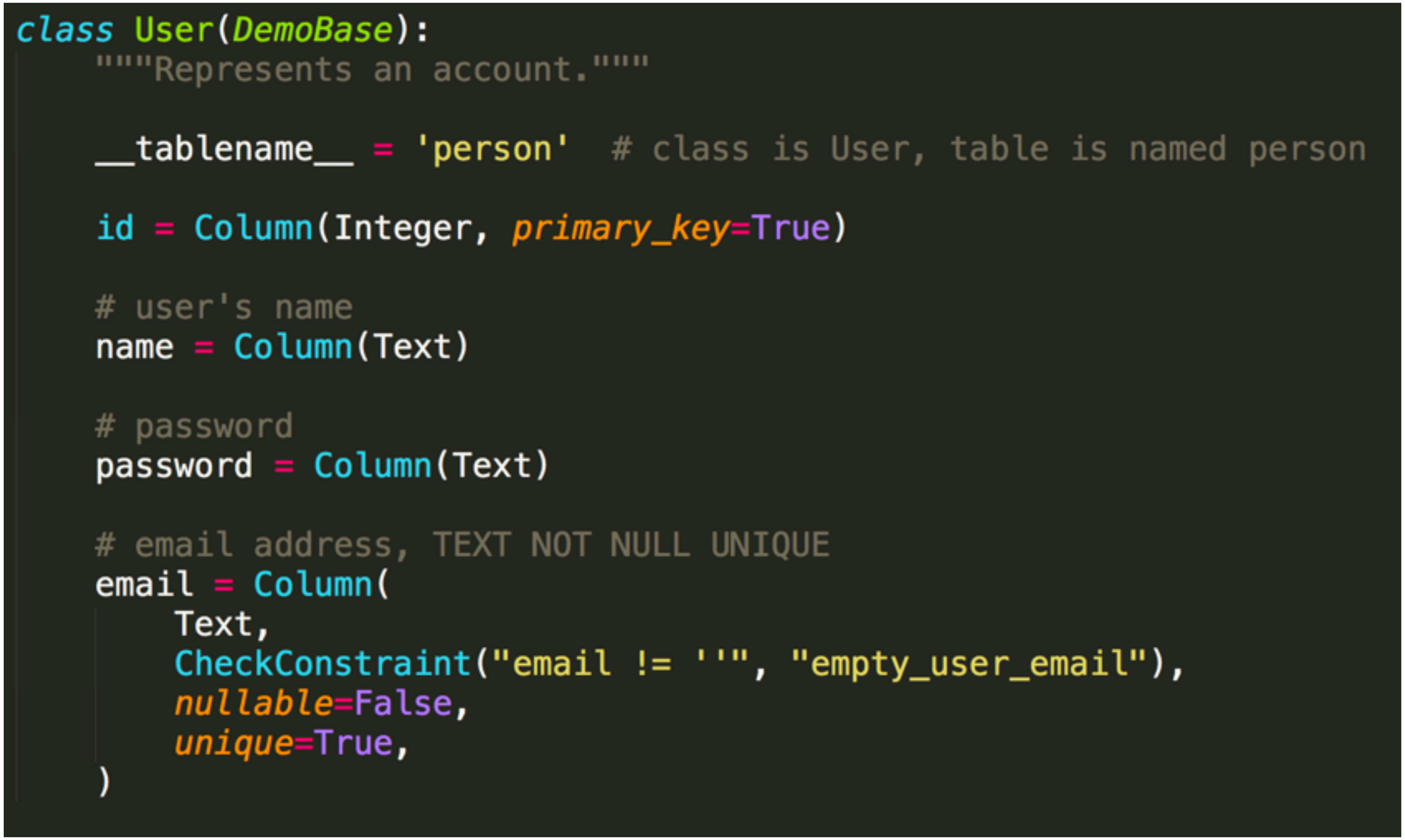
Why Use ORM?

- RDBMS abstraction
- Relationship mapping
- Data model with object methods
- Language-native, framework-native
- Mixins
- Polymorphism
- Introspection, API generation, doc generation
- Schema Migrations

Why Not Use ORM?

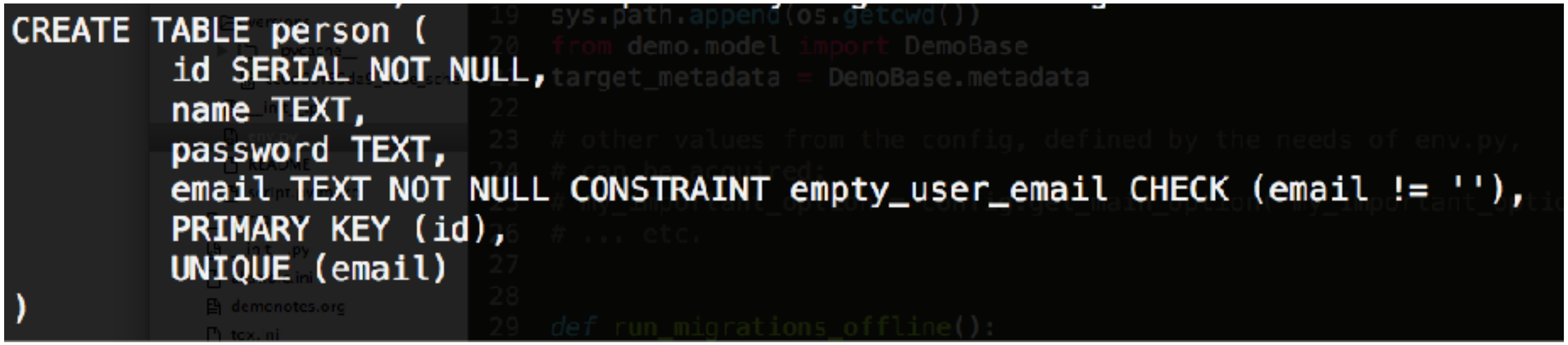
- Performance
  - (Easy to do unnecessary queries)
    - (But easy to fix)
- Learning curve and complexity
- Expression
- Difficult to use less generic database functionality

ORM Table Definition



```python
class User(DemoBase):
    """Represents an account."""

    __tablename__ = 'person'  # class is User, table is named person

    id = Column(Integer, primary_key=True)

    # user's name
    name = Column(Text)

    # password
    password = Column(Text)

    # email address, TEXT NOT NULL UNIQUE
    email = Column(
        Text,
        CheckConstraint("email != ''", "empty_user_email"),
        nullable=False,
        unique=True,
    )
```

ORM Table Definition



```
CREATE TABLE person (
        id SERIAL NOT NULL,
        name TEXT,
        password TEXT,
        email TEXT NOT NULL CONSTRAINT empty_user_email CHECK (email != ''),
        PRIMARY KEY (id),
        UNIQUE (email)
)
```

ORM Definitions: Reflect

```python
from sqlalchemy import create_engine
from sqlalchemy import MetaData
from sqlalchemy import Table

db_uri = 'sqlite:///db.sqlite'
engine = create_engine(db_uri)

# Create a MetaData instance
metadata = MetaData()
print metadata.tables

# reflect db schema to MetaData
metadata.reflect(bind=engine)
print metadata.tables
```

