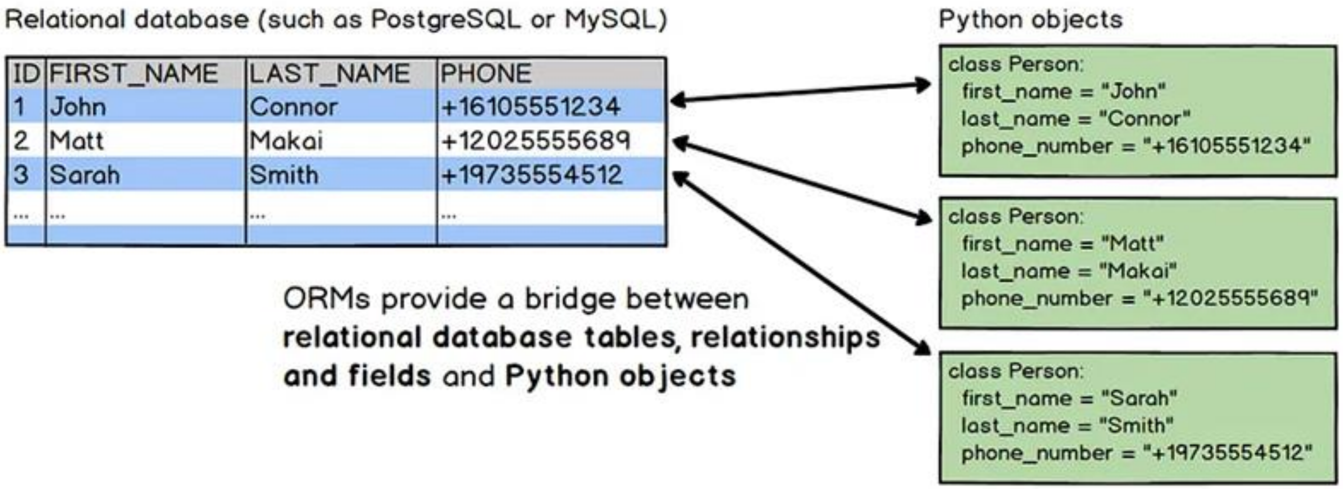
COMP642

Object Oriented Programming

Lectorial 8 – SQL Alchemy

ORM – Object Relational Mapper

- a programming technique that allows us to interact with relational databases using an object-oriented programming language (like Python, or Java).
- turn database records into objects so that we can interact and perform operations on those database records as if we are dealing with objects.



Relational database (such as PostgreSQL or MySQL)

ID	FIRST_NAME	LAST_NAME	PHONE
1	John	Connor	+16105551234
2	Matt	Makai	+12025555689
3	Sarah	Smith	+19735554512
...	...	...	...

Python objects

```
class Person:
  first_name = "John"
  last_name = "Connor"
  phone_number = "+16105551234"
```

```
class Person:
  first_name = "Matt"
  last_name = "Makai"
  phone_number = "+12025555689"
```

```
class Person:
  first_name = "Sarah"
  last_name = "Smith"
  phone_number = "+19735554512"
```

ORMs provide a bridge between **relational database tables, relationships** and **fields** and **Python objects**

https://python.plainenglish.io/a-beginners-guide-to-sqlalchemy-orm-simplifying-database-interactions-with-python-6874e88b6b3

SQLAlchemy

- a popular SQL toolkit and Object Relational Mapper
- written in Python and gives full power and flexibility of SQL to an application developer
- open source and cross-platform software
- famous for its object-relational mapper (ORM)

Features of SQLAlchemy

- **Object-Relational Mapping (ORM)**: allows developers to map Python classes to database tables and vice versa.
- **SQL Expression Language**: allows developers to generate complex SQL queries in a Pythonic way.
- **Database Connection Pooling**: allows developers to manage multiple database connections efficiently.
- **Data Integrity and Transactions**: support for transactions and data integrity constraints, such as foreign keys, unique constraints, and check constraints.
- **Cross-database Compatibility**: consistent API for interacting with different database systems.

SQLAlchemy ORM

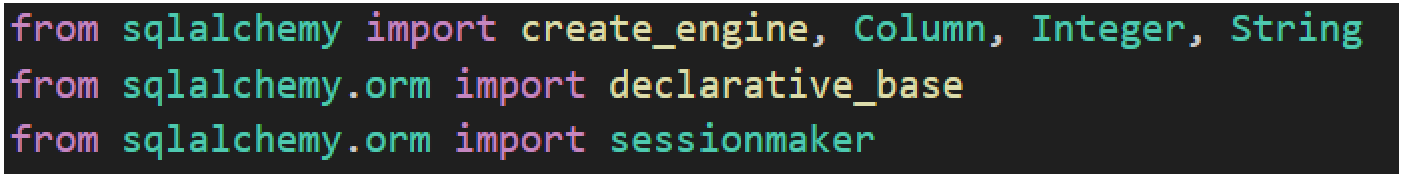
- a Python library that bridges the gap between Python code and relational database.
- a powerful tool that helps in managing and interacting with databases using Python classes and objects.

