Databases
E-R Model (contd…)
Object Relational Mapping

COMP 1531, 17s2

Aarthi Natarajan

Week 10

Exercise from Week 9:

Exercise 1: Give an ER design to model the following scenario ...

- Patients are identified by an **SSN**, and their **names**, **addresses** and **ages** must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty and years of experience must be recorded.
- Each pharmacy has a name, address and phone number. A pharmacy must be **managed** by a pharmacist.
- A pharmacist is identified by an SSN, he/she can only work for one pharmacy. For each pharmacist, the name, qualification must be recorded.
- For each drug, the trade name and formula must be recorded.
- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy **sells** several drugs, and has a price for each. A drug could be sold at several pharmacies, and the price could vary between pharmacies.
- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and quantity associated with it.

What kind of data, relationships and constraints exist?

Possible data:

- people: doctors, patients, pharmacists
- pharmacies, drugs
- person's SSN, name, address(?)
- doctor: as for person + specialty
- pharmacist: as for person + qualification etc.

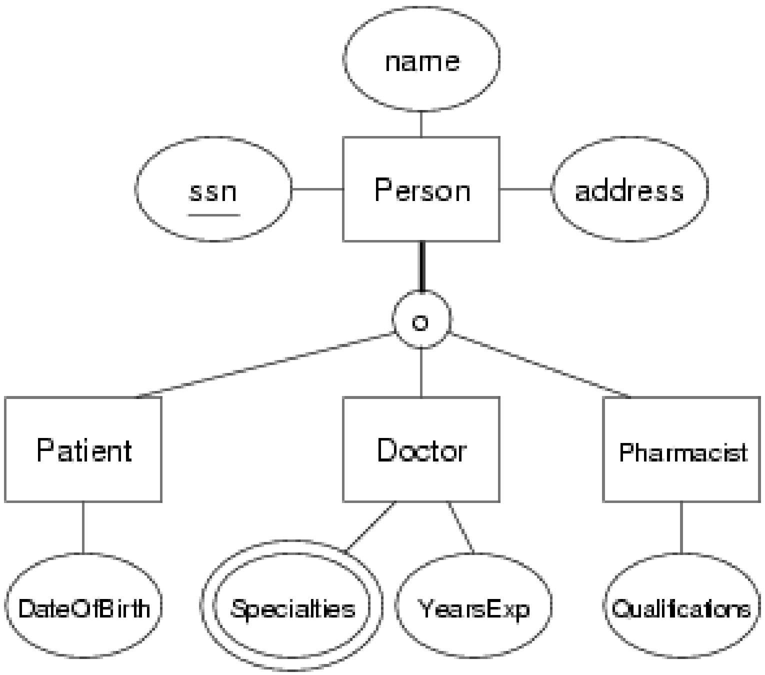
Possible relationships:

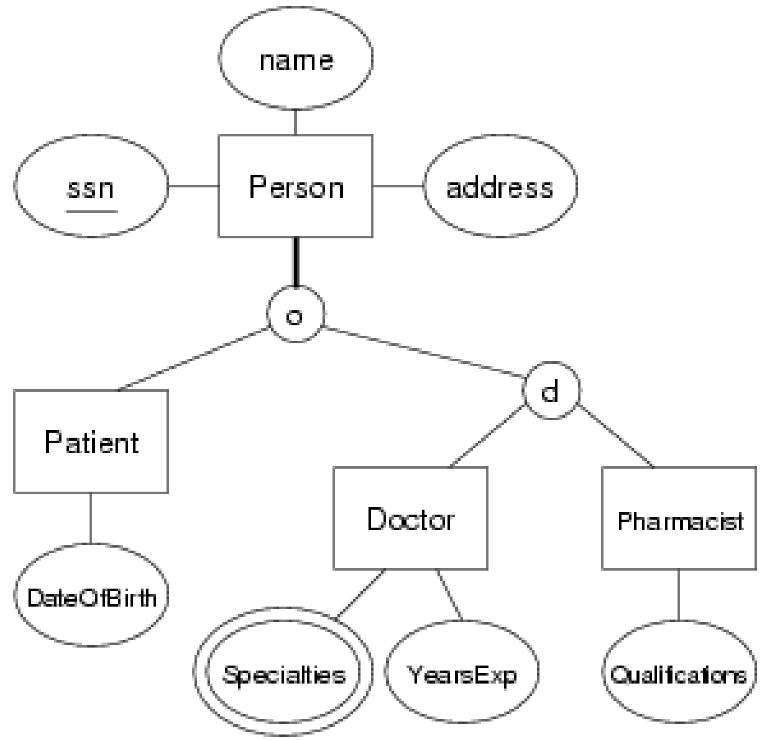
- doctors **treat** patients, patients **have primary** physician
- pharmacists **work in** pharmacies
- drugs **are sold in** pharmacies
- doctors **prescribe** drugs **for** patients etc.