Basics Demo

- Connect
- Deploy schema (DDL)
- Select 1+1 scalar value
- Create user
- Update user
- Delete user

Relationship Mapping

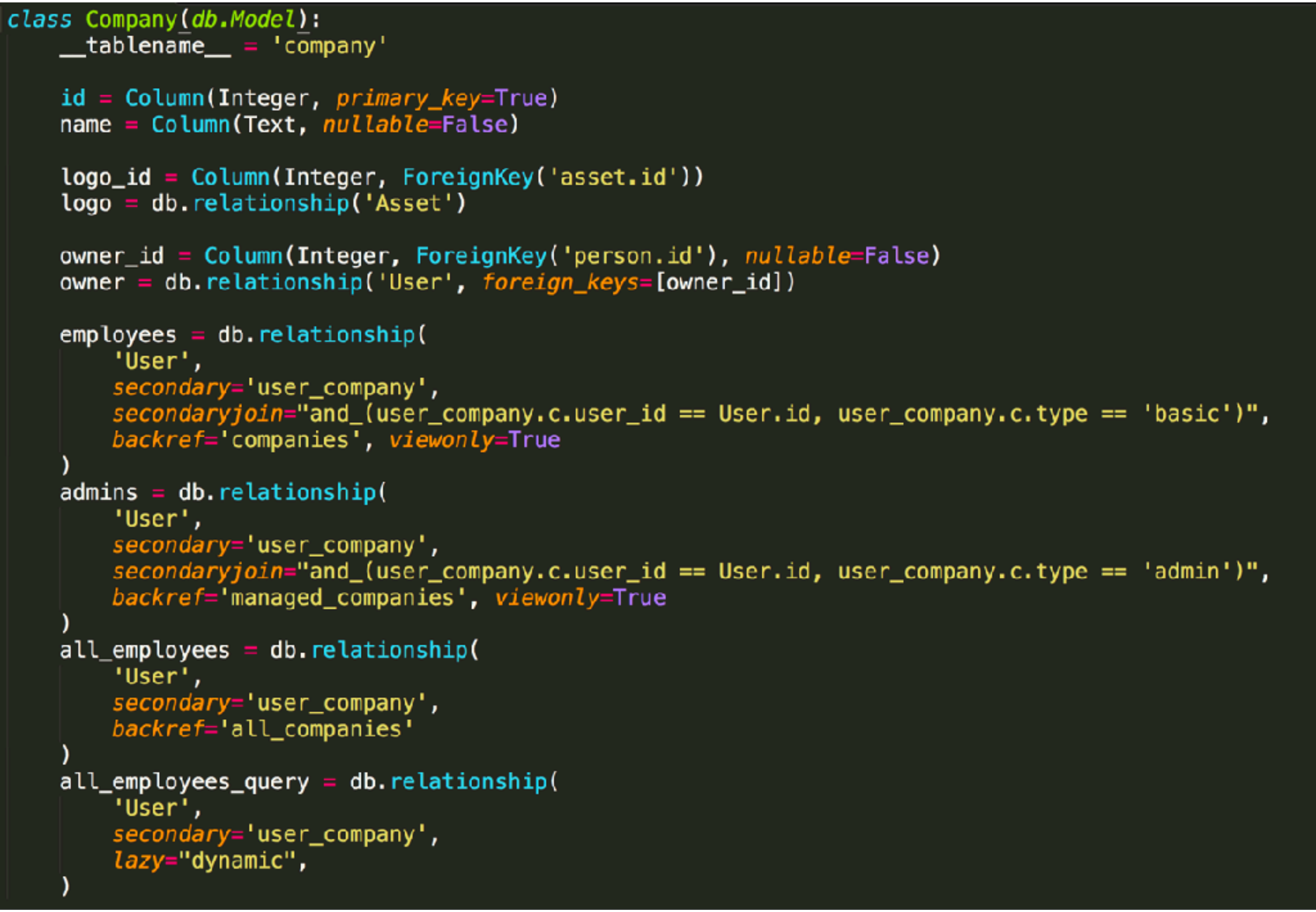
- Auto-join tables
- Define 1-1, 1-N, N-1, N-N relationships
- Relationship accessor
  - Scalar (1-1, N-1)
  - List (1-N, N-N)
  - “dynamic” query object

Relationship Mapping

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="children")
```

Relationship Mapping

- Easy joins
- Complex less-easy joins
- Many-to-many support
- Rel accessor can be “list” or “dynamic queries”

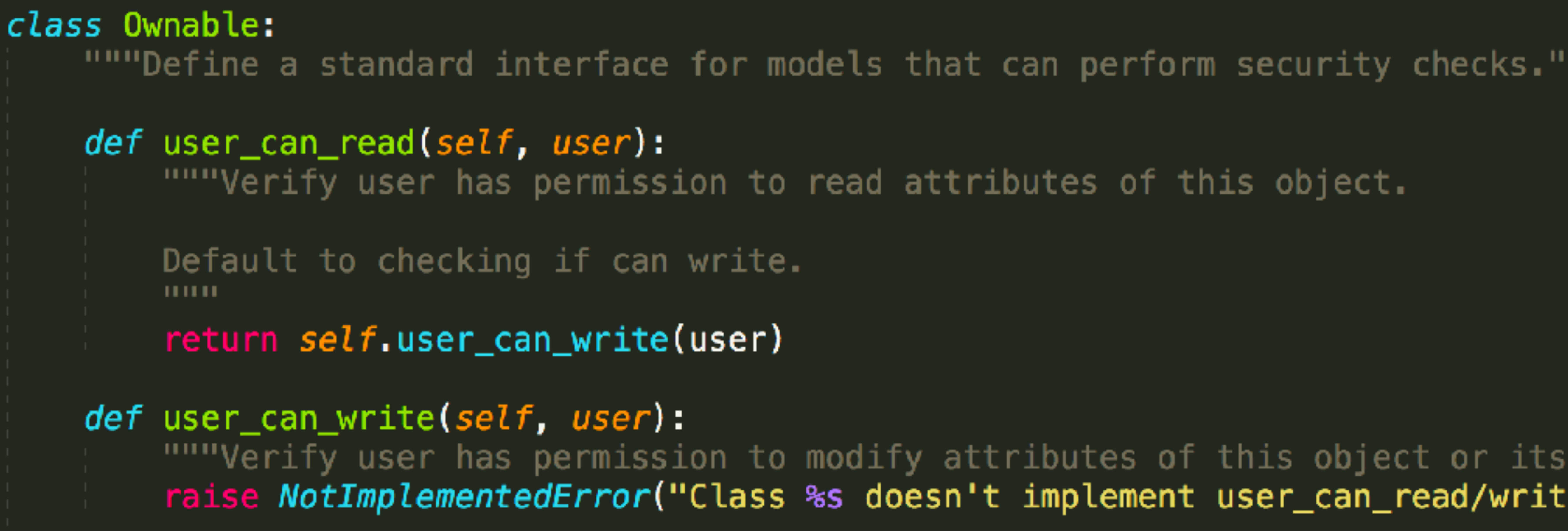