Setting Up SQLAlchemy (1)

1. Install SQLAlchemy using pip

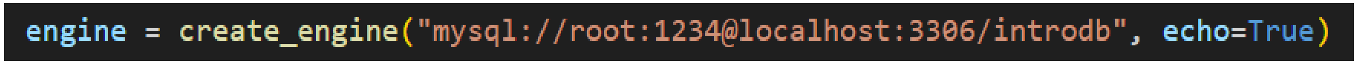
    pip install sqlalchemy

2. Import the necessary components



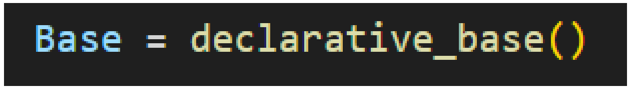
```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import sessionmaker
```

3. Create an engine – database connection



```
engine = create_engine("mysql://root:1234@localhost:3306/introdb", echo=True)
```

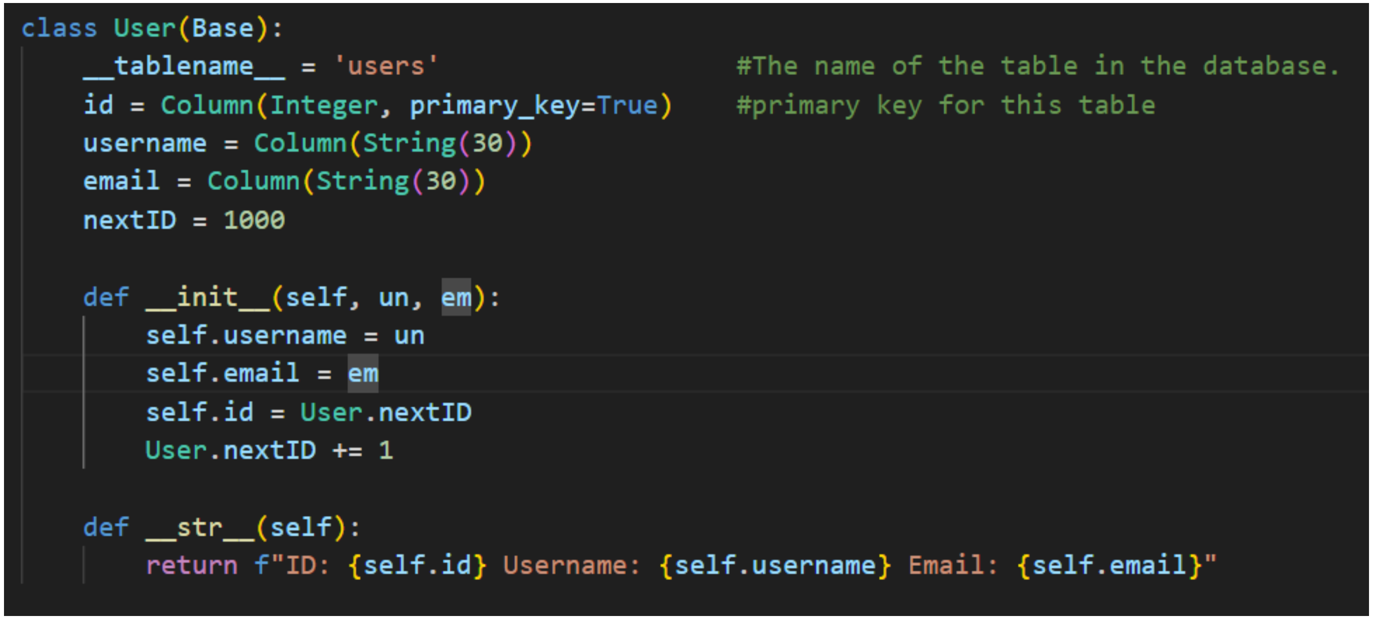
4. Creating a Base – base class for all the classes that will be mapped to the database tables



```
Base = declarative_base()
```

Setting Up SQLAlchemy (2)

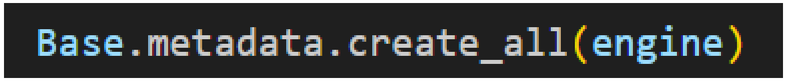
5. Define the model



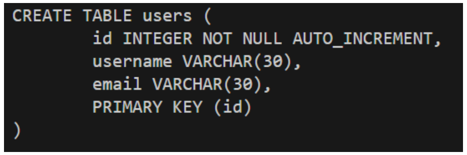
```python
class User(Base):
    __tablename__ = 'users'                   #The name of the table in the database.
    id = Column(Integer, primary_key=True)    #primary key for this table
    username = Column(String(30))
    email = Column(String(30))
    nextID = 1000

    def __init__(self, un, em):
        self.username = un
        self.email = em
        self.id = User.nextID
        User.nextID += 1

    def __str__(self):
        return f"ID: {self.id} Username: {self.username} Email: {self.email}"
```