Possible constraints:

- every person has **exactly one, unique** SSN
- pharamacist works in ≤ 1 pharmacy
- patient has **exactly one** primary physician
- doctor treats ≥ 1 patient etc.

First, modelling the classes of people





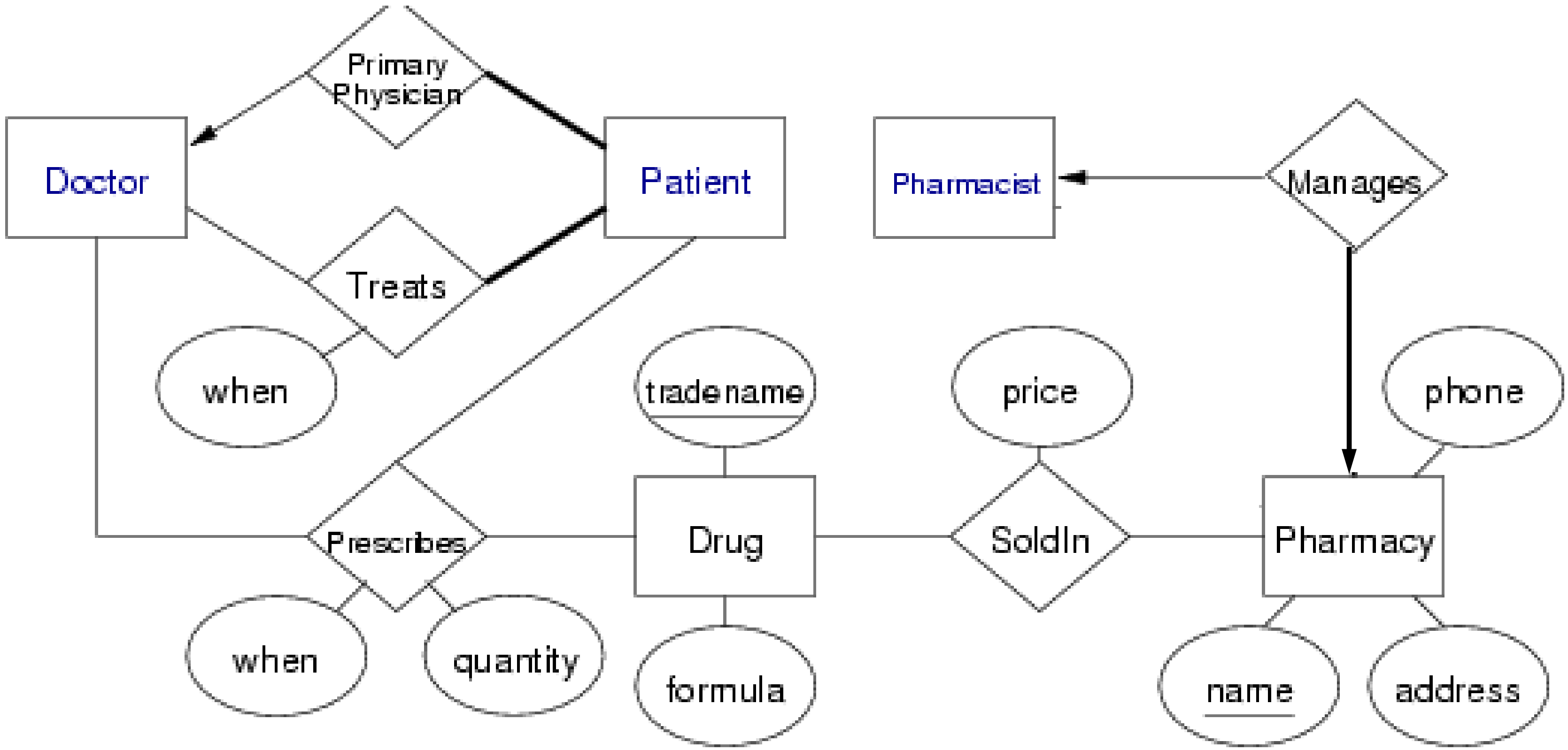
One possible model

- Here, Person can be either a Patient or Doctor or a Pharmacist

A more realistic design

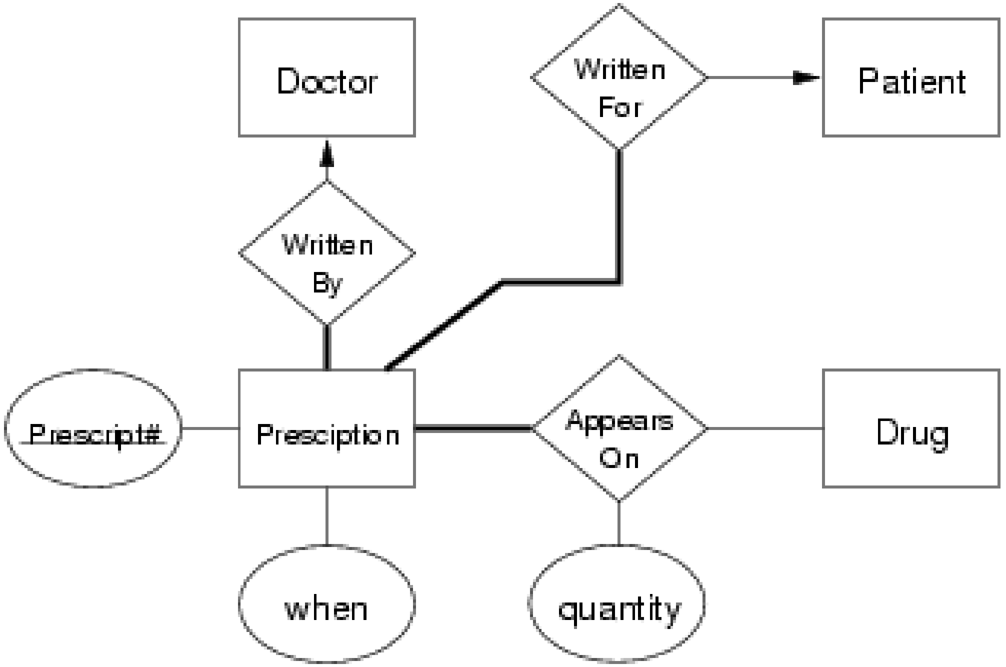
- Someone who is a Doctor cannot also be a Pharmacist

Now, the overall ER data model



- Here, we omit the Person entity, as it is not directly involved in any relationship
- Attributes of Patient, Pharmacist and Doctor are omitted for clarity
- This model treats line-items on a single prescription separately
- ***Is this a problem?***

- The previous model does not explicitly represent Prescriptions.
- If necessary, a revised model can be used to model two drugs prescribed by the same doctor for the same patient on the same day are part of the same prescription



