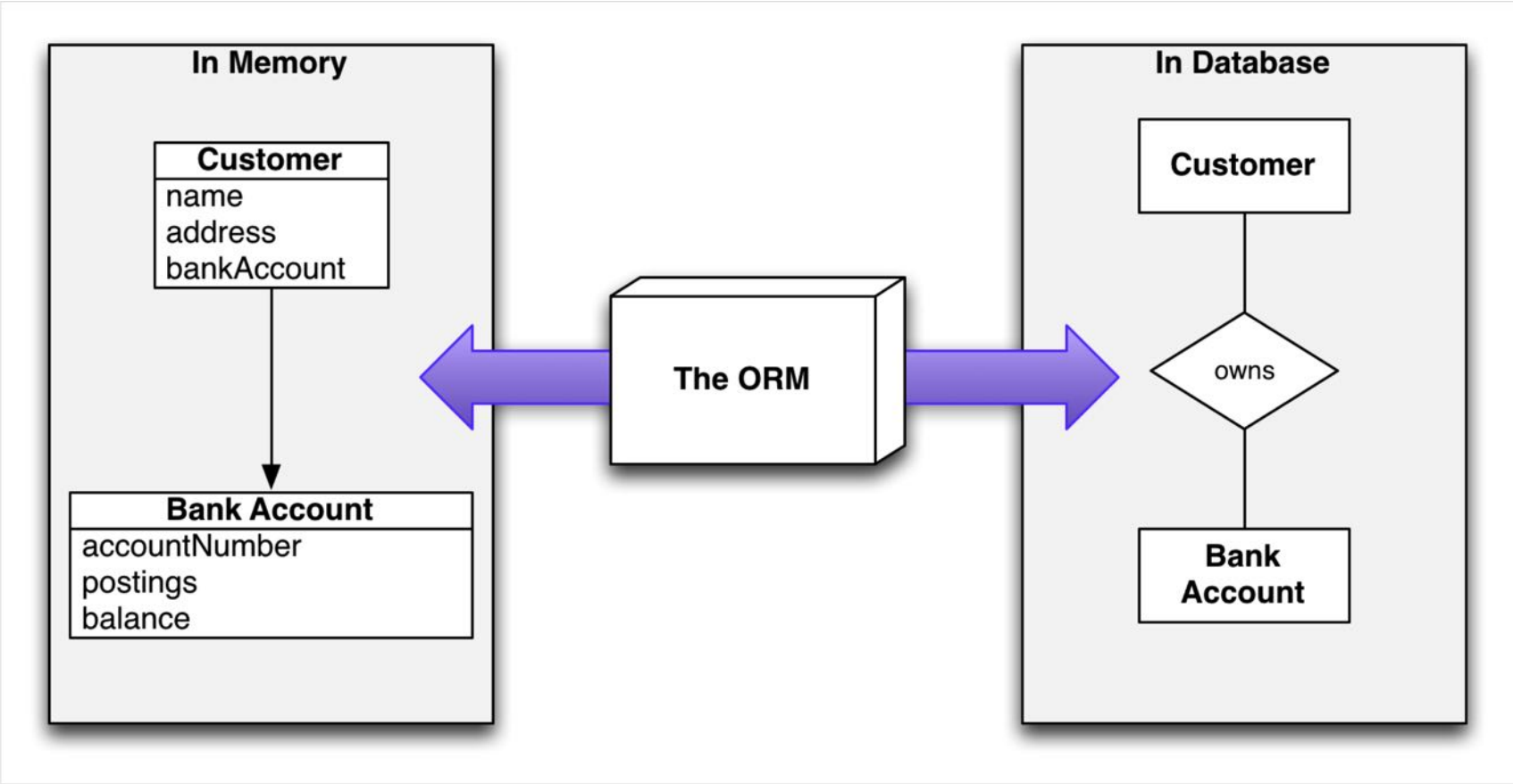
2. ORM, RE

2.1. ORM 소개



■ What is ORM?

○ ORM stands Object Relational Mapping

○ Programming technique for converting data between incompatible type systems using object-oriented programming languages

■ Why use ORM?

○ Mismatch between the object model and the relational database

- ➢ RDBSs represent data in tabular format
- ➢ Object-Oriented languages represent data as an interconnected graph of objects

○ ORM frees the programmer from dealing with simple repetitive database queries

○ Automatically mapping the database to business objects

○ Programmers focus more on business problems and less with data storage

○ The mapping process can aid in data verification and security before reaching the database

○ ORM can provide a caching layer above the database

■ Disadvantages

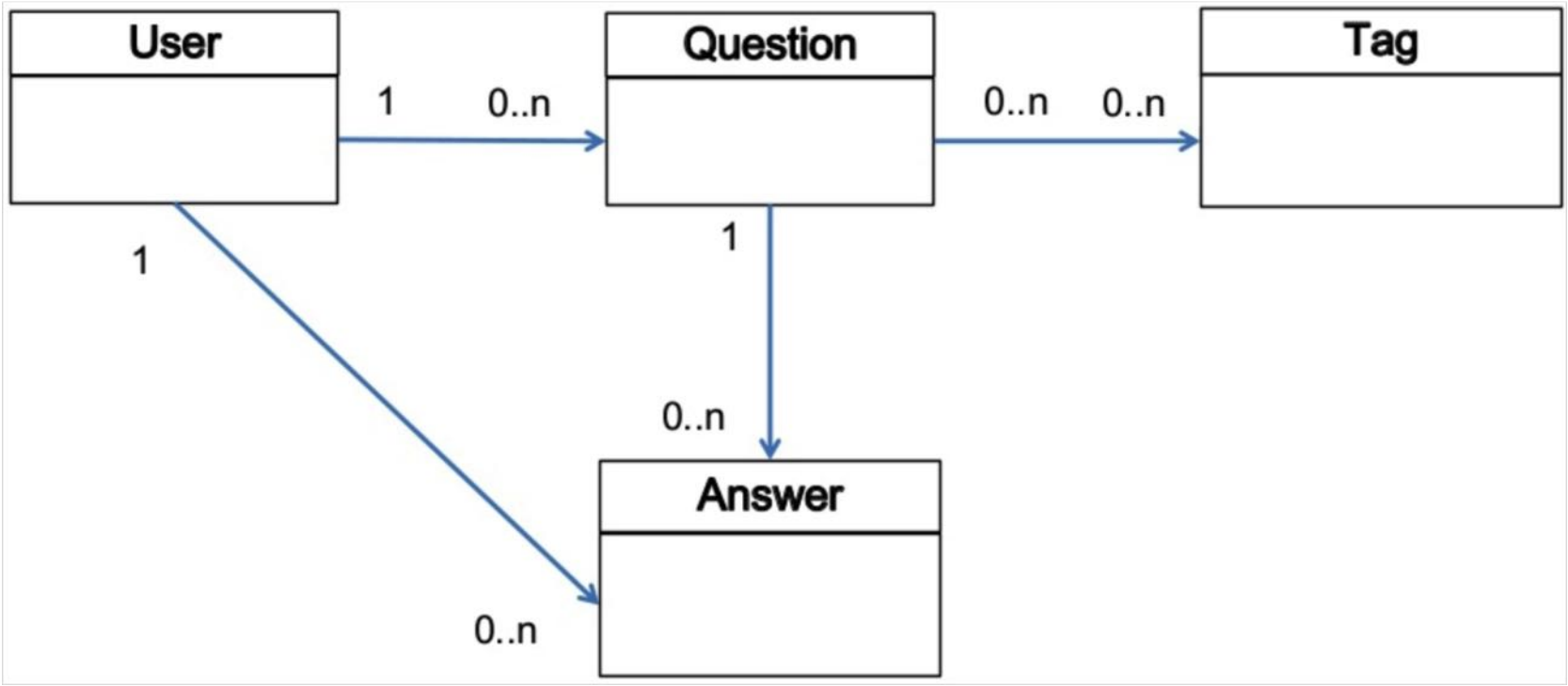
○ Potentially increasing processing overhead