Setting Up SQLAlchemy (3)

5. Create the table

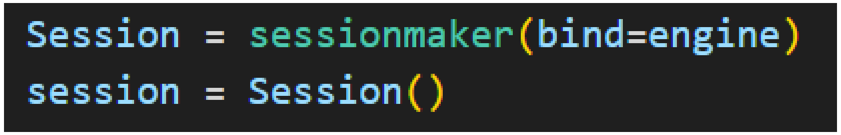


```
Base.metadata.create_all(engine)
```



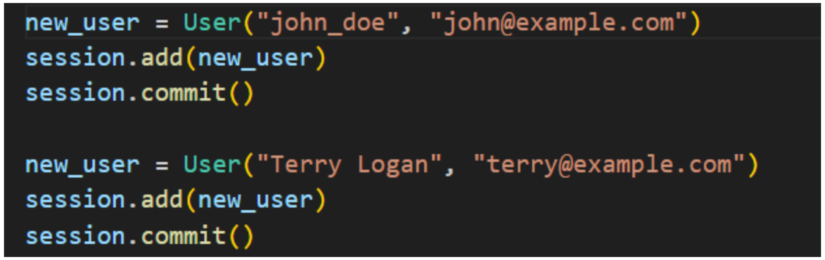
```
CREATE TABLE users (
        id INTEGER NOT NULL AUTO_INCREMENT,
        username VARCHAR(30),
        email VARCHAR(30),
        PRIMARY KEY (id)
)
```

6. Create the session – (like a workspace)

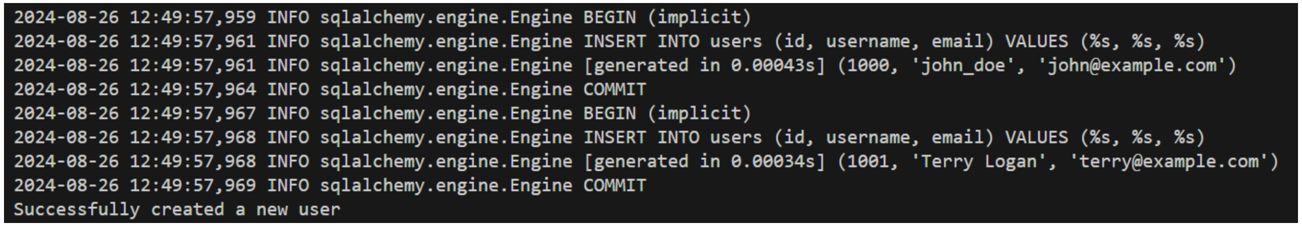


```
Session = sessionmaker(bind=engine)
session = Session()
```

7. Create new records (users)



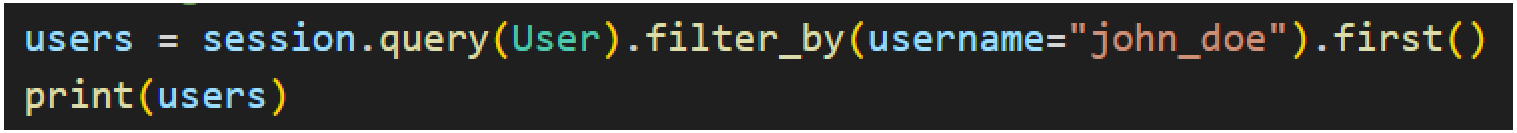
```
new_user = User("john_doe", "john@example.com")
session.add(new_user)
session.commit()

new_user = User("Terry Logan", "terry@example.com")
session.add(new_user)
session.commit()
```



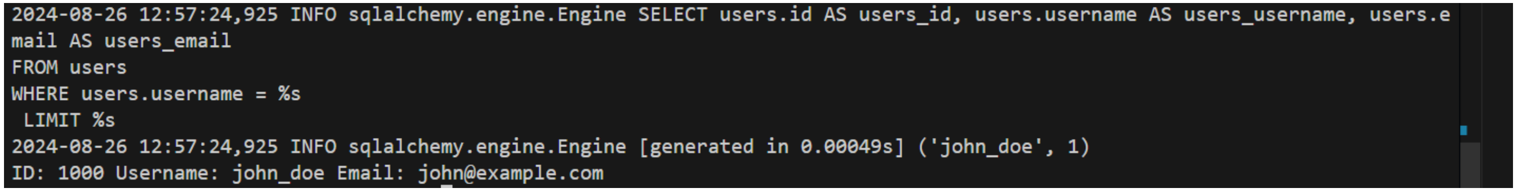
```
2024-08-26 12:49:57,959 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-08-26 12:49:57,961 INFO sqlalchemy.engine.Engine INSERT INTO users (id, username, email) VALUES (%s, %s, %s)
2024-08-26 12:49:57,961 INFO sqlalchemy.engine.Engine [generated in 0.00043s] (1000, 'john_doe', 'john@example.com')
2024-08-26 12:49:57,964 INFO sqlalchemy.engine.Engine COMMIT
2024-08-26 12:49:57,967 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-08-26 12:49:57,968 INFO sqlalchemy.engine.Engine INSERT INTO users (id, username, email) VALUES (%s, %s, %s)
2024-08-26 12:49:57,968 INFO sqlalchemy.engine.Engine [generated in 0.00034s] (1001, 'Terry Logan', 'terry@example.com')
2024-08-26 12:49:57,969 INFO sqlalchemy.engine.Engine COMMIT
Successfully created a new user
```