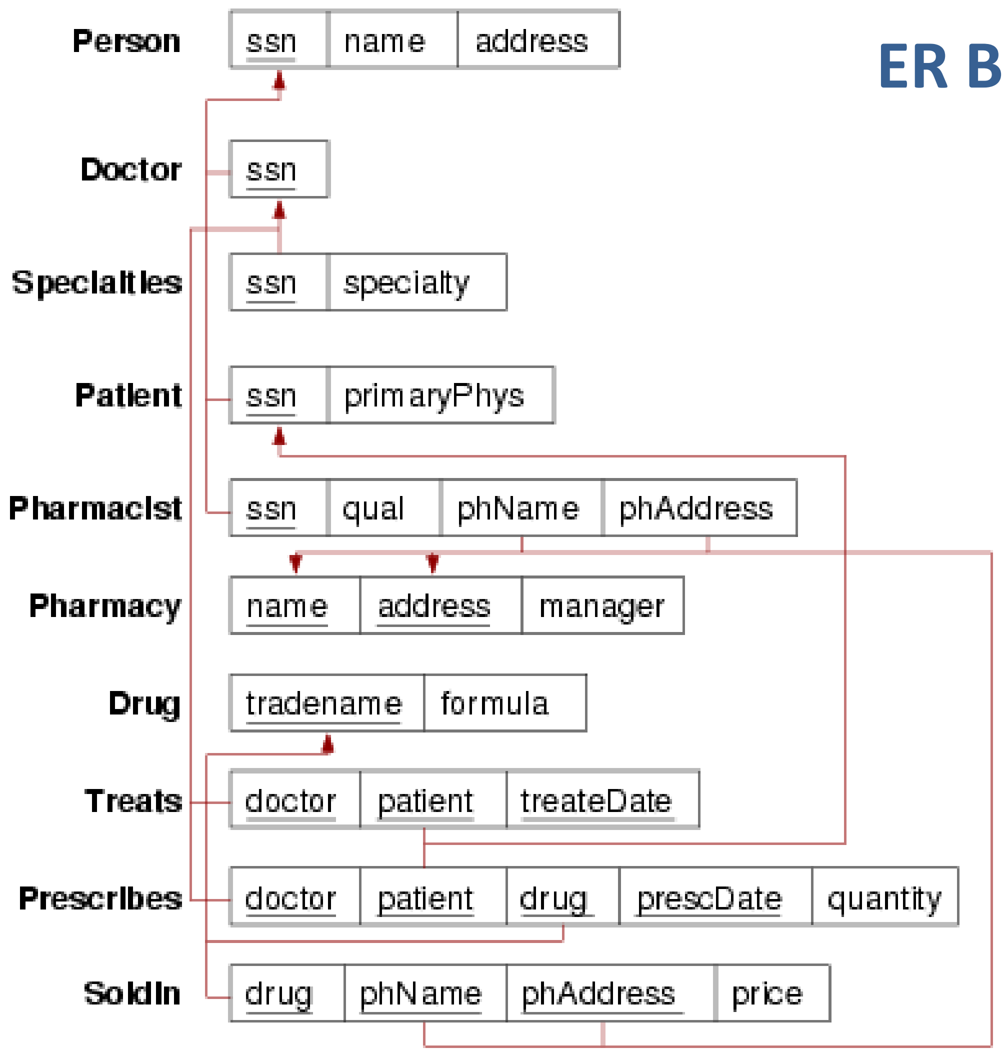
Some of the inherent assumptions in the above model:

- every prescription has some kind of unique identifying number
- every prescription must be written by a particular doctor
- every prescription must be written for a particular patient
- a prescription must contain some drugs, but may have several drugs

Exercise 2: Convert the ER design (in slide 6) to relational form

Which elements of the ER design do not appear in relational form?

ER Box Schema



- Person classes are mapped using the ER-style mapping
- The relational model cannot represent the total participation constraints for patients (i.e. every patient must be treated by at least one doctor.

Exercise 3: Provide the SQL schema for the relationships

Assume the definitions of Person/Patient/Doctor/Pharmacist -- as given in the ER-style

```
create table Pharmacy (
   name varchar (30),
   address varchar(100),
   manager integer not null, -- total participation
   primary key (name,address),
   foreign key (manager) references Pharmacist(ssn)
);

alter table Pharmacist add
          foreign key (phName,phAddress)
          references Pharmacy(name,address);

create table Drug (
           tradename varchar(40),
           formula varchar(100),
           primary key (tradename)
);
```