■ 예제

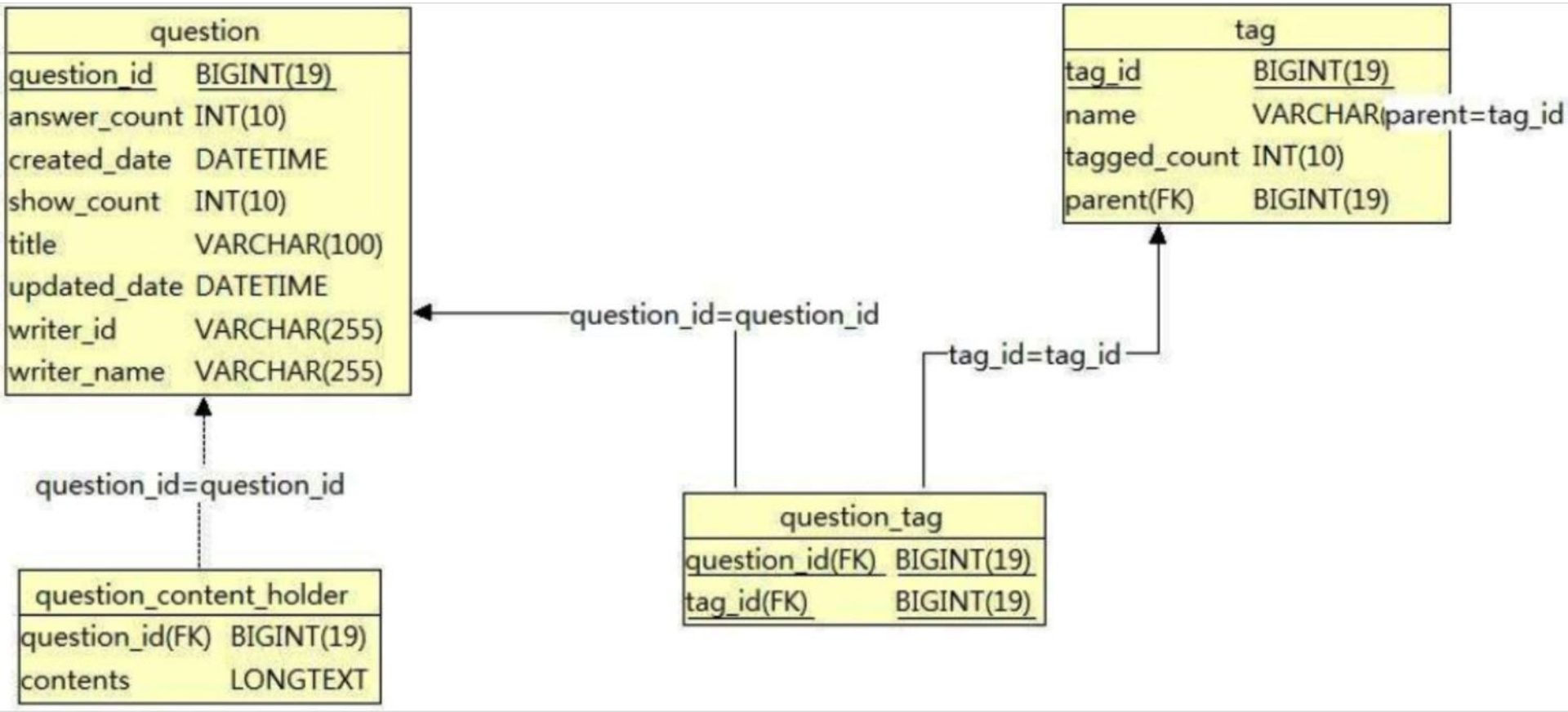
○ 요구사항

- 사용자는 질문 할 수 있어야 한다
- 질문에 대한 답변을 할 수 있어야 한다
- 질문할 때 태그를 추가할 수 있어야 한다
- 태그는 태그 풀에 존재하는 태그만 추가할 수 있다
- 태그가 추가될 경우 해당 태그 수는 +1 증가, 삭제될 경우 해당 태그 수는 -1 감소되어야 한다

○ 개체-관계 모델(ER Model)



○ 데이터베이스 설계


question
question_id BIGINT(19)
answer_count INT(10)
created_date DATETIME
show_count INT(10)
title VARCHAR(100)
updated_date DATETIME
writer_id VARCHAR(255)
writer_name VARCHAR(255)
tag
tag_id BIGINT(19)
name VARCHAR
tagged_count INT(10)
parent(FK) BIGINT(19)
parent=tag_id
question_id=question_id
tag_id=tag_id
question_id=question_id
question_tag
question_id(FK) BIGINT(19)
tag_id(FK) BIGINT(19)
question_content_holder
question_id(FK) BIGINT(19)
contents LONGTEXT

○ 질문 추가 시

- 질문 등록
- 태그 등록
  - 태그 풀에서 해당 태그 ID 가져오기
  - 태그 풀에서 해당 태그 수 증감하기

```
INSERT INTO question VALUES(?, ?, ?, ?, ?); → question_id = 1

SELECT tag_id, name FROM tag WHERE name="python"; → tag_id = 1
SELECT tag_id, name FROM tag WHERE name="alchemy"; → tag_id = 2

INSERT INTO question_tag VALUES(1, 1);
INSERT INTO question_tag VALUES(1, 2);

UPDATE tag SET tagged_count = tagged_count + 1 where name="python";
UPDATE tag SET tagged_count = tagged_count + 1 where name="alchemy";
```

○ 질문 수정 시

- 질문 수정
- 태그 수정
  - 태그 풀에서 해당 태그 ID 가져오기
  - 태그 풀에서 해당 태그 수 증감하기

```
UPDATE question SET title=?, contents=? WHERE question_id = 1;

SELECT tag_id, name FROM tag WHERE name="python"; → tag_id = 1
SELECT tag_id, name FROM tag WHERE name="alchemy"; → tag_id = 2
SELECT tag_id, name FROM tag WHERE name="orm"; → tag_id = 3

INSERT INTO question_tag VALUES(1, 3);
DELETE FROM question_tag where question_id=1 and tag_id=2;

UPDATE tag SET tagged_count = tagged_count - 1 where name="alchemy"
UPDATE tag SET tagged_count = tagged_count + 1 where name="orm"
```

*테이블 간의 관계보다 **데이터베이스 대한 처리**에 집중*

○ 객체-관계로 접근

```
public class Question {
    private long qid;
    private string title;
    [ … ]

    public processTags(string tag) {
    }
}
```

```
public class Tag {
    private long tid;
    private string tag;
    [ … ]

    public add(string tag) {
    }
}
```

INSERT INTO question VALUES(?, ?, ?, ?, ?); → question(1)
SELECT tag_id, name FROM tag WHERE name="python"; → tag.getByName("python")

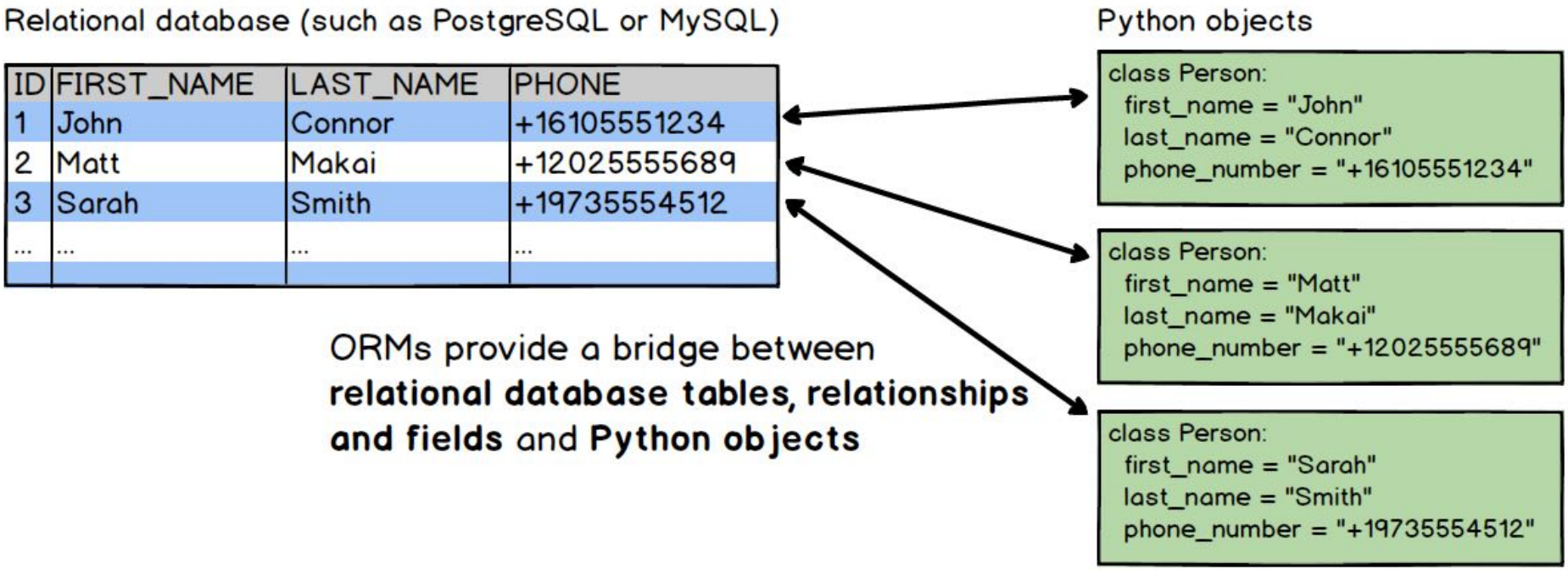
■ ORM in Python

○ ORM allows a developer to write Python code instead of SQL to create, read, update and delete

- ➢ Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures

○ ORMs make it theoretically possible to switch an application between various relational databases

- ➢ In practice however, it's best to use the same database for local development as is used in production



■ ORMs

○ Python ORM libraries are not required for accessing relational databases

- ➢ In fact, the low-level access is typically provided by another library called a *database connector*

web framework	None	Flask	Flask	Django
ORM	SQLAlchemy	SQLAlchemy	SQLAlchemy	Django ORM
database connector	(built into Python stdlib)	MySQL-python	psycopg	psycopg
relational database	SQLite	MySQL	PostgreSQL	PostgreSQL