Setting Up SQLAlchemy (4)

8. Read the data

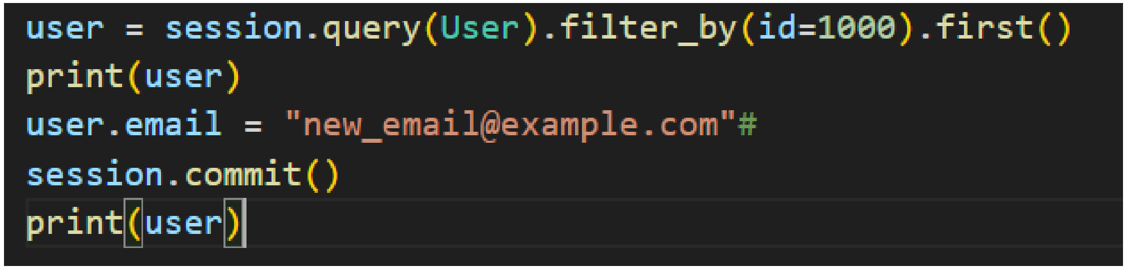


```
users = session.query(User).filter_by(username="john_doe").first()
print(users)
```

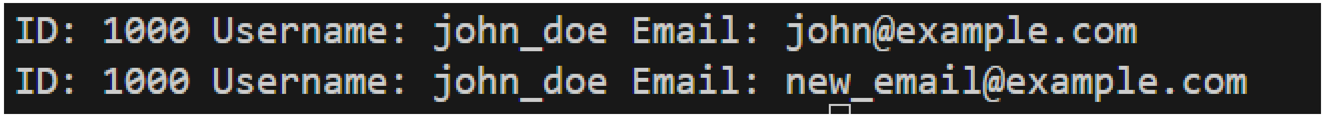


```
2024-08-26 12:57:24,925 INFO sqlalchemy.engine.Engine SELECT users.id AS users_id, users.username AS users_username, users.e
mail AS users_email
FROM users
WHERE users.username = %s
 LIMIT %s
2024-08-26 12:57:24,925 INFO sqlalchemy.engine.Engine [generated in 0.00049s] ('john_doe', 1)
ID: 1000 Username: john_doe Email: john@example.com
```

9. Update the data



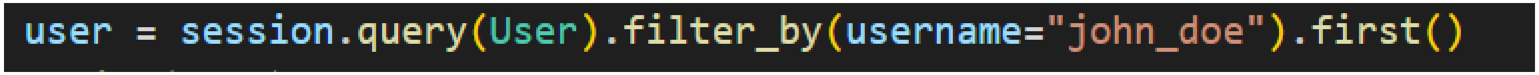
```
user = session.query(User).filter_by(id=1000).first()
print(user)
user.email = "new_email@example.com"#
session.commit()
print(user)
```



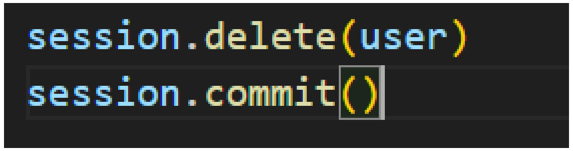
```
ID: 1000 Username: john_doe Email: john@example.com
ID: 1000 Username: john_doe Email: new_email@example.com
```

Setting Up SQLAlchemy (5)

8. Delete a record



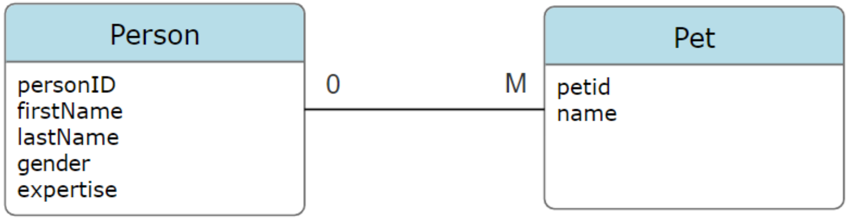
```
user = session.query(User).filter_by(username="john_doe").first()
```



```
session.delete(user)
session.commit()
```