```python
class Company(db.Model):
    __tablename__ = 'company'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)

    logo_id = Column(Integer, ForeignKey('asset.id'))
    logo = db.relationship('Asset')

    owner_id = Column(Integer, ForeignKey('person.id'), nullable=False)
    owner = db.relationship('User', foreign_keys=[owner_id])

    employees = db.relationship(
        'User',
        secondary='user_company',
        secondaryjoin="and_(user_company.c.user_id == User.id, user_company.c.type == 'basic')",
        backref='companies', viewonly=True
    )
    admins = db.relationship(
        'User',
        secondary='user_company',
        secondaryjoin="and_(user_company.c.user_id == User.id, user_company.c.type == 'admin')",
        backref='managed_companies', viewonly=True
    )
    all_employees = db.relationship(
        'User',
        secondary='user_company',
        backref='all_companies'
    )
    all_employees_query = db.relationship(
        'User',
        secondary='user_company',
        lazy="dynamic",
    )
```

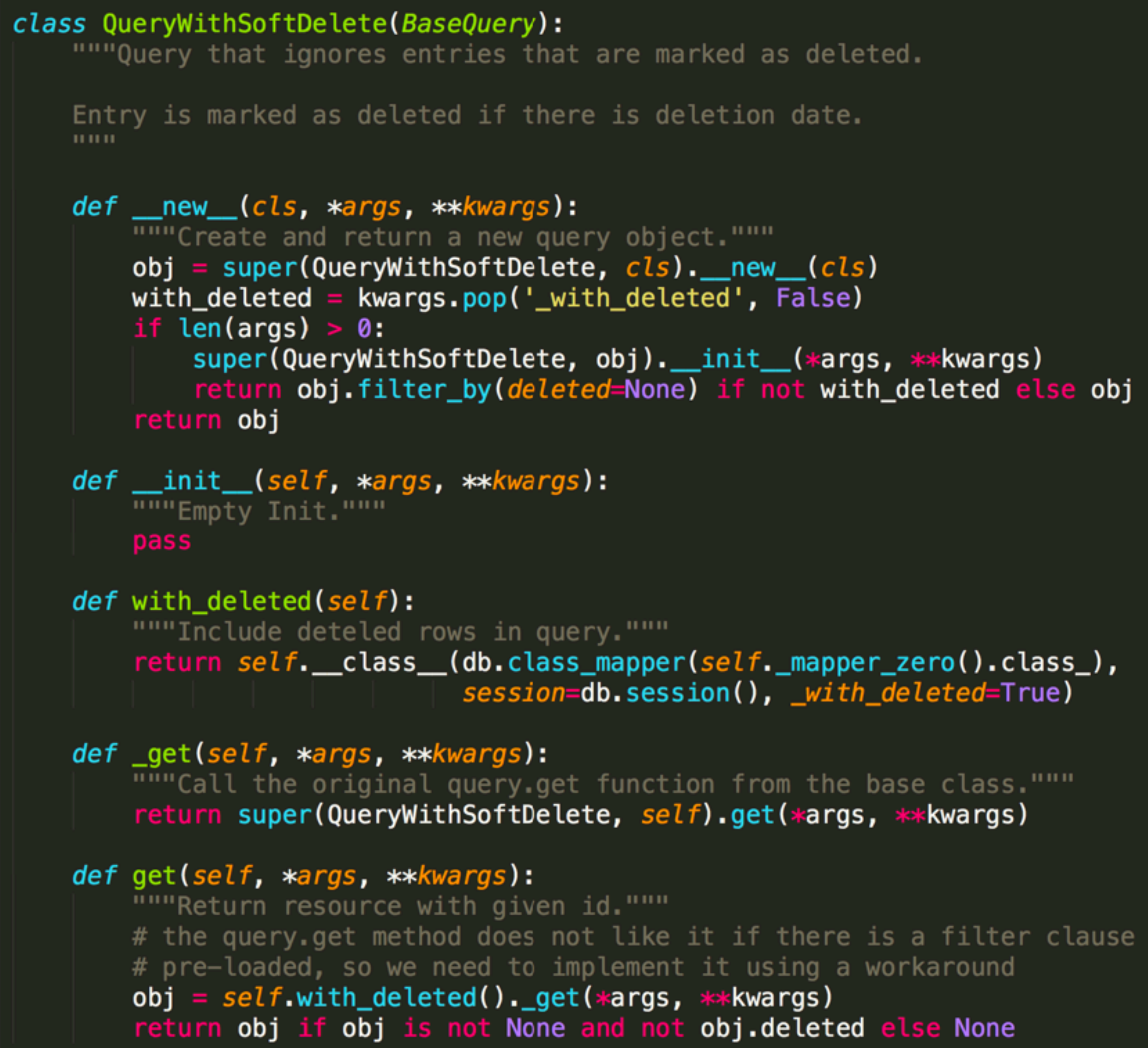
Mixins

```
class SoftDeleteable:
    """Mark items as deleted instead of deleting rows."""

    query_class = QueryWithSoftDelete
    deleted = Column(TSTZ, nullable=True)
```

Mixins



```
class Ownable:
    """Define a standard interface for models that can perform security checks."

    def user_can_read(self, user):
        """Verify user has permission to read attributes of this object.

        Default to checking if can write.
        """
        return self.user_can_write(user)

    def user_can_write(self, user):
        """Verify user has permission to modify attributes of this object or its
        raise NotImplementedError("Class %s doesn't implement user_can_read/writ
```



```python
class QueryWithSoftDelete(BaseQuery):
    """Query that ignores entries that are marked as deleted.

    Entry is marked as deleted if there is deletion date.
    """

    def __new__(cls, *args, **kwargs):
        """Create and return a new query object."""
        obj = super(QueryWithSoftDelete, cls).__new__(cls)
        with_deleted = kwargs.pop('_with_deleted', False)
        if len(args) > 0:
            super(QueryWithSoftDelete, obj).__init__(*args, **kwargs)
            return obj.filter_by(deleted=None) if not with_deleted else obj
        return obj

    def __init__(self, *args, **kwargs):
        """Empty Init."""
        pass

    def with_deleted(self):
        """Include deteled rows in query."""
        return self.__class__(db.class_mapper(self._mapper_zero().class_),
                              session=db.session(), _with_deleted=True)

    def _get(self, *args, **kwargs):
        """Call the original query.get function from the base class."""
        return super(QueryWithSoftDelete, self).get(*args, **kwargs)