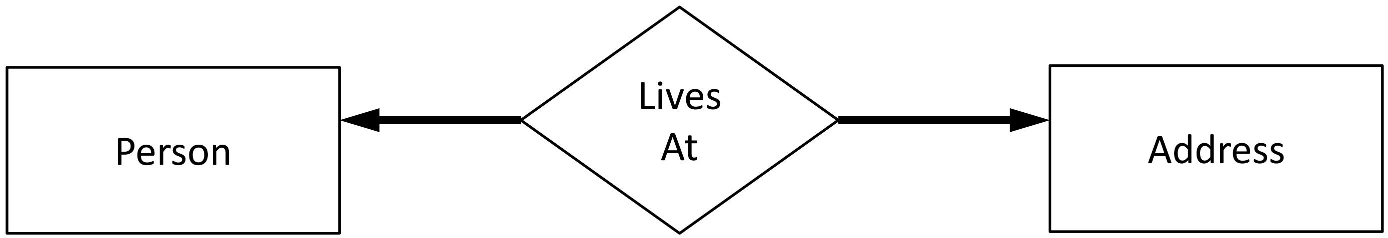
```sql
-- if "treatDate" is date only, and is part of primary key,
-- then a doctor cannot treat a patient more than once/day.
-- if this is required, make "treatDate" as a timestamp
create table Treats (
       doctor integer,
       patient integer,
       treatDate date,
       primary key (doctor, patient, treatDate),
       foreign key (doctor) references Doctor(ssn),
       foreign key (patient) references Patient(ssn) );

-- if "prescDate" is date only, then cannot prescibe
-- the same drug more than once on the same day
create table Prescribes (
       doctor integer,   patient integer,
       drug varchar(40), prescDate date,
       quantity integer, -- float if mass/volume/...
       primary key (doctor, patient, drug, prescDate),
       foreign key (doctor) references Doctor(ssn),
       foreign key (patient) references Patient(ssn),
       foreign key (drug) references Drug(tradename) );
```

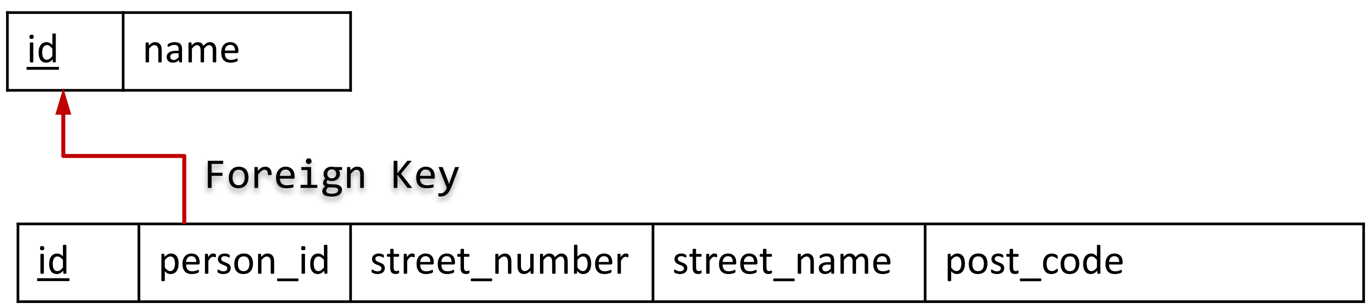
```
create table SoldIn (
       drug varchar(40),
       phName varchar(30),
       phAddress varchar(100),
       price money,
       primary key (drug,phName,phAddress),
       foreign key (phName,phAddress) references
                                        Pharmacy(name,address),
       foreign key (drug) references Drug(tradename) );
```

Object Relational Mapping

Exercise 4: Implement the ER design in SQLite3



Mapping the ER design to the relational model, we have:



Using Python Code with raw SQL to access database

```
import sqlite3

conn = sqlite3.connect('data.db')

c = conn.cursor()
c.execute(''' CREATE TABLE person (
             id INTEGER PRIMARY KEY ASC, name varchar(250) NOT NULL)''')

c.execute('''CREATE TABLE address (
          id INTEGER PRIMARY KEY ASC, street_name varchar(250),
          street_number varchar(250), post_code varchar(250) NOT NULL,
          person_id INTEGER NOT NULL,
          FOREIGN KEY(person_id) REFERENCES person(id))''')

c.execute(''' INSERT INTO person VALUES(1, 'Mary Poppins') ''')
c.execute(''' INSERT INTO address VALUES(1, 'Magic Lane', '01', '12000', 1) ''')
conn.commit()

#*******Query to select the records************

c.execute('SELECT * FROM person')
print(c.fetchall())
c.execute('SELECT * FROM address')
print(c.fetchall())

conn.close()
```

Using raw SQL in Python code requires:

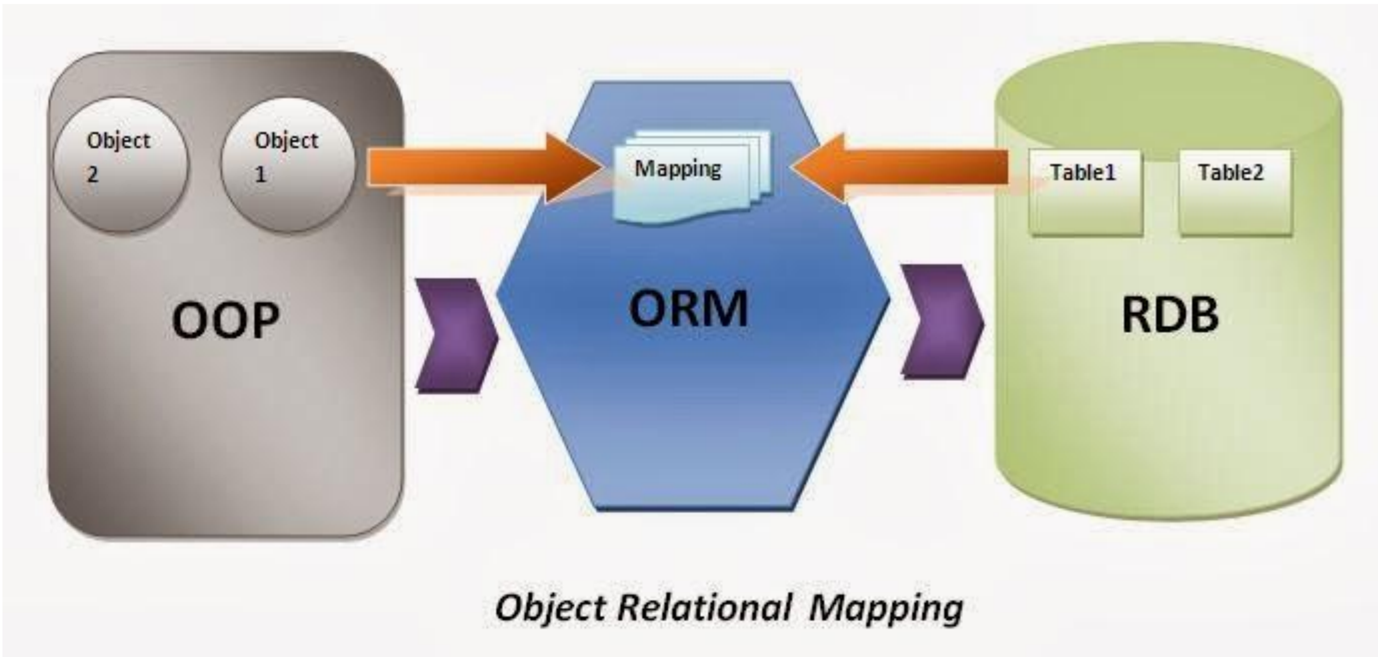
- Manually building the raw SQL statements
- Passing these to the database engine
- Parse the returned results as an array of records

Writing raw SQL in Python code can be:

- Error-Prone
- Hard-to-maintain

Object Relational Mapper (ORMs)