9. Querying the database
   - Filters, joins (more examples later)
10. Transactions and commits – changes to the database are made within a transaction. Need to explicitly commit changes using **session.commit()** to save them permanently.
11. Close the session – **session.close()**

Creating Relationships between Tables (1)



Person

personID
firstName
lastName
gender
expertise

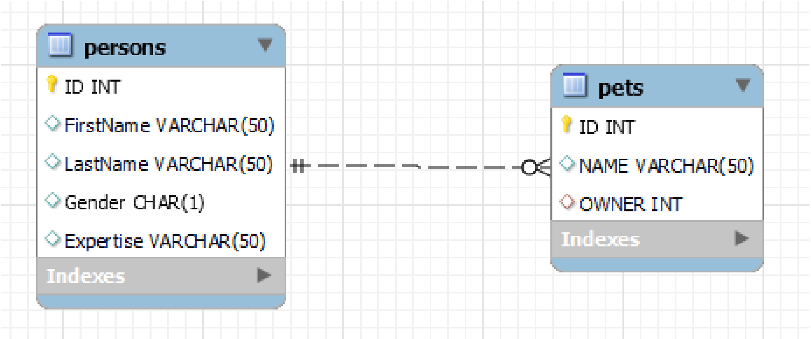
0

M

Pet

petid
name

A person has zero or more pets
A pet belongs to a person.



persons

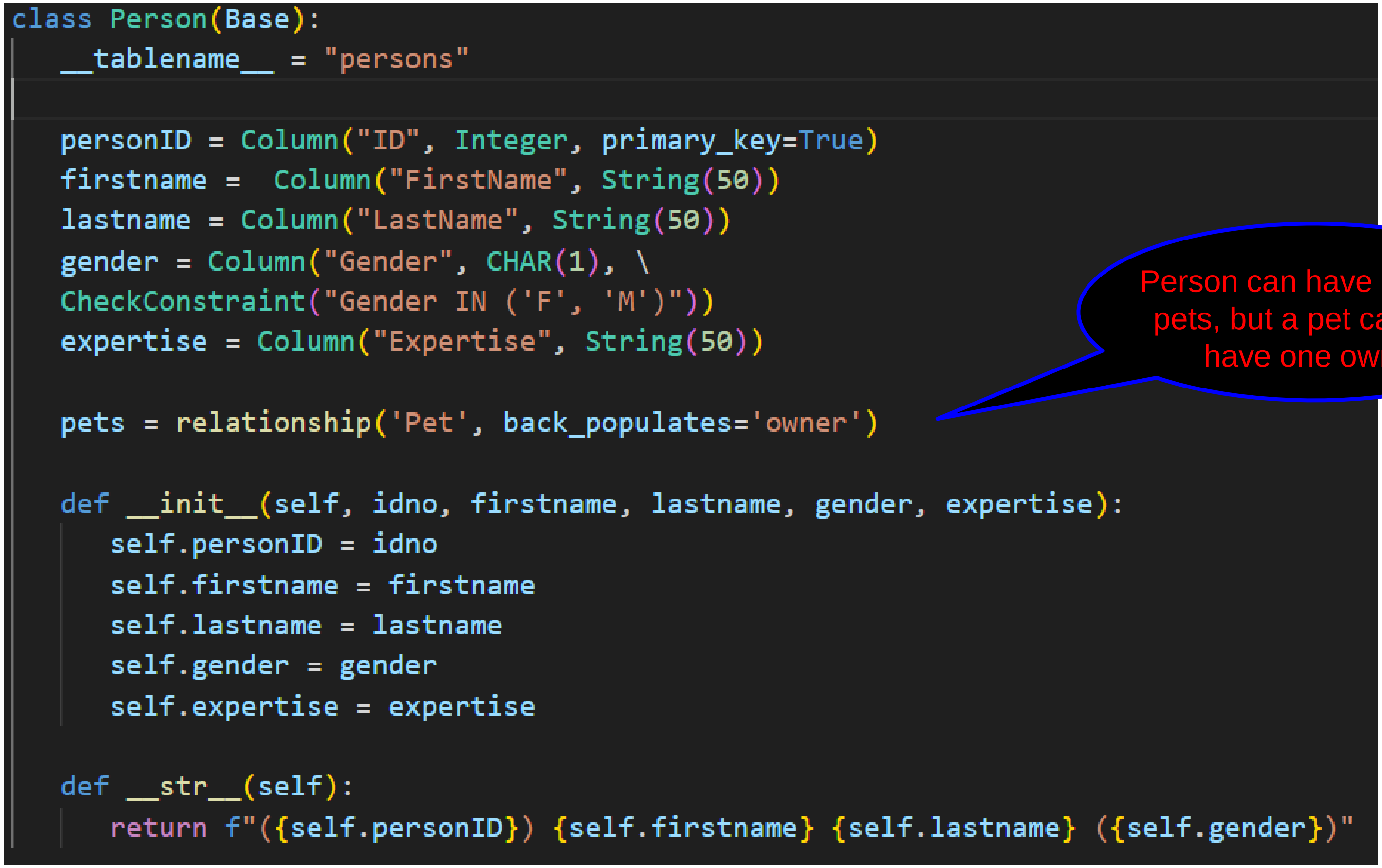
ID INT
FirstName VARCHAR(50)
LastName VARCHAR(50)
Gender CHAR(1)
Expertise VARCHAR(50)