    def get(self, *args, **kwargs):
        """Return resource with given id."""
        # the query.get method does not like it if there is a filter clause
        # pre-loaded, so we need to implement it using a workaround
        obj = self.with_deleted()._get(*args, **kwargs)
        return obj if obj is not None and not obj.deleted else None
```

ORM Philosophy

- Decouple model from views

Performance

- Not *inherently* less performant vs. raw SQL
- Easier to emit extra queries, fetching related rows
- Object instantiation overhead

Performance

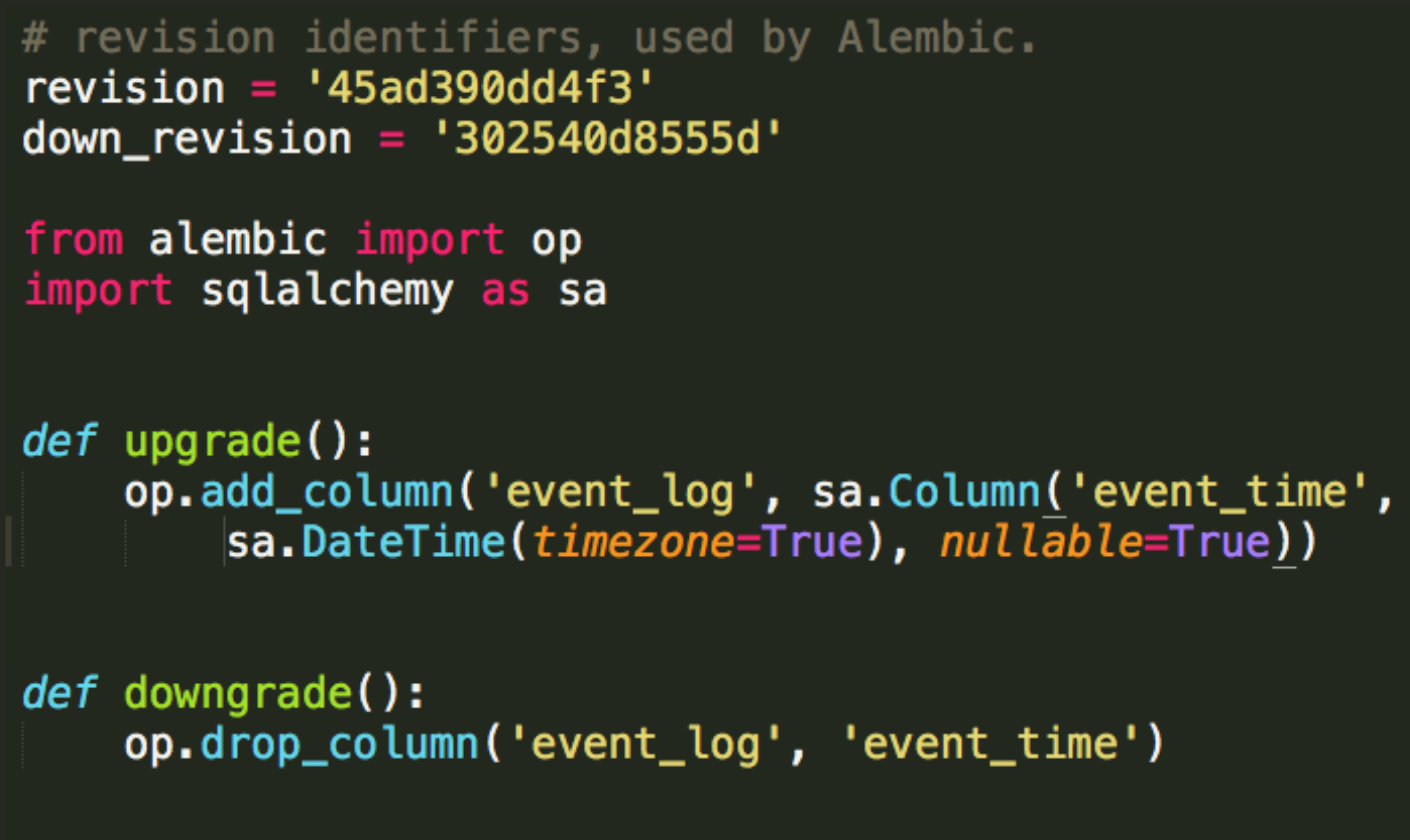
- Not always clear when queries get issued
- Demo

Polymorphism (SQLAlchemy)

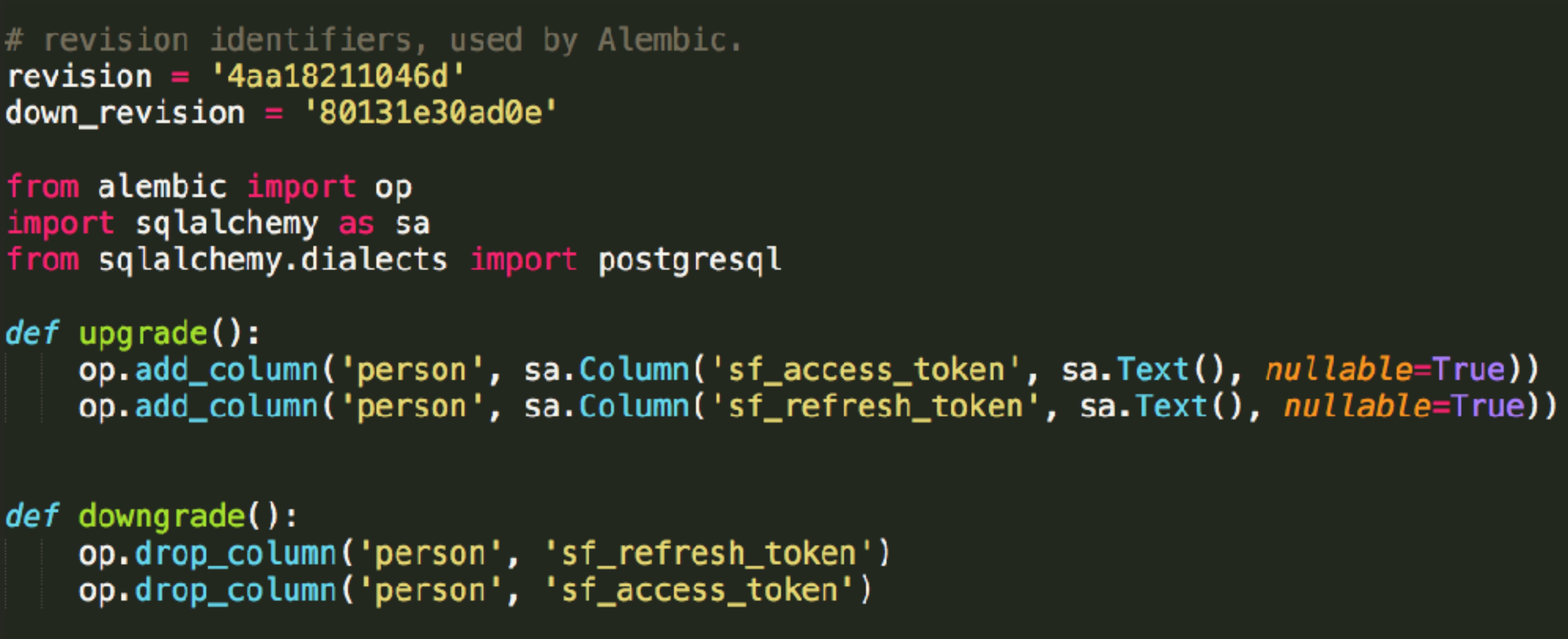
- Column discriminators
- Single-table
- Multi-table
- Etc

Schema Migration

- Transactional, atomic DDL changes
- Git-style forking/merges
- DDL -> SQL text file



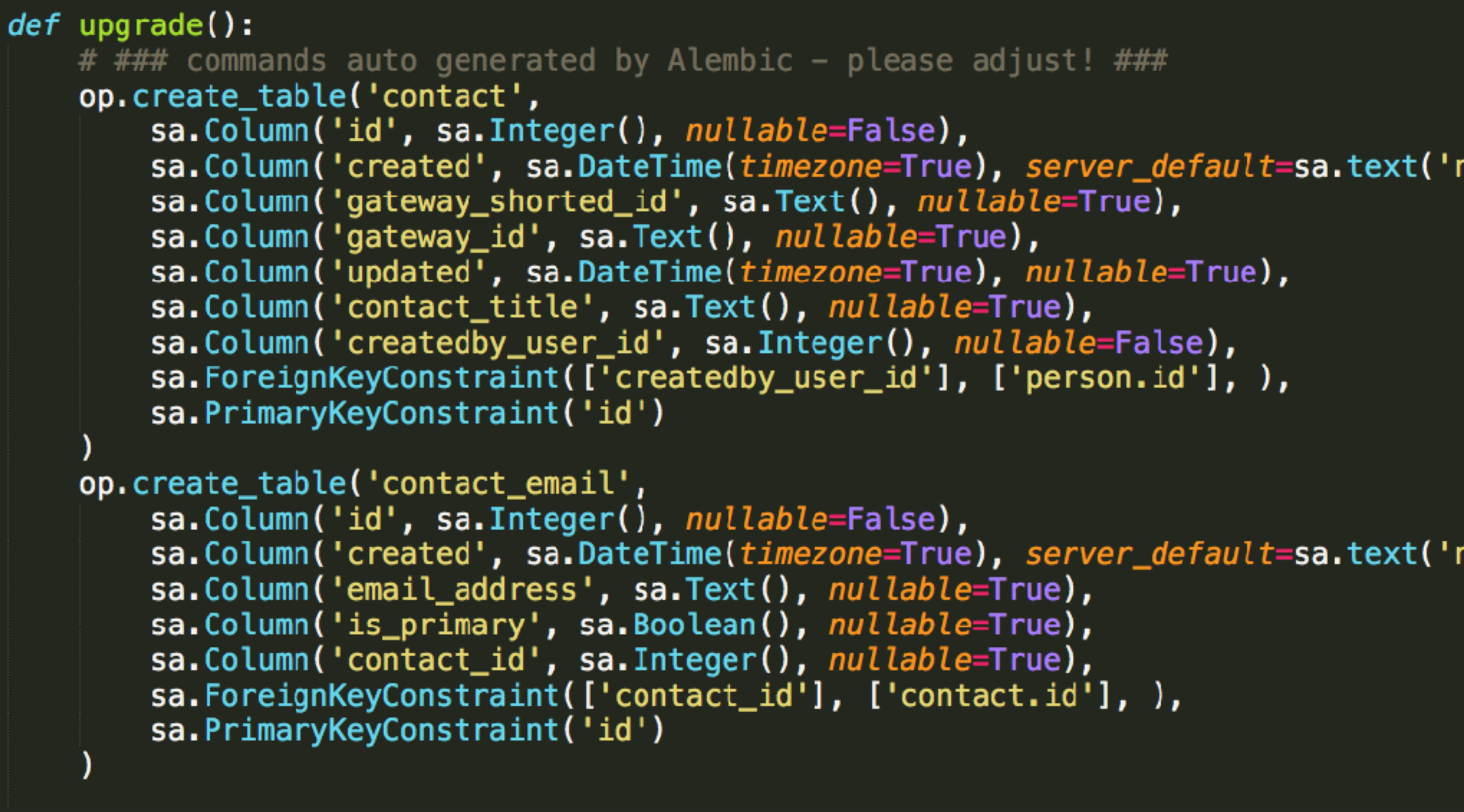
```
# revision identifiers, used by Alembic.
revision = '45ad390dd4f3'
down_revision = '302540d8555d'