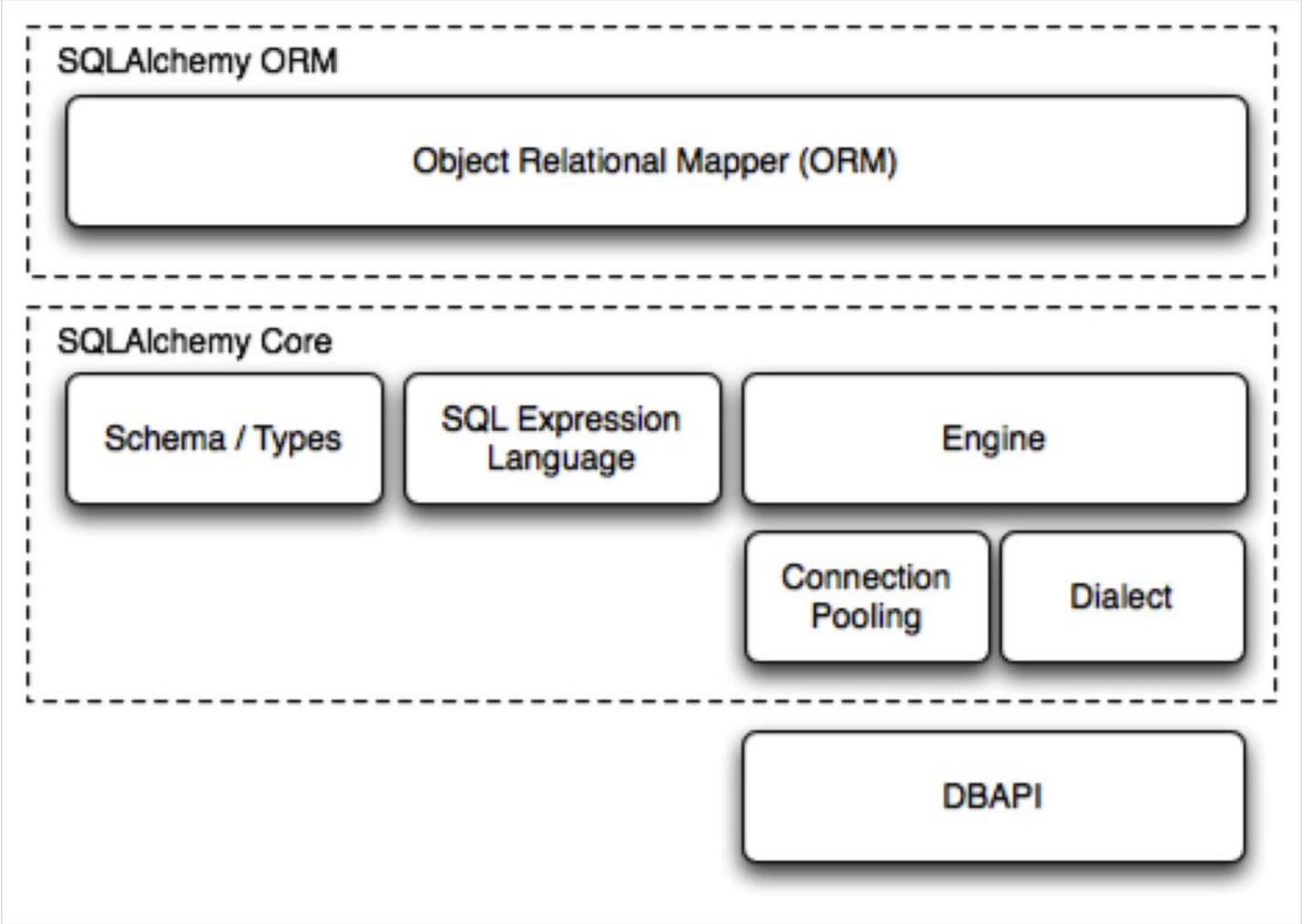
2.2. SQLAlchemy

■ What is SQLAlchemy?

○ SQLAlchemy is a well-regarded database toolkit and ORM implementation written in Python

○ SQLAlchemy provides a generalized interface for creating and executing database-agnostic code without needing to write SQL statements.

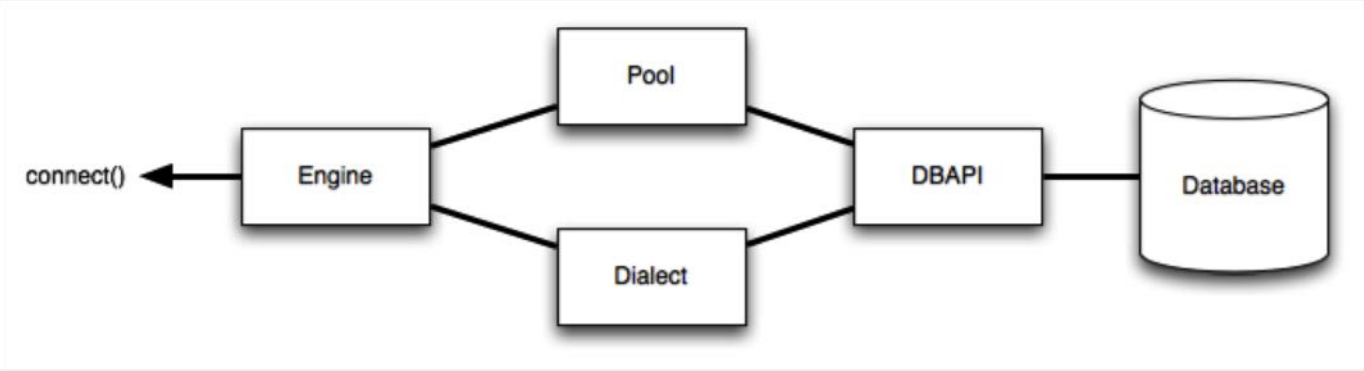
■ SQLAlchemy architecture



■ SQLAlchemy CORE

○ Engine

- ➢ starting point for any SQLAlchemy application
- ➢ a registry which provides connectivity to a particular database server



○ Dialect

- ➢ communicate with various types of DBAPI implementations and databases
- ➢ interprets generic SQL and database commands in terms of specific DBAPI and database backend
  - – Firebird, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, Sybase

○ Connection Pool

- ➢ holds a collection of database connections in memory for fast re-use

■ Connecting

○ create_engine

```
sqlalchemy.create_engine(*args, **kwargs)
```

```
dialect+driver://username:password@host:port/database
```

```
# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')
```

○ 예제

```
import sqlalchemy

sqlalchemy.__version__
```

```
from sqlalchemy import create_engine

engine = create_engine("sqlite://", echo=True)
#engine = create_engine("sqlite:///:memory:", echo=True)
#engine = create_engine("sqlite:///test.db", echo=True)

print(engine)
```

- Lazy connecting
- The echo flag is a shortcut to setting up SQLAlchemy logging

■ Create

○ Table

```
class sqlalchemy.schema.Table(*args, **kw)
```

- Table object constructs a unique instance of itself based on its name and optional schema name within the given MetaData object
- name, metadata, columns, constraint, …

○ Column

```
class sqlalchemy.schema.Column(*args, **kwargs)
```

- Represents a column in a database table
- name, type, constraint, authoincreament, default, nullable, …

○ MetaData

```
class sqlalchemy.schema.MetaData
```

- A collection of Table objects and their associated schema constructs
- Holds a collection of Table objects as well as an optional binding to an Engine or Connection. If bound, the Table objects in the collection and their columns may participate in implicit SQL execution
- Table objects themselves are stored in the MetaData.tables dictionary

○ 예제

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

metadata = MetaData()
users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('fullname', String),
)

addresses = Table('addresses', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_id', None, ForeignKey('users.id')),
    Column('email_address', String, nullable=False)
)

metadata.create_all(engine)
```

■ Insert

○ insert

insert(*values=None, inline=False, **kwargs*)

- Represents an INSERT construct
- generate an insert() construct against this TableClause

○ compile

compile(*bind=None, dialect=None, **kw*)

- Compile this SQL expression
- Return value is a Compiled object
- Compiled object also can return a dictionary of bind parameter names and values using the params accessor

○ 예제

```
insert = users.insert()
print(insert)

insert = users.insert().values(name='kim', fullname='Anonymous, Kim')
print(insert)

insert.compile().params
```

■ Executing

○ Connection

```
class sqlalchemy.engine.Connection
```

- Provides high-level functionality for a wrapped DB-API connection
- Provides execution support for string-based SQL statements as well as ClauseElement, Compiled and DefaultGenerator objects

○ execute

```
execute(object, *multiparams, **params)
```

- Executes a SQL statement construct and returns a ResultProxy

○ ResultProxy

```
class sqlalchemy.engine.ResultProxy(context)
```

- Wraps a DB-API cursor object to provide easier access to row columns

○ 예제

```
conn = engine.connect()
conn

insert.bind = engine
str(insert)

result = conn.execute(insert)

result.inserted_primary_key
```

➢ execute의 params 사용

```
insert = users.insert()

result = conn.execute(insert, name="lee", fullname="Unknown, Lee")

result.inserted_primary_key
```

➢ DBAPI의 executemany() 사용

```
conn.execute(addresses.insert(), [
    {"user_id":1, "email_address":"anonymous.kim@test.com"},
    {"user_id":2, "email_address":"unknown.lee@test.com"}
])
```

■ Select

○ select

```
sqlalchemy.sql.expression.select
```

- Construct a new Select
- columns, whereclause, from_obj, group_by, order_by, ...

○ 예제

```
from sqlalchemy.sql import select

query = select([users])
result = conn.execute(query)

for row in result:
    print(row)
```

```
result = conn.execute(select([users.c.name, users.c.fullname]))

for row in result:
    print(row)
```

■ ResultProxy

○ fetchone

```
fetchone()
```

➢ Fetch one row, just like DB-API cursor.fetchone()

○ fetchall

```
fetchall()
```

➢ Fetch all rows, just like DB-API cursor.fetchall()

○ 예제

```
result = conn.execute(query)

row = result.fetchone()
print("id -", row["id"], ", name -", row["name"], ", fullname -", row["fullname"])

row = result.fetchone()
print("id -", row[0], ", name -", row[1], ", fullname -", row[2])

result = conn.execute(query)
rows = result.fetchall()

for row in rows:
    print("id -", row[0], ", name -", row[1], ", fullname -", row[2])

result.close()
```

■ Conjunctions

○ 예제

```
from sqlalchemy import and_, or_, not_

print(users.c.id == addresses.c.user_id)

print(users.c.id == 1)

print((users.c.id == 1).compile().params)

print(or_(users.c.id == addresses.c.user_id, users.c.id == 1))

print(and_(users.c.id == addresses.c.user_id, users.c.id == 1))

print(and_(
        or_(
            users.c.id == addresses.c.user_id,
            users.c.id == 1
        ),
        addresses.c.email_address.like("a%")
    )
)

print((
        (users.c.id == addresses.c.user_id) |
        (users.c.id == 1)
     ) & (addresses.c.email_address.like("a%")))
```

■ Selecting

○ 예제