pets

ID INT
NAME VARCHAR(50)
OWNER INT

Creating Relationships between Tables (2)

1. Create the person class

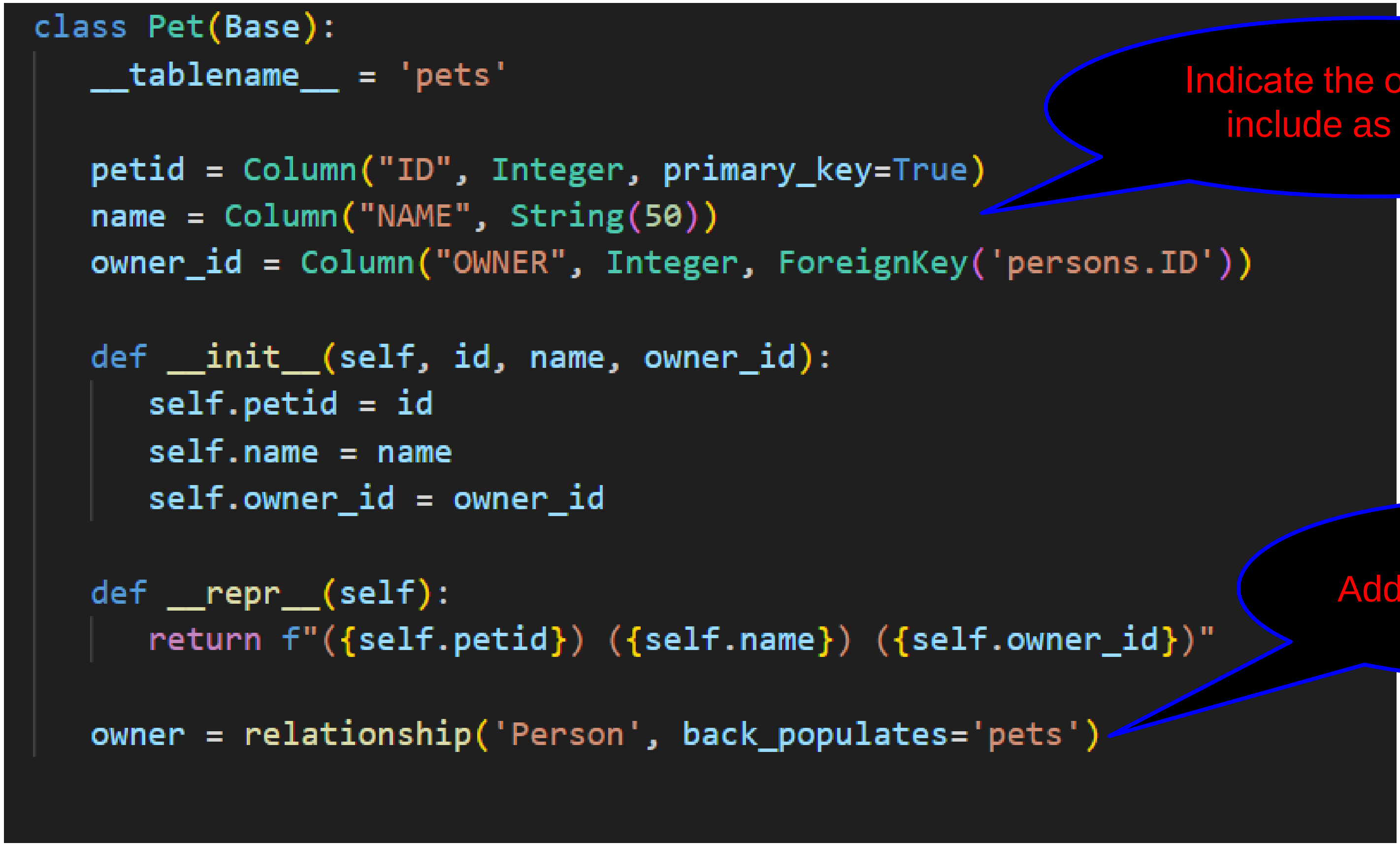


```
class Person(Base):
    __tablename__ = "persons"

    personID = Column("ID", Integer, primary_key=True)
    firstname =  Column("FirstName", String(50))
    lastname = Column("LastName", String(50))
    gender = Column("Gender", CHAR(1), \
    CheckConstraint("Gender IN ('F', 'M')"))
    expertise = Column("Expertise", String(50))

    pets = relationship('Pet', back_populates='owner')

    def __init__(self, idno, firstname, lastname, gender, expertise):
        self.personID = idno
        self.firstname = firstname
        self.lastname = lastname
        self.gender = gender
        self.expertise = expertise

    def __str__(self):
        return f"({self.personID}) {self.firstname} {self.lastname} ({self.gender})"
```