from alembic import op
import sqlalchemy as sa


def upgrade():
    op.add_column('event_log', sa.Column('event_time',
        sa.DateTime(timezone=True), nullable=True))


def downgrade():
    op.drop_column('event_log', 'event_time')
```



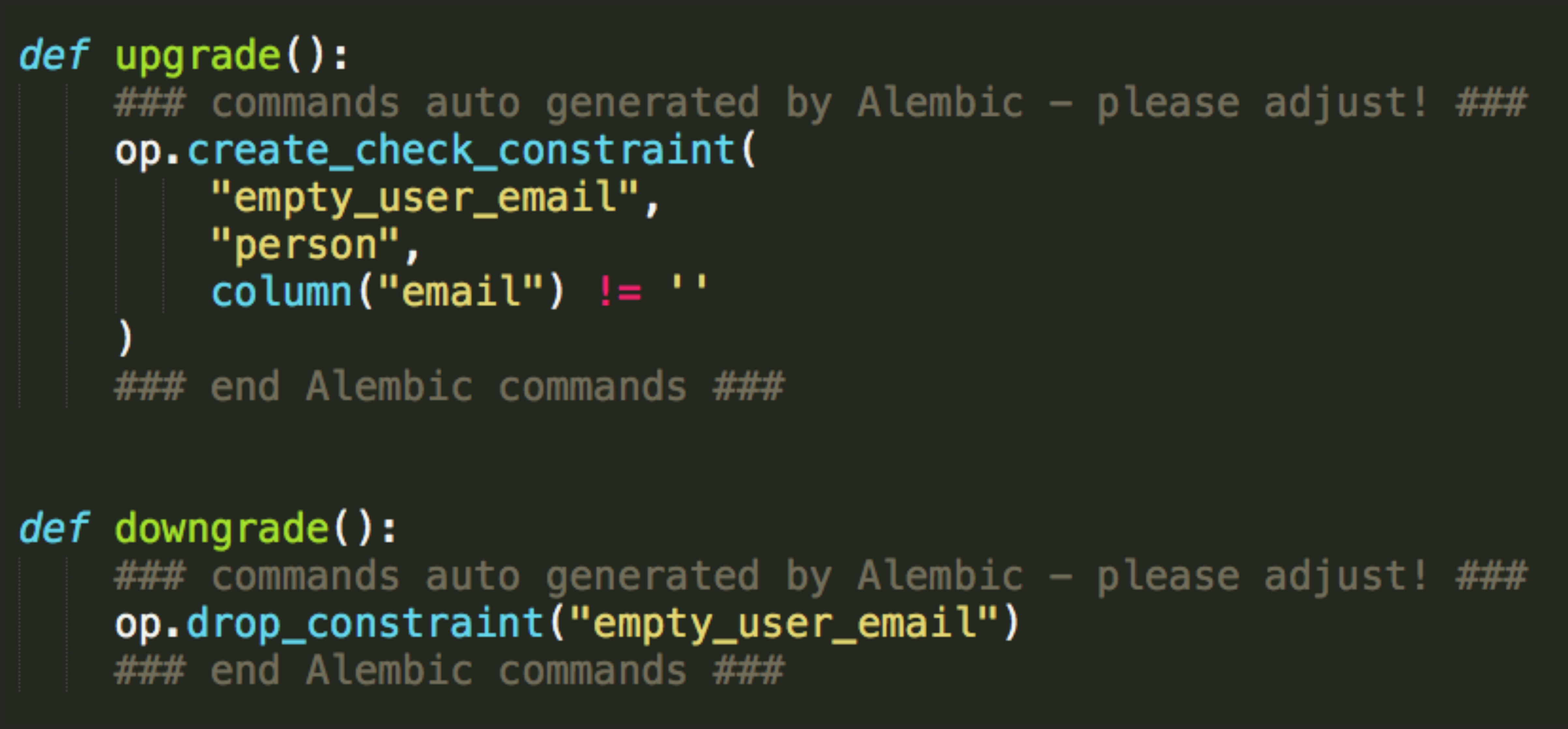
```python
# revision identifiers, used by Alembic.
revision = '4aa18211046d'
down_revision = '80131e30ad0e'