```
result = conn.execute(select([users]).where(users.c.id==1))
for row in result:
    print(row)

result = conn.execute(select([users, addresses]).where(users.c.id==addresses.c.user_id))
for row in result:
    print(row)

result = conn.execute(select([users.c.id, users.c.fullname, addresses.c.email_address])
                      .where(users.c.id==addresses.c.user_id))
for row in result:
    print(row)

result = conn.execute(select([users.c.id, users.c.fullname, addresses.c.email_address])
                      .where(users.c.id==addresses.c.user_id)
                      .where(addresses.c.email_address.like("un%")))
for row in result:
    print(row)
```

■ Join

○ join

```
join(right, onclause=None, isouter=False, full=False)
```

➢ Return a Join from this FromClause to another FromClause

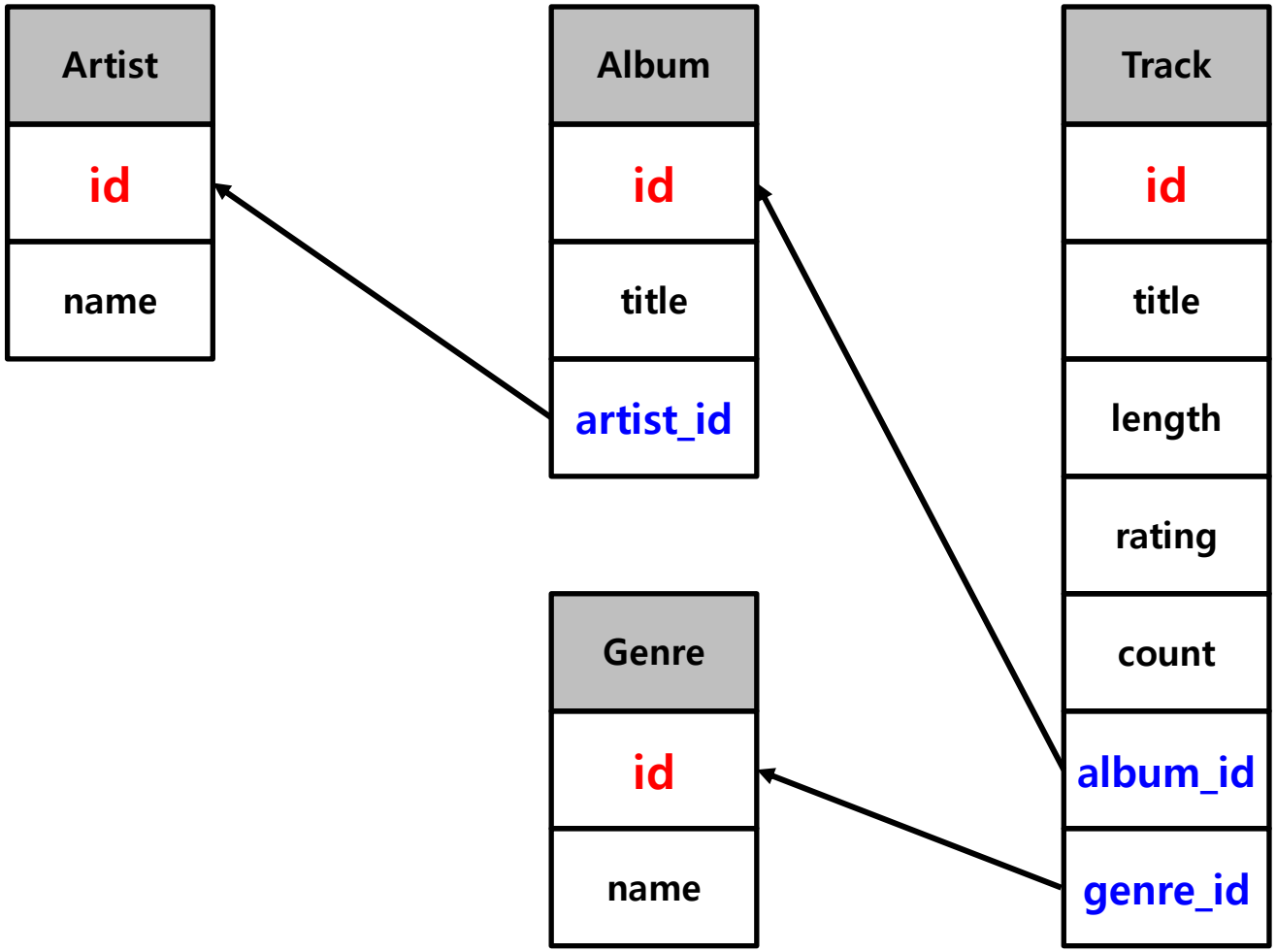
○ 예제

```
from sqlalchemy import join

print(users.join(addresses))

print(users.join(addresses, users.c.id == addresses.c.user_id))
```

➢ ON condition of the join, as it's called, was automatically generated based on the ForeignKey object

```
query = select([users.c.id, users.c.fullname, addresses.c.email_address]) \
        .select_from(users.join(addresses))

result = conn.execute(query).fetchall()
for row in result:
    print(row)
```

■ 예제


Artist
id
name
Album
id
title
artist_id
Track
id
title
length
rating
count
album_id
genre_id
Genre
id
name

○ Create

```
artist = Table("Artist", metadata,
               Column("id", Integer, primary_key=True),
               Column("name", String, nullable=False),
               extend_existing=True)

album = Table("Album", metadata,
              Column("id", Integer, primary_key=True),
              Column("title", String, nullable=False),
              Column("artist_id", Integer, ForeignKey("Artist.id")),
              extend_existing=True)

genre = Table("Genre", metadata,
              Column("id", Integer, primary_key=True),
              Column("name", String, nullable=False),
              extend_existing=True)

track = Table("Track", metadata,
              Column("id", Integer, primary_key=True),
              Column("title", String, nullable=False),
              Column("length", Integer, nullable=False),
              Column("rating", Integer, nullable=False),
              Column("count", Integer, nullable=False),
              Column("album_id", Integer, ForeignKey("Album.id")),
              Column("genre_id", Integer, ForeignKey("Genre.id")),
              extend_existing=True)

metadata.create_all(engine)
```

○ SHOW TABLES

```
tables = metadata.tables
for table in tables:
    print(table)

for table in engine.table_names():
    print(table)
```

○ Insert

```
conn.execute(artist.insert(), [
    {"name":"Led Zepplin"},
    {"name":"AC/DC"}
])

conn.execute(album.insert(), [
    {"title":"IV", "artist_id":1},
    {"title":"Who Made Who", "artist_id":2}
])

conn.execute(genre.insert(), [
    {"name":"Rock"},
    {"name":"Metal"}
])

conn.execute(track.insert(), [
    {"title":"Black Dog", "rating":5, "length":297, "count":0, "album_id":1, "genre_id":1},
    {"title":"Stairway", "rating":5, "length":482, "count":0, "album_id":1, "genre_id":1},
    {"title":"About to rock", "rating":5, "length":313, "count":0, "album_id":2, "genre_id":2},
    {"title":"Who Made Who", "rating":5, "length":297, "count":0, "album_id":2, "genre_id":2}
])
```

○ Select

```
artistResult = conn.execute(artist.select())
for row in artistResult:
    print(row)

albumResult = conn.execute(album.select())
for row in albumResult:
    print(row)

genreResult = conn.execute(genre.select())
for row in genreResult:
    print(row)

trackResult = conn.execute(track.select())
for row in trackResult:
    print(row)
```

○ Where

```
trackResult = conn.execute(select([track])
                           .where(
                               and_(track.c.album_id == 1, track.c.genre_id == 1)
                           )
                          )
for row in trackResult:
    print(row)
```

○ Update

```
from sqlalchemy import update

conn.execute(track.update().values(genre_id=2).where(track.c.id==2))
conn.execute(track.update().values(genre_id=1).where(track.c.id==3))
```

○ Where

```
trackResult = conn.execute(select([track])
                           .where(
                               and_(track.c.album_id == 1,
                                    or_(track.c.genre_id == 1,
                                        track.c.genre_id == 2,)
                                   )
                           )
                          )
for row in trackResult:
    print(row)
```

○ Join

```
print(track.join(album))

result = conn.execute(track
                      .select()
                      .select_from(track.join(album)))

for row in result.fetchall():
    print(row)

result = conn.execute(track
                      .select()
                      .select_from(track.join(album))
                      .where(album.c.id==1))