Person can have multiple pets, but a pet can only have one owner

Creating Relationships between Tables (3)

1. Create the person class

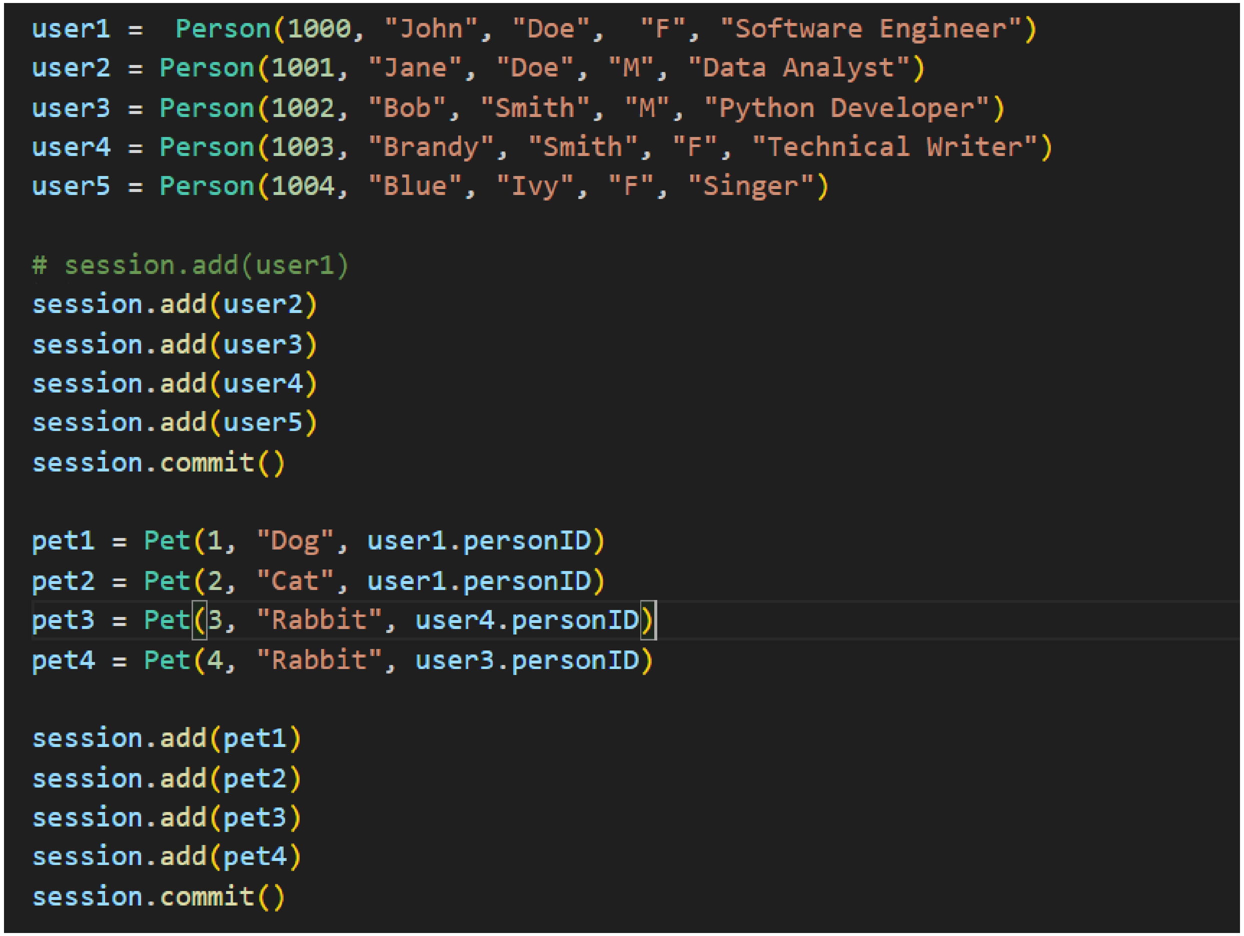


```
class Pet(Base):
    __tablename__ = 'pets'

    petid = Column("ID", Integer, primary_key=True)
    name = Column("NAME", String(50))
    owner_id = Column("OWNER", Integer, ForeignKey('persons.ID'))

    def __init__(self, id, name, owner_id):
        self.petid = id
        self.name = name
        self.owner_id = owner_id

    def __repr__(self):
        return f"({self.petid}) ({self.name}) ({self.owner_id})"

    owner = relationship('Person', back_populates='pets')
```