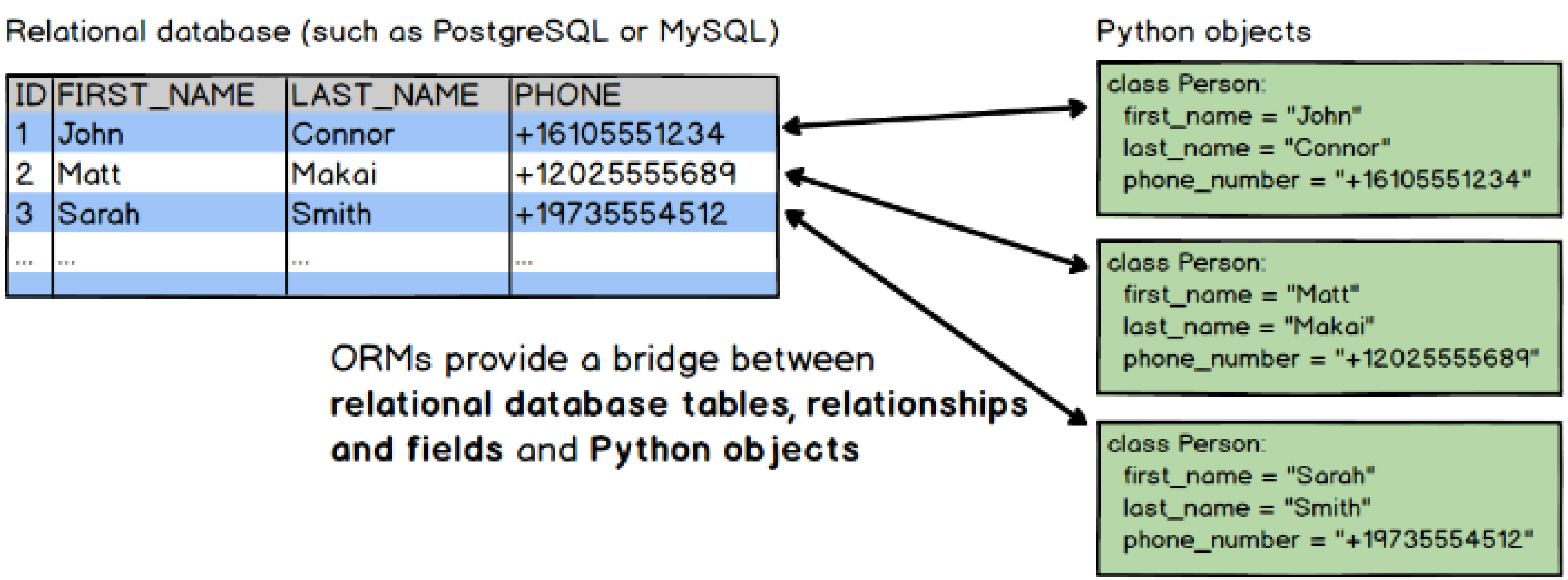
- A high-level abstraction framework that maps a relational database system to objects
- Automates all the CRUD (create/retrieve/update/delete) operations
- ORM is agnostic to which relational database is used (well at least theoretically…)



- Many different ORM frameworks around e.g., Hibernate, TopLink, SQLAlchemy

Why are ORMs useful?

- Shields the developer from having to write complex SQL statements and focus on the application logic using their choice of programming language
- Harmonisation of data types between the OO language and the SQL database
- Automates transfer of data stored in relational database tables into objects



SQLAlchemy

- A Python SQL Toolkit and ORM providing developers a comprehensive set of tools to work with databases and Python
- Supported databases include SQLite, Postgresql, MySQL, Oracle, MS-SQL, Firebird, Sybase
- Supports Python 2.5 onwards
- Supports three ways of working with database data:
  - Raw SQL
  - SQL Expression language
  - ORM
- Next, let us look how to use ORM with SQLAlchemy

How does SQLAlchemy ORM work?

The SQLAlchemy Object Relational Mapper maps:

a) user-defined Python classes to database tables
b) table rows to instance objects and
c) columns to instance attributes

There are two approaches to using SQLAlchemy ORM:

1. **Manual ORM**
2. **Using *Declarative Extensions***

Manual Object Relational Mapping

Three components in implementing ORM manually in SQLAlchemy:

- A *Table* that represents a table in a database.
- A *mapper* that maps a Python class to a table in a database.
- A *user-defined class* that defines how a database record maps to a normal Python object.