for row in result.fetchall():
    print(row)
```

○ Multiple Join

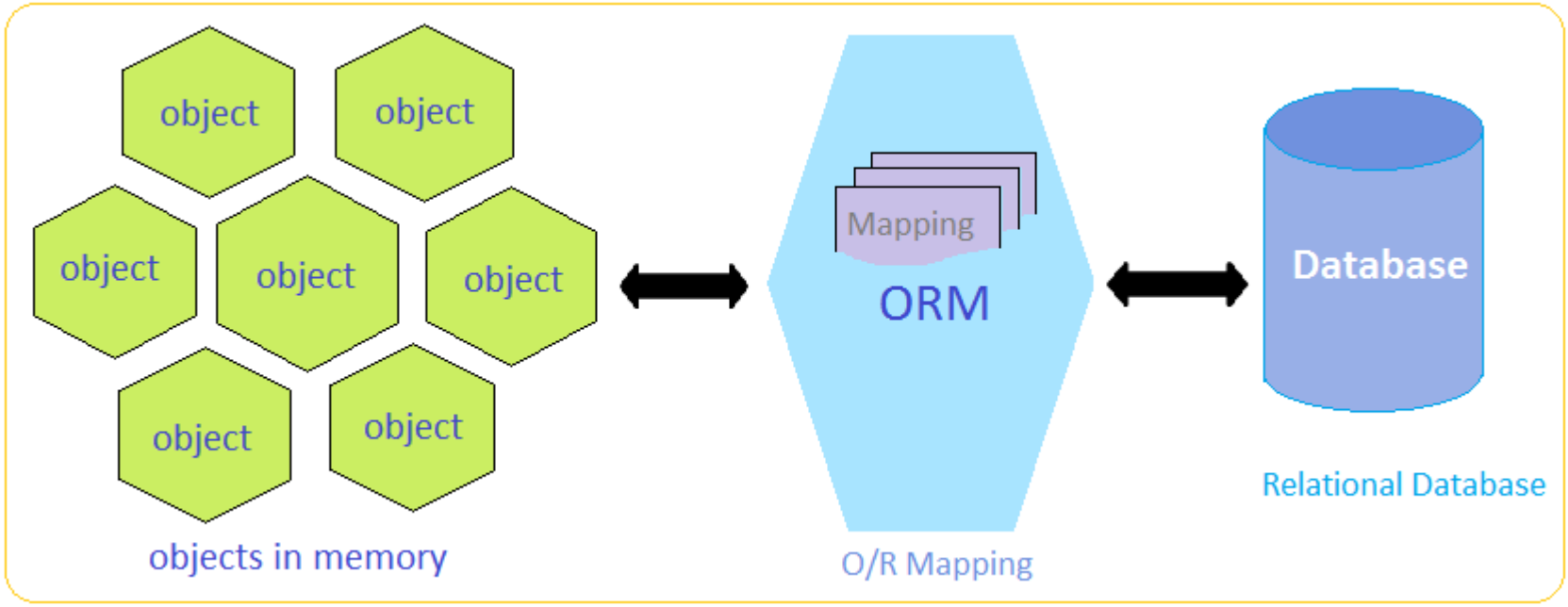
```
print(track.join(album))
print(track.join(album).join(genre))
print(track.join(album).join(artist))
print(track.join(album).join(genre).join(artist))

result = conn.execute(select([track.c.title, album.c.title, genre.c.name, artist.c.name])
                      .select_from(track.join(album).join(genre).join(artist)))

for row in result.fetchall():
    print(row)


result = conn.execute(track
                      .select()
                      .select_from(track.join(album).join(genre).join(artist))
                      .where(
                          and_(
                              genre.c.id==1,
                              artist.c.id==1,
                          )
                      )
                     )

for row in result.fetchall():
    print(row)
```

○ Open/Close

```
from sqlalchemy import create_engine, MetaData

engine = create_engine("sqlite:///alchemy_core.db", echo=True)
conn = engine.connect()

metadata = MetaData(bind=engine, reflect=True)
metadata.reflect(bind=engine)

for row in metadata.tables:
    print(row)
```

```
tables = metadata.tables
for table in tables:
    print(table)

#album

track = metadata.tables["Track"]
track

for row in conn.execute(track.select()).fetchall():
    print(row)

conn.close()
metadata.clear()
```

■ SQLAlchemy ORM



Data Mapping to Classes

■ Declare

○ declarative_base

```
sqlalchemy.ext.declarative.declarative_base
```

➢ Construct a base class for declarative class definitions

○ 예제

```
from sqlalchemy import create_engine

engine = create_engine("sqlite:///:memory:", echo=True)
```

```
from sqlalchemy.ext.declarative import declarative_base

base = declarative_base()
```

➢ declarative_base() callable returns a new base class from which all mapped classes should inherit

■ Create

○ declarative_base 상속 class 선언

```
class User(base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column("passwd", String)

    def __repr__(self):
        return "<T'User(name='%s', fullname='%s', password='%s')>" \
                % (self.name, self.fullname, self.password)
```

- a new Table and mapper() will have been generated
- table and mapper are accessible via __table__ and __mapper__ attributes

○ Schema

```
User.__table__
```

○ Create Table

```
base.metadata.create_all(engine)
```

○ Create Instance

```
kim = User(name="kim", fullname="anonymous, Kim", password="kimbap heaven")

print(kim)
print(kim.id)
```

■ Session

○ Session

```
class sqlalchemy.orm.session.Session
```

- Session establishes all conversations with the database and represents all the objects
- Manages persistence operations for ORM-mapped objects

○ sessionmaker

```
class sqlalchemy.orm.session.sessionmaker
```

- sessionmaker factory generates new Session objects when called
- sessionmaker class is normally used to create a top level Session configuration

○ 예제

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

■ Insert

○ add

```
add(instance, _warn=True)
```

- Place an object in the Session, persisted to the database on the next flush operation.
- Repeated calls to add() will be ignored

○ addall

```
add_all(instances)
```

- Add the given collection of instances to this Session

○ 예제

```
session.add(kim)

session.add_all([
    User(name="lee", fullname="unknown, Lee", password="123456789a"),
    User(name="park", fullname="nobody, Park", password="Parking in Park")
])
```

- Pending. no SQL has yet been issued and the object is not yet represented by a row in the database

■ Update

○ dirty

```
dirty
```

- The set of all persistent instances considered dirty
- Instances are considered dirty when they were modified but not deleted

○ is_modified

```
is_modified(instance, include_collections=True, passive=True)
```

- Return True if the given instance has locally modified attributes

○ 예제

```
kim.password = "password"

session.dirty

session.is_modified(kim)
```

■ Commit

○ commit

```
commit()
```

- ➢ Flush pending changes and commit the current transaction
- ➢ If no transaction is in progress, this method raises an InvalidRequestError

■ Select

○ query

```
class sqlalchemy.orm.query.Query(entities, session=None)
```

- ➢ ORM-level SQL construction object
- ➢ Query is the source of all SELECT statements generated by the ORM

○ 예제

```
for row in session.query(User):
    print(type(row))
    print(row.id, row.name, row.fullname, row.password)
```

○ filter

filter(**criterion*)

- Apply the given filtering criterion to a copy of this Query, using SQL expressions
- Allow you to use regular Python operators with the class-level attributes on your mapped class

○ filter_by

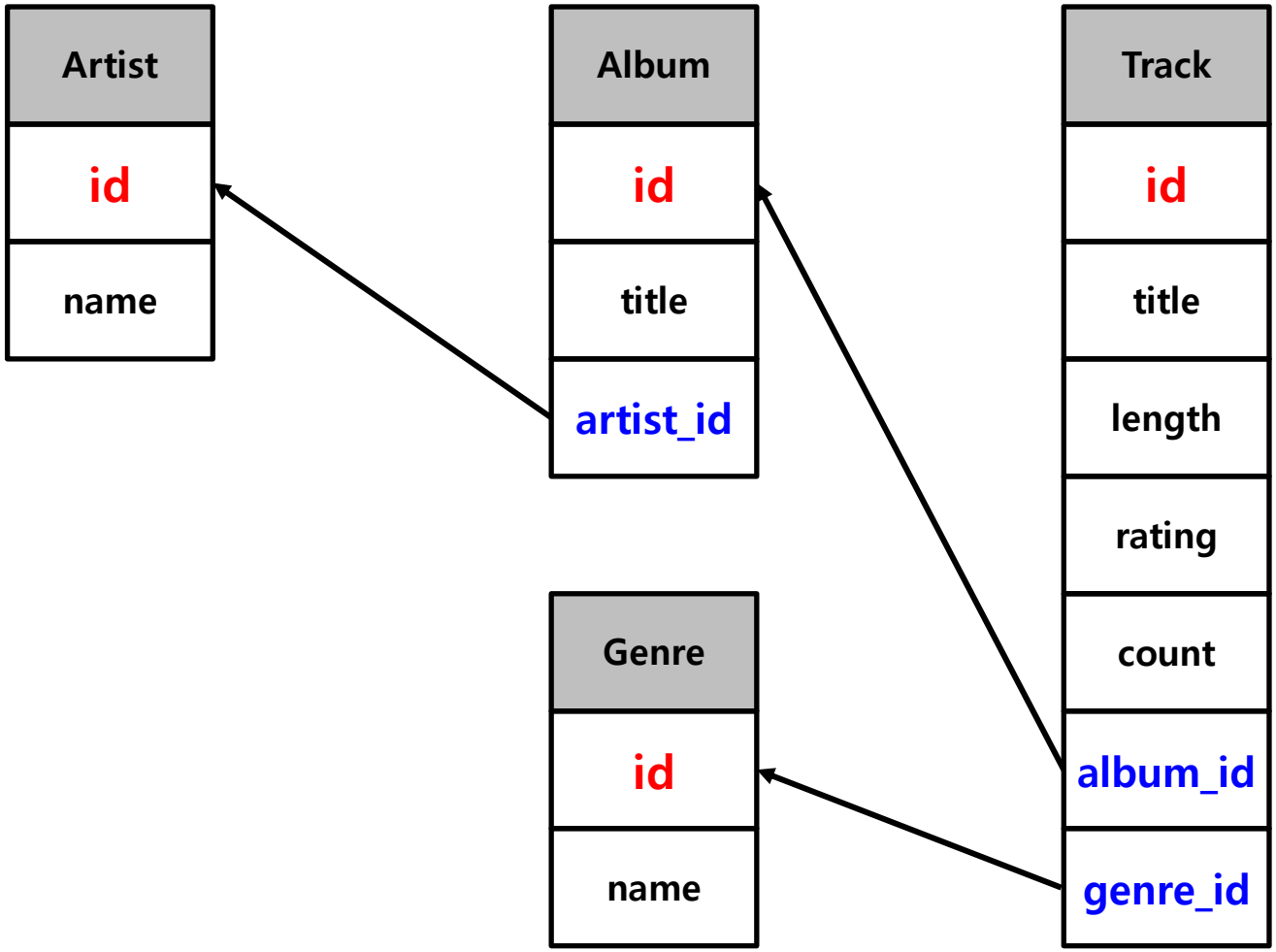
filter_by(***kwargs*)

- apply the given filtering criterion to a copy of this Query, using keyword expressions

○ 예제