from alembic import op
import sqlalchemy as sa
from sqlalchemy.dialects import postgresql

def upgrade():
    op.add_column('person', sa.Column('sf_access_token', sa.Text(), nullable=True))
    op.add_column('person', sa.Column('sf_refresh_token', sa.Text(), nullable=True))


def downgrade():
    op.drop_column('person', 'sf_refresh_token')
    op.drop_column('person', 'sf_access_token')
```

```python
def upgrade():
    connection = op.get_bind()
    connection.execute("CREATE EXTENSION postgis")


def downgrade():
    connection = op.get_bind()
    connection.execute("DROP EXTENSION postgis")
```

```python
def upgrade():
    op.create_index('quiz_attempt_quiz_user_idx', 'quiz_attempt', ['quiz_id', 'user_id'])


def downgrade():
    op.drop_index('quiz_attempt_quiz_user_idx')
```



```python
def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('contact',
        sa.Column('id', sa.Integer(), nullable=False),
        sa.Column('created', sa.DateTime(timezone=True), server_default=sa.text('
        sa.Column('gateway_shorted_id', sa.Text(), nullable=True),
        sa.Column('gateway_id', sa.Text(), nullable=True),
        sa.Column('updated', sa.DateTime(timezone=True), nullable=True),
        sa.Column('contact_title', sa.Text(), nullable=True),
        sa.Column('createdby_user_id', sa.Integer(), nullable=False),
        sa.ForeignKeyConstraint(['createdby_user_id'], ['person.id'], ),
        sa.PrimaryKeyConstraint('id')
    )
    op.create_table('contact_email',
        sa.Column('id', sa.Integer(), nullable=False),
        sa.Column('created', sa.DateTime(timezone=True), server_default=sa.text('
        sa.Column('email_address', sa.Text(), nullable=True),
        sa.Column('is_primary', sa.Boolean(), nullable=True),
        sa.Column('contact_id', sa.Integer(), nullable=True),
        sa.ForeignKeyConstraint(['contact_id'], ['contact.id'], ),
        sa.PrimaryKeyConstraint('id')
    )
```



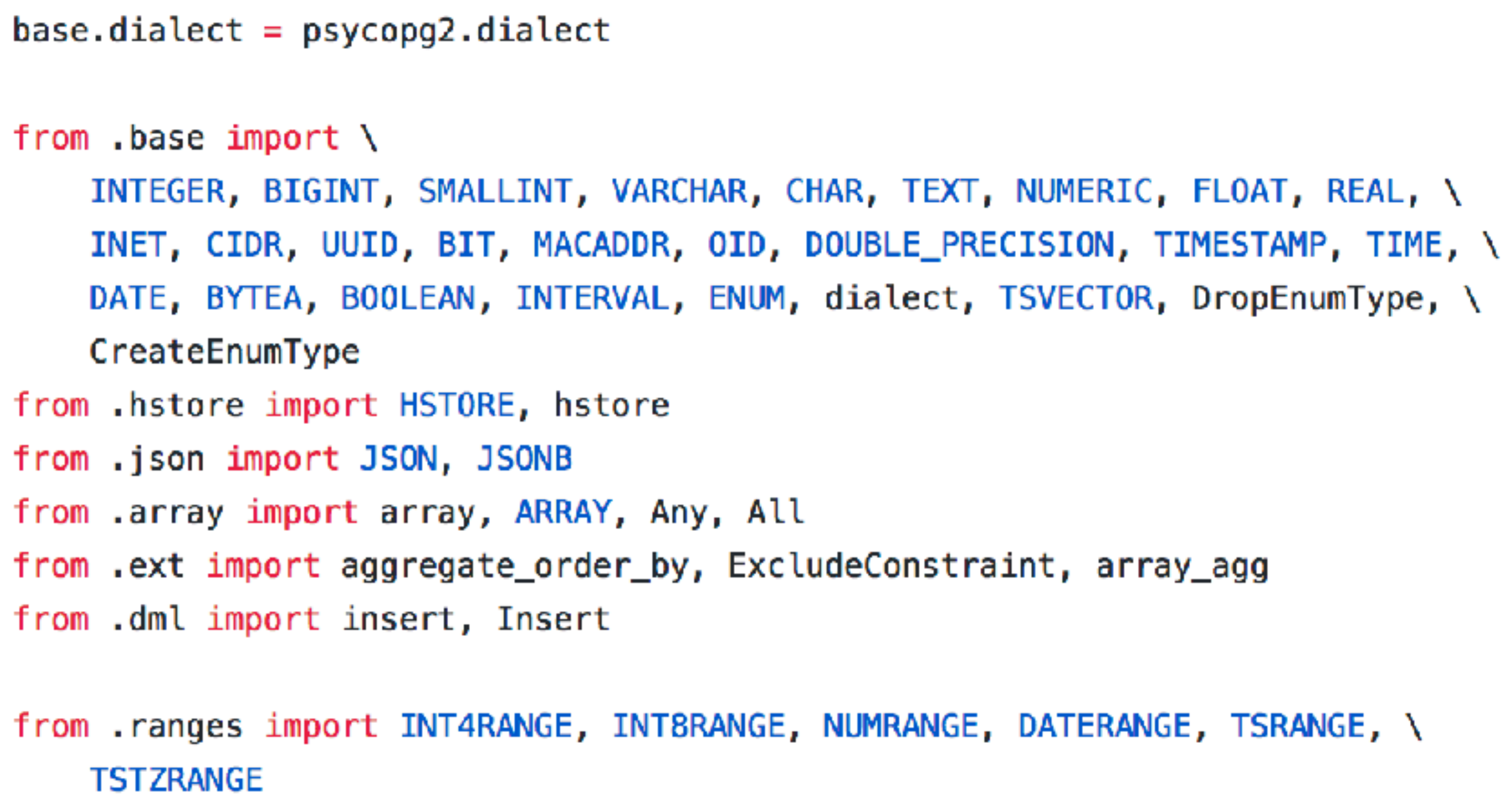
```python
def upgrade():
    ### commands auto generated by Alembic - please adjust! ###
    op.create_check_constraint(
        "empty_user_email",
        "person",
        column("email") != ''
    )
    ### end Alembic commands ###


def downgrade():
    ### commands auto generated by Alembic - please adjust! ###
    op.drop_constraint("empty_user_email")
    ### end Alembic commands ###
```

High-Level Reflection

- API generation
- API doc generation (Swagger etc)

MySQL-specific

PostgreSQL-specific