Step 1: Open a connection to the database

```
from sqlalchemy import create_engine
db = create_engine('sqlite:///tutorial.db')
```

The `create_engine()` method:

- returns a `SQLEngine` object, which:
  - knows how to talk to a particular type of database (SQLite, MySQL)
  - serves as a connection object, create a pool of database connections and re-use them automatically as needed (but you don't worry about these…)
- takes a URI of the form:
  - **"engine://user:password@host:port/database"**
  - e.g., "sqlite:///tutorial.db"

(to connect to a SQLite database in the local directory URI)

Step 2: Configuring the database

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import sessionmaker,mapper

# Connect to database
engine = create_engine('sqlite:///person_alchemy.db')

# Create a metadata instance –
# An object that manages all the metadata, i.e. information about
data e.g., table definitions (columns, datatypes etc)

metadata = MetaData(engine)

# Create a Session
#  A Session is the primary interface for persistence operations in SQLAlchemy ORM.
#  It establishes and maintains all conversations between the program and the database.

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

Step 3: Map a user-defined class to the relational table

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

# Create a user-defined class
class Person(object):
    def __init__(self,id,name):
        self.id = id
        self.name = name

# Declare a table
person = Table('user10', metadata,
               Column('id',Integer,primary_key = True),
               Column('name',String)
               )
# Create a table
metadata.create_all()

# Map the user-defined class to table
mapper(Person,person)
```

SQLAlchemy with Declarative Extension

- Instead of having to write code for *Table*, *mapper* and the *user-defined class* at different places, SQLAlchemy's *declarative* allows the three entities to be defined at once in one class definition.
- Makes use of a *declarative base class* which maintains a catalog of classes and tables
- The base class is created with the *declarative_base()* function.

Step 1: Initial configurations

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base

# Create an engine that stores data in the local directory's
# sqlalchemy_example.db file.

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

# Create a declarative base class
Base=declarative_base()

#create a session

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

Step 2: Create all the tables in the database

```
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship

class Person(Base):
  __tablename__ = 'person'
  # Here we define columns for the table person
  # Notice that each column is also a normal Python instance attribute.
  id   = Column(Integer, primary_key=True)
  name = Column(String(250), nullable=False)

class Address(Base):
  __tablename__ = 'address'
  # Here we define columns for the table address.
  id = Column(Integer, primary_key=True)
  street_name = Column(String(250))
  street_number = Column(String(250))
  post_code = Column(String(250), nullable=False)
  person_id = Column(Integer, ForeignKey('person.id'))
  person = relationship(Person)

# Create all tables in the engine
Base.metadata.create_all(engine)
```

Step 3: Insert the records into the tables

```
# Insert a Person in the person table
p1 = Person(id = 1, name='Mary Poppins')
session.add(p1)

# Insert an Address in the address table
addr = Address(street_name = 'Magic Lane',
               street_number = 22,
               post_code='12000', person=p1)
session.add(addr)

# Insert a Person in the person table
p2 = Person(id = 2, name = ‘Roald Dahl’)
session.add(p2)

# Insert an Address in the address table
addr = Address(street_name = 'Kids Pde',
               street_number = 26,
               post_code='26000', person=p2)

session.add(addr)
```

Step 4: Query the tables

```
#Make a query to find all Persons in the database

rowset = session.query(Person).all()
for record in rowset:
    print(record.name)
    addr =
       session.query(Address).filter(Address.person==record).one()
    print(record.name + “ lives at " + addr.street_number + " " +
          addr.street_name + " " + addr.post_code)
```

Benefits of ORM:

- SQLAlchemy's *declarative* ORM is much more object-oriented, enabling you to easily map objects and relational data
- Enables developers to easily create, read, update and delete SQLAlchemy objects as if they were normal Python objects
- Flexible design, enterprise-level APIs, making code robust and adaptable

Cons:

- Affects performance due to the overhead of mapping relational to ORM
- A heavyweight API; leading to a long learning curve

Tomorrow:

Guest Lecture on Cyber Security by Sean Yeoh