```
for row in session.query(User.id, User.fullname).filter(User.name == "lee"):
    print(type(row))
    print(row.id, row.fullname)

for row in session.query(User.id, User.fullname).filter_by(name = "lee"):
    print(type(row))
    print(row.id, row.fullname)
```

■ 예제


Artist
id
name
Album
id
title
artist_id
Track
id
title
length
rating
count
album_id
genre_id
Genre
id
name

○ Create

```
class Artist(Base):
    __tablename__ = "Artist"

    id = Column(Integer, primary_key=True)
    name = Column(String)

class Album(Base):
    __tablename__ = "Album"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    artist_id = Column(Integer, ForeignKey("Artist.id"))

class Genre(Base):
    __tablename__ = "Genre"

    id = Column(Integer, primary_key=True)
    name = Column(String)

class Track(Base):
    __tablename__ = "Track"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    length = Column(Integer)
    rating = Column(Integer)
    count = Column(Integer)
    album_id = Column(Integer, ForeignKey("Album.id"))
    genre_id = Column(Integer, ForeignKey("Genre.id"))
```

○ Insert

```
artist1 = Artist(name="Led Zepplin")
artist2 = Artist(name="AC/DC")

session.add_all([artist1, artist2])
session.commit()

album = [Album(title="IV", artist_id=artist1.id),
         Album(title="Who Made Who", artist_id=artist2.id)]

session.add_all(album)
session.commit()

session.add_all([Genre(name="Rock"), Genre(name="Metal")])
session.commit()

album1 = session.query(Album).filter(Album.artist_id==artist1.id).one()
album2 = session.query(Album).filter(Album.artist_id==artist2.id).one()

genre1 = session.query(Genre).filter(Genre.name=="Rock").filter(Genre.id==1).one()
genre2 = session.query(Genre).filter(Genre.name=="Metal").filter(Genre.id==2).one()

track = [Track(title="Black Dog", rating=5, length=297, count=0, album_id=album1.id, genre_id=genre1.id),
         Track(title="Stairway", rating=5, length=482, count=0, album_id=album1.id, genre_id=genre2.id),
         Track(title="About to rock", rating=5, length=313, count=0, album_id=album2.id, genre_id=genre1.id),
         Track(title="Who Made Who", rating=5, length=297, count=0, album_id=album2.id, genre_id=genre2.id)]
session.add_all(track)
session.commit()
```

○ Join

```
result = session.query(Track.title, Album.title, Genre.name, Artist.name) \
         .select_from(Track) \
         .join(Album)\
         .join(Genre)\
         .join(Artist).all()

for row in result:
    print(row)

songList = [dict(Track=row[0], Album=row[1], Genre=row[2], Artist=row[3]) for row in result]

songList
```

■ Relationship

○ relationship

```
sqlalchemy.orm.relationship
```

- Provide a relationship between two mapped classes
- This corresponds to a parent-child or associative table relationship

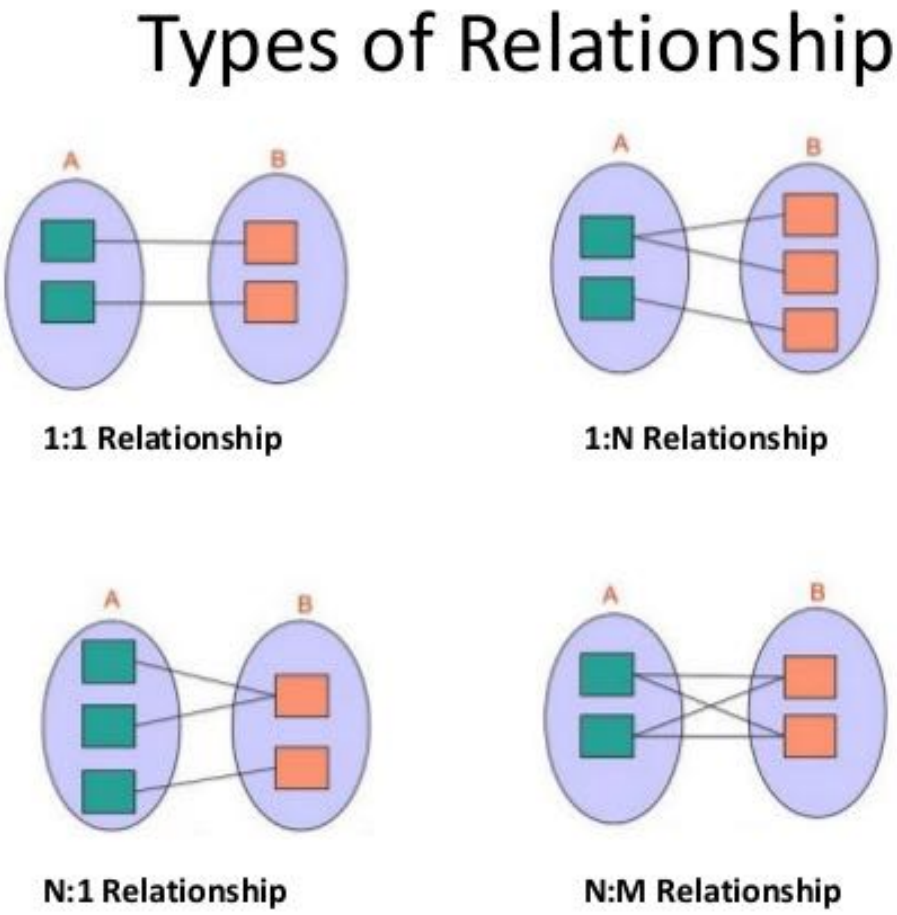
○ back_populates / backref

- indicates the string name of a property to be placed on the related mapper's class that will handle this relationship in the other direction

○ uselist

- a boolean that indicates if this property should be loaded as a list or a scalar

■ Relationship Model



■ 예제

```
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column("passwd", String)

    #addresses = relationship("Address", back_populates="user")
    #addresses = relationship("Address", back_populates="user", uselist=False)

    def __repr__(self):
        return "<User(name='%s', fullname='%s', password='%s')>" % (self.name, self.fullname, self.password)
```

```
class Address(Base):
    __tablename__ = "addresses"

    id = Column(Integer, primary_key=True)
    email_address = Column(String, nullable=False)
    user_id = Column(Integer, ForeignKey("users.id"))

    #user = relationship("User", back_populates="addresses", uselist=False)
    #user = relationship("User", back_populates="addresses")
    #user = relationship("User")

    def __repr__(self):
        return "<Address(email_address='%s')>" % self.email_address
```

■ 예제

```
class Artist(Base):
    __tablename__ = "Artist"

    id = Column(Integer, primary_key=True)
    name = Column(String)

    albumList = relationship("Album", back_populates="artist")

class Album(Base):
    __tablename__ = "Album"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    artist_id = Column(Integer, ForeignKey("Artist.id"))

    artist = relationship("Artist", back_populates="albumList", uselist=False)
    trackList = relationship("Track", back_populates="album")

class Genre(Base):
    __tablename__ = "Genre"

    id = Column(Integer, primary_key=True)
    name = Column(String)

    trackList = relationship("Track", back_populates="genre")

class Track(Base):
    __tablename__ = "Track"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    length = Column(Integer)
    rating = Column(Integer)
    count = Column(Integer)
    album_id = Column(Integer, ForeignKey("Album.id"))
    genre_id = Column(Integer, ForeignKey("Genre.id"))

    album = relationship("Album", back_populates="trackList", uselist=False)
    genre = relationship("Genre", back_populates="trackList", uselist=False)
```

Insert

```
track1 = Track(title="Black Dog", rating=5, length=297, count=0)
track2 = Track(title="Stairway", rating=5, length=482, count=0)
track3 = Track(title="About to rock", rating=5, length=313, count=0)
track4 = Track(title="Who Made Who", rating=5, length=297, count=0)
```

```
track1.album = track2.album = Album(title="IV")
track3.album = track4.album = Album(title="Who Made Who")
```

```
track1.genre = track3.genre = Genre(name="Rock")
track2.genre = track4.genre = Genre(name="Metal")
```

```
track1.album.artist = track2.album.artist = Artist(name="Led Zepplin")
track3.album.artist = track4.album.artist = Artist(name="AC/DC")
```

Select

```
print("Title: %s, Album: %s, Genre: %s, Artist: %s" %
      (track1.title, track1.album.title, track1.genre.name, track1.album.artist.name))
print("Title: %s, Album: %s, Genre: %s, Artist: %s" %
      (track3.title, track3.album.title, track3.genre.name, track3.album.artist.name))
```

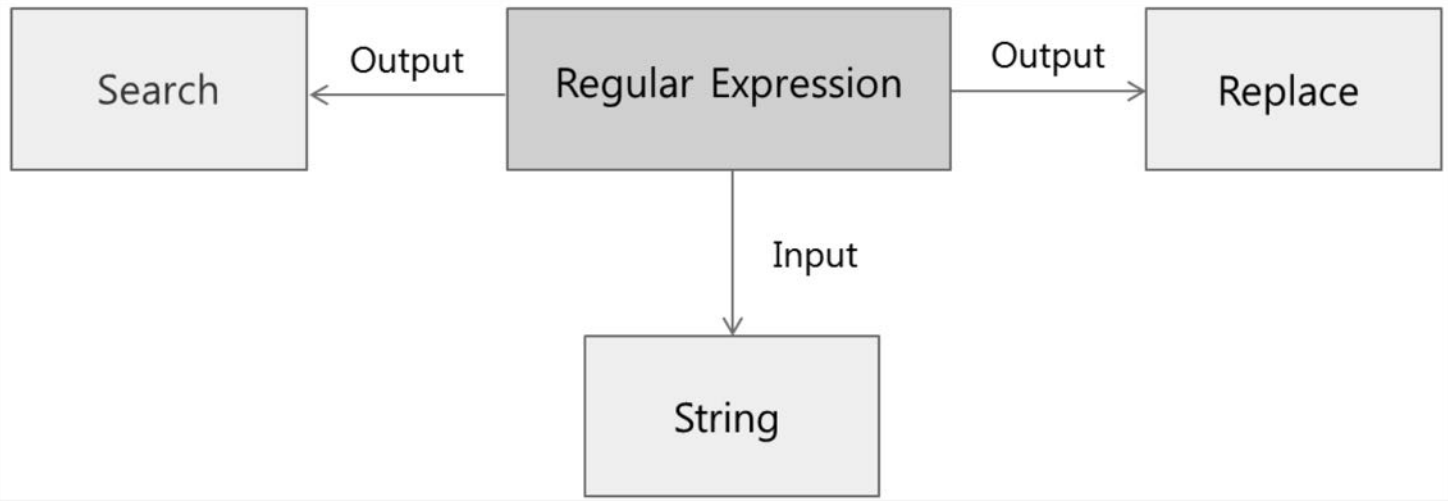
```
print("TrackID: %d, AlbumID: %d, GenreID: %d, Artist: %d" %
      (track1.id, track1.album.id, track1.genre.id, track1.album.artist.id))