```
base.dialect = psycopg2.dialect

from .base import \
    INTEGER, BIGINT, SMALLINT, VARCHAR, CHAR, TEXT, NUMERIC, FLOAT, REAL, \
    INET, CIDR, UUID, BIT, MACADDR, OID, DOUBLE_PRECISION, TIMESTAMP, TIME, \
    DATE, BYTEA, BOOLEAN, INTERVAL, ENUM, dialect, TSVECTOR, DropEnumType, \
    CreateEnumType
from .hstore import HSTORE, hstore
from .json import JSON, JSONB
from .array import array, ARRAY, Any, All
from .ext import aggregate_order_by, ExcludeConstraint, array_agg
from .dml import insert, Insert

from .ranges import INT4RANGE, INT8RANGE, NUMRANGE, DATERANGE, TSRANGE, \
    TSTZRANGE
```

https://github.com/zzzeek/sqlalchemy/blob/master/lib/sqlalchemy/dialects/postgresql/__init__.py

PostgreSQL-specific

- INSERT/UPDATE...RETURNING
- INSERT...ON CONFLICT (Upsert)
- Full Text Search (“@@“ operator)
- SELECT/UPDATE/DELETE FROM ONLY …
- Partial Indexes
- Indexes with CONCURRENTLY
- TABLESPACE
- ON COMMIT
- INHERITS

```
try:
    from uuid import UUID as _python_UUID
except ImportError:
    _python_UUID = None

from sqlalchemy.types import INTEGER, BIGINT, SMALLINT, VARCHAR, \
    CHAR, TEXT, FLOAT, NUMERIC, \
    DATE, BOOLEAN, REAL

RESERVED_WORDS = set(
    ["all", "analyse", "analyze", "and", "any", "array", "as", "asc",
     "asymmetric", "both", "case", "cast", "check", "collate", "column",
     "constraint", "create", "current_catalog", "current_date",
     "current_role", "current_time", "current_timestamp", "current_user",
     "default", "deferrable", "desc", "distinct", "do", "else", "end",
     "except", "false", "fetch", "for", "foreign", "from", "grant", "group",
     "having", "in", "initially", "intersect", "into", "leading", "limit",
     "localtime", "localtimestamp", "new", "not", "null", "of", "off",
     "offset", "old", "on", "only", "or", "order", "placing", "primary",
     "references", "returning", "select", "session_user", "some", "symmetric",
     "table", "then", "to", "trailing", "true", "union", "unique", "user",
     "using", "variadic", "when", "where", "window", "with", "authorization",
     "between", "binary", "cross", "current_schema", "freeze", "full",
     "ilike", "inner", "is", "isnull", "join", "left", "like", "natural",
     "notnull", "outer", "over", "overlaps", "right", "similar", "verbose"
     ])

_DECIMAL_TYPES = (1231, 1700)
_FLOAT_TYPES = (700, 701, 1021, 1022)
_INT_TYPES = (20, 21, 23, 26, 1005, 1007, 1016)
```

```
class PGDialect(default.DefaultDialect):
    name = 'postgresql'
    supports_alter = True
    max_identifier_length = 63
    supports_sane_rowcount = True

    supports_native_enum = True
    supports_native_boolean = True
    supports_smallserial = True

    supports_sequences = True
    sequences_optional = True
    preexecute_autoincrement_sequences = True
    postfetch_lastrowid = False

    supports_default_values = True
    supports_empty_insert = False
    supports_multivalues_insert = True
    default_paramstyle = 'pyformat'
    ischema_names = ischema_names
    colspecs = colspecs
```

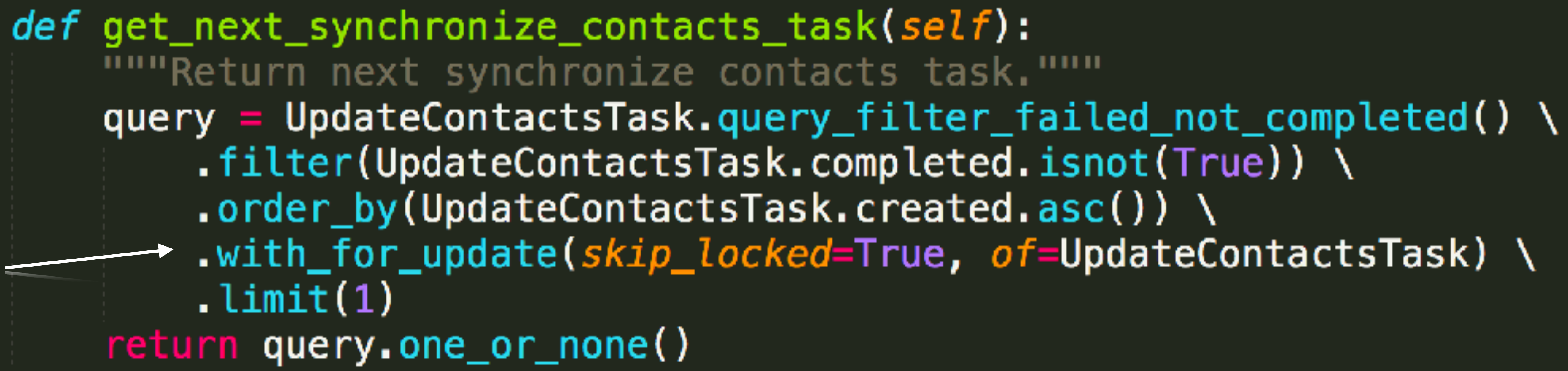
```
_DECIMAL_TYPES = (1231, 1700)
_FLOAT_TYPES = (700, 701, 1021, 1022)
_INT_TYPES = (20, 21, 23, 26, 1005, 1007,

class BYTEA(sqltypes.LargeBinary):
    __visit_name__ = 'BYTEA'


class DOUBLE_PRECISION(sqltypes.Float):
    __visit_name__ = 'DOUBLE_PRECISION'
```

```python
def on_connect(self):
    if self.isolation_level is not None:
        def connect(conn):
            self.set_isolation_level(conn, self.isolation_level)
        return connect
    else:
        return None

_isolation_lookup = set(['SERIALIZABLE', 'READ UNCOMMITTED',
                         'READ COMMITTED', 'REPEATABLE READ'])

def set_isolation_level(self, connection, level):
    level = level.replace('_', ' ')
    if level not in self._isolation_lookup:
        raise exc.ArgumentError(
            "Invalid value '%s' for isolation_level. "
            "Valid isolation levels for %s are %s" %
            (level, self.name, ", ".join(self._isolation_lookup))
        )
    cursor = connection.cursor()
    cursor.execute(
        "SET SESSION CHARACTERISTICS AS TRANSACTION "
        "ISOLATION LEVEL %s" % level)
    cursor.execute("COMMIT")
    cursor.close()
```

PostgreSQL-ish

- SELECT … FOR UPDATE
- SKIPPING LOCKED

PostgreSQL-ish

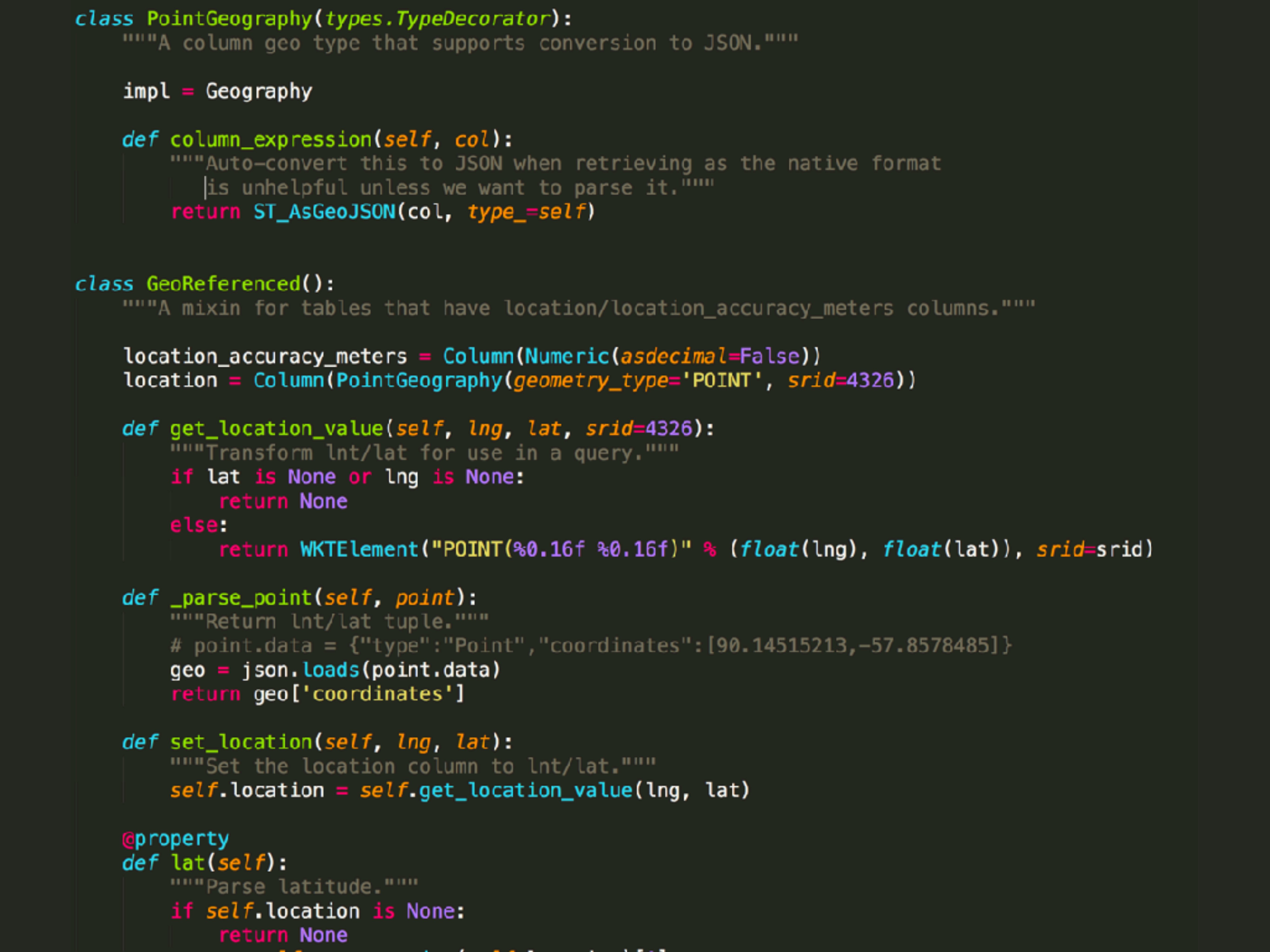


```python
def get_next_synchronize_contacts_task(self):
    """Return next synchronize contacts task."""
    query = UpdateContactsTask.query_filter_failed_not_completed() \
        .filter(UpdateContactsTask.completed.isnot(True)) \
        .order_by(UpdateContactsTask.created.asc()) \
        .with_for_update(skip_locked=True, of=UpdateContactsTask) \
        .limit(1)
    return query.one_or_none()
```

SELECT … FOR UPDATE SKIP LOCKED

PostGIS

```
from geoalchemy2.types import Geography
from geoalchemy2.functions import ST_AsGeoJSON
from geoalchemy2.elements import WKTElement
```



```python
class PointGeography(types.TypeDecorator):
    """A column geo type that supports conversion to JSON."""

    impl = Geography

    def column_expression(self, col):
        """Auto-convert this to JSON when retrieving as the native format
           is unhelpful unless we want to parse it."""
        return ST_AsGeoJSON(col, type_=self)


class GeoReferenced():
    """A mixin for tables that have location/location_accuracy_meters columns."""

    location_accuracy_meters = Column(Numeric(asdecimal=False))
    location = Column(PointGeography(geometry_type='POINT', srid=4326))

    def get_location_value(self, lng, lat, srid=4326):
        """Transform lnt/lat for use in a query."""
        if lat is None or lng is None:
            return None
        else:
            return WKTElement("POINT(%0.16f %0.16f)" % (float(lng), float(lat)), srid=srid)

    def _parse_point(self, point):
        """Return lnt/lat tuple."""
        # point.data = {"type":"Point","coordinates":[90.14515213,-57.8578485]}
        geo = json.loads(point.data)
        return geo['coordinates']

    def set_location(self, lng, lat):
        """Set the location column to lnt/lat."""
        self.location = self.get_location_value(lng, lat)

    @property
    def lat(self):
        """Parse latitude."""
        if self.location is None:
            return None
```

Reasons to use an ORM

- Write clean OO code
- Abstract database
- Simplify simple queries and relationships
- Migrations

Reasons to not use an ORM

- Learning curve
- Difficult to express some more complex or esoteric SQL constructs abstractly
- Sizeable dependency
- Small application

Fin