print("TrackID: %d, AlbumID: %d, GenreID: %d, Artist: %d" %
      (track3.id, track3.album.id, track3.genre.id, track3.album.artist.id))
```

2.3. Regular Expression

■ What is RE?

○ a special text string for describing a search pattern

○ searching, replacing, and parsing text with complex patterns of characters



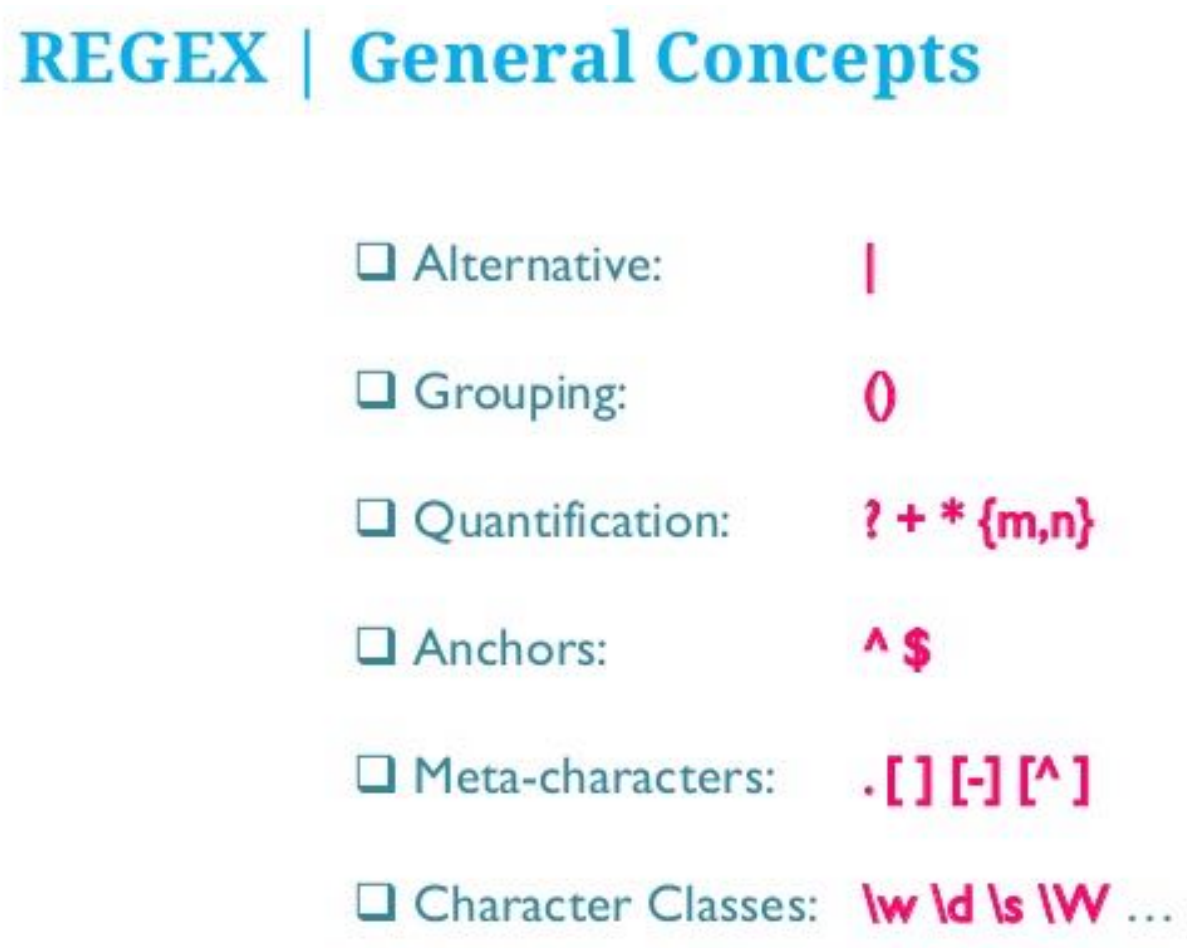
■ Why use RE?

○ Processes large amounts of text over and over again / Extremely fast

○ Usually this pattern is then used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation

■ But, There is a learning curve

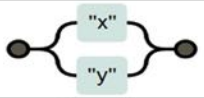
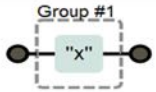
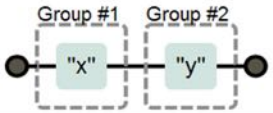
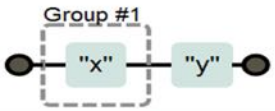
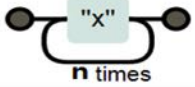
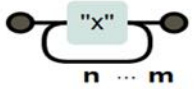
■ General concepts



■ Meta characters

○ . ^ $ * + ? { } [ ] ₩ | ( )

- .(dot)
  - a.b
  - a+모든문자+b
  - aab, a0b, ~~abc~~
- *(asterisk)
  - ab*c
  - a+b(0번 이상 반복)+c
  - ac, abc, abbbbbbbc
- +(plus)
  - ab+c
  - a+b(1번 이상 반복)+c
  - ~~ac~~, abc, abbbbbbbc
- {n | min, | min, max}(curly braces)
  - {0,}=*, {1,}=+
  - ab{1}c, ab{2,6}c
  - a+b(1번 반복, 2번-6번)+c
  - abc, abbc, abbbbbbc
- ?(question)
  - ab?c, b{0,1}
  - a+b(1개 있거나, 없거나)+c
  - ac, abc, ~~abbbc~~

정규표현식	표현	설명
^x		문자열이 x로 시작합니다.
x$		문자열이 x로 끝납니다.
.x		임의의 한 문자를 표현합니다. (x가 마지막으로 끝납니다.)
x+		x가 1번이상 반복합니다.
x?		x가 존재하거나 존재하지 않습니다.
x*		x가 0번이상 반복합니다.
x|y		x 또는 y를 찾습니다. (or연산자를 의미합니다.)
(x)		( )안의 내용을 캡쳐하며, 그룹화 합니다.
(x)(y)		그룹화 할 때, 자동으로 앞에서부터 1번부터 그룹 번호를 부여해서 캡쳐합니다. 결과값에 그룹화한 Data가 배열 형식으로 그룹번호 순서대로 들어갑니다.
(x)(?:y)		캡쳐하지 않는 그룹을 생성할 경우 ?:를 사용합니다. 결과값 배열에 캡쳐하지 않는 그룹은 들어가지 않습니다.
x{n}		x를 n번 반복한 문자를 찾습니다.
x{n,}		x를 n번이상 반복한 문자를 찾습니다.
x{n,m}		x를 n번이상 m번이하 반복한 문자를 찾습니다.

■ Character classes

○ [], [-], [^], \d, \D, \s, \S, \w, \W

- ➢ [](square braket)
  - – [abc]
  - – a, b or c
- ➢ -
  - – [a-z]
  - – a부터 z까지
- ➢ ^
  - – [^a-z]
  - – a부터 z까지를 제외

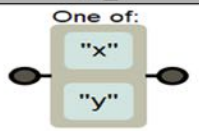
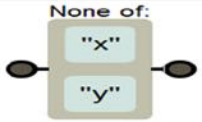
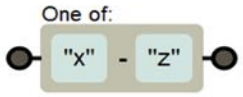
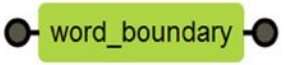
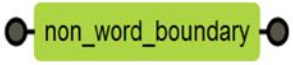
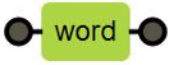
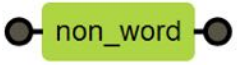
■ Anchors

○ ^, $, \b, \B

- ➢ ^ - Starting position
- ➢ $ - Ending position

■ Grouping

- ➢ ()(parenthesis)
  - – (abc)

정규표현식	표현	설명
[xy]		x,y중 하나를 찾습니다.
[^xy]		x,y를 제외하고 문자 하나를 찾습니다. (문자 클래스 내의 ^는 not을 의미합니다.)
[x-z]		x~z 사이의 문자중 하나를 찾습니다.
₩^		^(특수문자)를 식에 문자 자체로 포함합니다. (escape)
₩b		문자와 공백사이의 문자를 찾습니다.
₩B		문자와 공백사이가 아닌 값을 찾습니다.
₩d		숫자를 찾습니다.
₩D		숫자가 아닌 값을 찾습니다.
₩s		공백문자를 찾습니다.
₩S		공백이 아닌 문자를 찾습니다.
₩t		Tab 문자를 찾습니다.
₩v		Vertical Tab 문자를 찾습니다.
₩w		알파벳 + 숫자 + _ 를 찾습니다.
₩W		알파벳 + 숫자 + _ 을 제외한 모든 문자를 찾습니다.

■ Match method

○ Greedy vs Lazy(non greedy)

- ➢ Greedy – tries to find the last possible match
- ➢ Lazy – tries to find the first possible match

Greedy quantifier	Lazy quantifier	Description
*	*?	Match zero or more times.
+	+?	Match one or more times.
?	??	Match zero or one time.
{n}	{n}?	Match exactly n times.
{n,}	{n,}?	Match at least n times.
{n,m}	{n,m}?	Match from n to m times.

Add a ? to a quantifier to make it ungreedy i.e lazy.

Example:

test string : *stackoverflow*

greedy reg expression : `s.*o` output: **stackoverflo**w

lazy reg expression : `s.*?o` output: **stacko**verflow

■ 예제

○ https://www.regexr.com/

○ https://www.regexper.com/

○ Alternative

- cat|mat → 'car' or 'mat'
- regular|expression → 'regular' or 'expression'

○ Grouping

- gr(e|a)y → 'grey' or 'gray'
- py(pi|tho(n|nic) → 'pypi' or 'python' or 'pythonic'

○ Quantification

- colou?r → 'color' or 'colour'
- 81*5 → '85' or '815' or '8111115'
- 81+5 → '815' or '8111115'
- go{2,3}gle → 'google' or 'gooogle'
- go{2,} → 'goo' or 'gooooooooooooo'

○ Anchors – match the starting or ending position

- ^obje → 'object' or 'object-oriented'
- ^2018 → '2018' or '2018-07-10'
- gram$ → 'program' or 'kilogram'

○ Meta characters – match a single character

- bat. → 'bat' or 'bats' or 'bata'
- [xyz] → 'x' or 'y' or 'z'
- [aeiou] → any vowel
- [0123456789] → any digit
- [a-c] → 'a' or 'b' or 'c'
- [a-zA-Z] → all letters(uppercase, lowercase)
- [0-9] → all digits
- [^aeiou] → any non-vowel
- [^0-9] → any non digit
- [^xyz] → any character, but not 'x' or 'y' or 'z'

○ Character classes

- \d – digits / \D – non digits
- \s – a single white space character / \S – non white space
- \w – alphanumeric character [a-ZA-Z0-9_] / \W – non alphanumeric
- \b – word boundaries / \B – non word boundaries

○ 'Hello World'

- (H . .) . (o . .)
  - 'Hell', 'o W'
- l+
  - He'll'o, Wor'l'd
- H.?e
  - 'He'
- l.+?o
  - 'llo'
- el*o
  - 'ello'
- l{1,2}
  - He'l'lo, He'll'o, Wor'l'd
- [aeiou]+
  - 'e', 'o', 'o'
- (Hello|Hi|Pogo)
  - 'Hello'
- llo\b
  - 'llo'
- \w
  - 'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'
- \W
  - ' '

○ 'In Hello World'

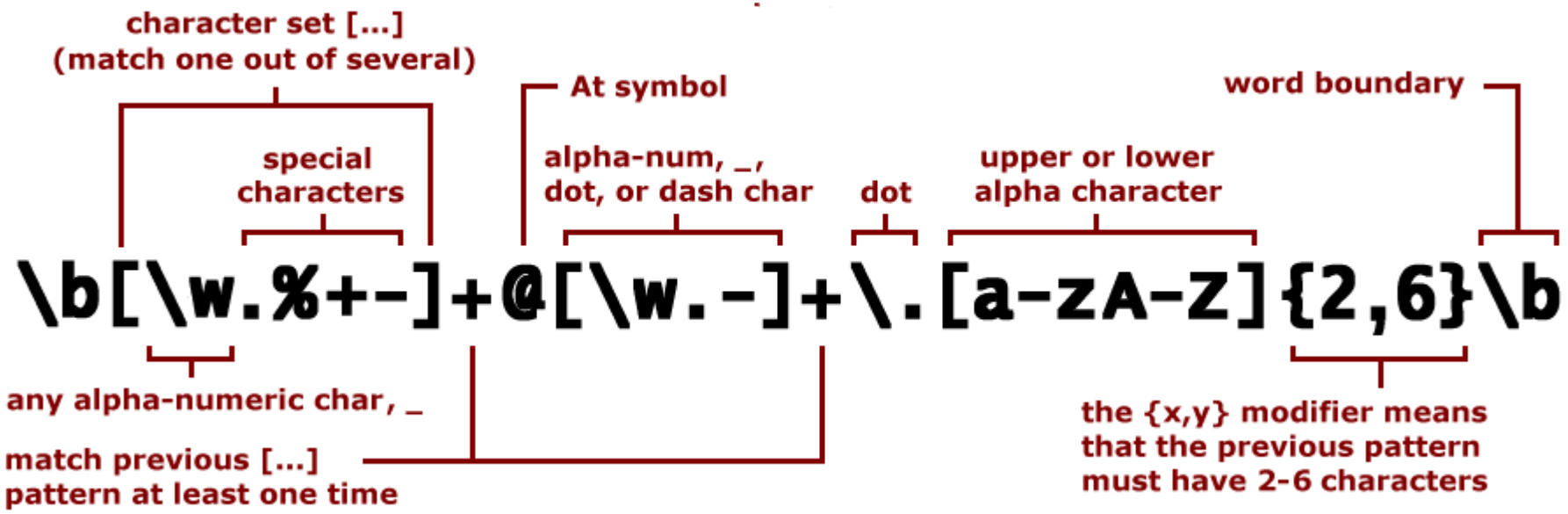
- \s.*\s
  - ' Hello '

○ 'Hello World'

- \S.*\S
  - 'Hello World'
- ^He
  - 'He'
- rld$
  - 'rld'
- el*o
  - 'ello'
- [^Hlo]
  - 'e', ' ', 'W', r', 'd'

○ 예제

➢ username@domain.com



○ 예제

<h1 style="color:red">Heading</h1>

<([A-Z][A-Z0-9]*)[^>]*>(.*?)<\/\1>

+00-00-0000-0000

+00 00 0000 0000

000-0000-0000

000 0000 0000

([\+]?[0-9]{2}[\s\-]?)?0?1[0-9]{1}[\s\-]?[0-9]{4}[\s\-]?[0-9]{4}

○ re

```
import re
```

re.compile(*pattern*, *flags=0*)

Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods, described below.

re.search(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding match object. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

re.match(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in `MULTILINE` mode, `re.match()` will only match at the beginning of the string and not at the beginning of each line.

re.split(*pattern*, *string*, *maxsplit=0*, *flags=0*) ¶

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.

○ search vs match

```
content = "Hello World"

print(re.search("W", content))
print(re.match("W", content))
```

- search
  - find something ***anywhere*** in the string and return a match object
- match
  - find something at the ***beginning*** of the string and return a match object

Method	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사한다.
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사한다.
findall()	정규식과 매치되는 모든 문자열(substring)을 리스트로 리턴한다
finditer()	정규식과 매치되는 모든 문자열(substring)을 iterator 객체로 리턴한다

○ sub

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged.

```
data = """
park 800905-1049118
kim  700905-1059119
"""

result = []
for line in data.split("\n"):
    word_result = []
    for word in line.split(" "):
        if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():
            word = word[:6] + "-" + "*******"
        word_result.append(word)
    result.append(" ".join(word_result))
print("\n".join(result))
```

```
data = """
park 800905-1049118
kim  700905-1059119
"""

pat = re.compile("(\d{6})[-]\d{7}")
print(pat.sub("\g<1>-*******", data))
```

○ Meta characters

➢ |

```
p = re.compile('Crow|Servo')
m = p.match('CrowHello')
print(m)
```

➢ ^

```
print(re.search('^Life', 'Life is too short'))
print(re.search('^Life', 'My Life'))
```

➢ $

```
print(re.search('short$', 'Life is too short'))
print(re.search('short$', 'Life is too short, you need python'))
```

➢ +

```
p = re.compile('(ABC)+')
m = p.search('ABCABCABC OK?')
print(m.group())
```

○ Meta characters

➢ \b

```
p = re.compile(r'\bclass\b')
print(p.search('no class at all'))
print(p.search('one subclass is'))
print(p.search('the declassified algorithm'))
```

➢ \B

```
p = re.compile(r'\Bclass\B')
print(p.search('no class at all'))
print(p.search('one subclass is'))
print(p.search('the declassified algorithm'))
```

➢ group

```
p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
m = p.search("park 010-1234-1234")
print(m)

p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
m = p.search("park 010-1234-1234")
print(m.group(1))
```

○ group

`Match.`**`group`**`([`*`group1`*`, ...])`

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an `IndexError` exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

method	목적
group()	매치된 문자열을 리턴한다.
start()	매치된 문자열의 시작 위치를 리턴한다.
end()	매치된 문자열의 끝 위치를 리턴한다.
span()	매치된 문자열의 (시작, 끝) 에 해당되는 튜플을 리턴한다.

➢ 예제

```
m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(1, 2))
```

```
p = re.compile(r"(\w+) (\w+)")
m = p.search("Isaac Newton, physicist")
print(m.group())

p.sub("\g<2> \g<1>", "Isaac Newton, physicist")
```

■ 문제

다음 중 정규식 `a[.]{3,}b` 과 매치되는 문자열은 무엇일까?

1. acccb
2. a....b
3. aaab
4. a.cccb

```
>>> import re
>>> p = re.compile("[a-z]+")
>>> m = p.search("5 python")
>>> m.start() + m.end()
```

다음과 같은 문자열에서 핸드폰 번호 뒷자리인 숫자 4개를 ####로 바꾸는 프로그램을 정규식을 이용하여 작성해 보자.

```
"""
park 010-9999-9988
kim 010-9909-7789
lee 010-8789-7768
"""
```

다음은 이메일 주소를 나타내는 정규식이다. 이 정규식은 park@naver.com, kim@daum.net, lee@myhome.co.kr 등과 매치된다. 긍정형 전방 탐색 기법을 이용하여 .com, .net이 아닌 이메일 주소는 제외시키는 정규식을 작성해 보자.

```
.*[@].*[.].*$
```