Indicate the owner’s id and include as foreign key

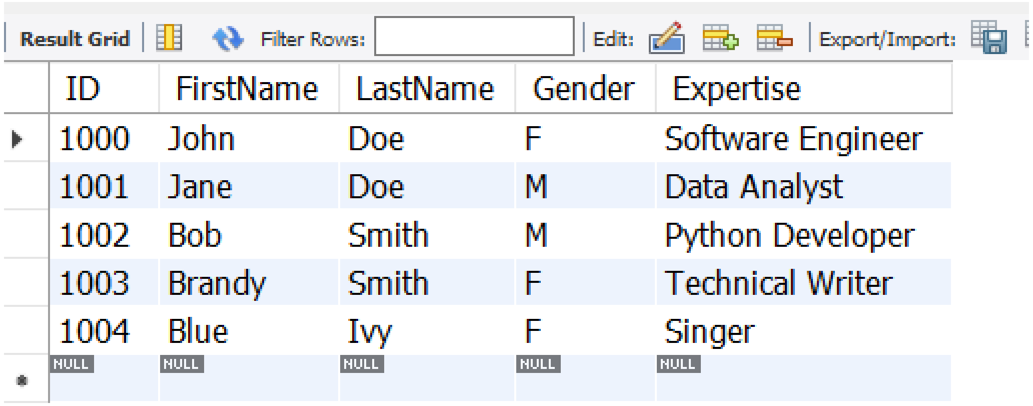
Add the relationship

Populating the tables

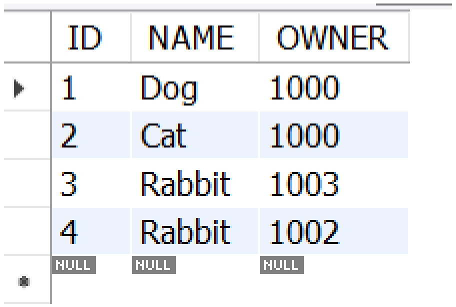


```
user1 =  Person(1000, "John", "Doe",  "F", "Software Engineer")
user2 = Person(1001, "Jane", "Doe", "M", "Data Analyst")
user3 = Person(1002, "Bob", "Smith", "M", "Python Developer")
user4 = Person(1003, "Brandy", "Smith", "F", "Technical Writer")
user5 = Person(1004, "Blue", "Ivy", "F", "Singer")

# session.add(user1)
session.add(user2)
session.add(user3)
session.add(user4)
session.add(user5)
session.commit()

pet1 = Pet(1, "Dog", user1.personID)
pet2 = Pet(2, "Cat", user1.personID)
pet3 = Pet(3, "Rabbit", user4.personID)
pet4 = Pet(4, "Rabbit", user3.personID)

session.add(pet1)
session.add(pet2)
session.add(pet3)
session.add(pet4)
session.commit()
```

Tables in MySQL



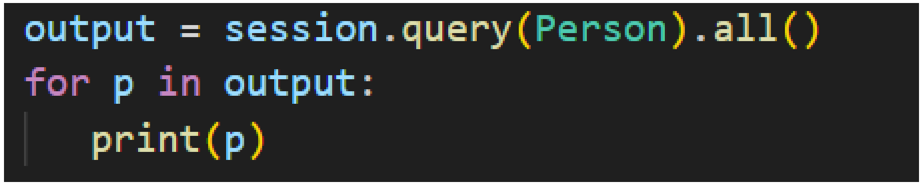
ID	FirstName	LastName	Gender	Expertise
1000	John	Doe	F	Software Engineer
1001	Jane	Doe	M	Data Analyst
1002	Bob	Smith	M	Python Developer
1003	Brandy	Smith	F	Technical Writer
1004	Blue	Ivy	F	Singer
NULL	NULL	NULL	NULL	NULL



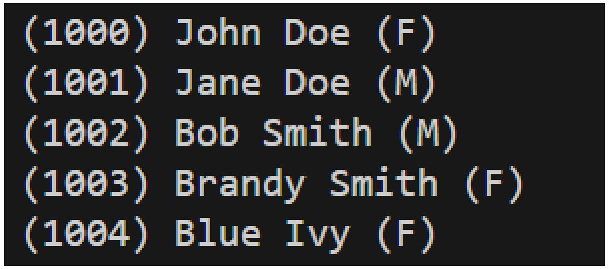
ID	NAME	OWNER
1	Dog	1000
2	Cat	1000
3	Rabbit	1003
4	Rabbit	1002
NULL	NULL	NULL

Querying the Tables (1)

- List all the entries from the “persons” table.



```
output = session.query(Person).all()
for p in output:
    print(p)
```

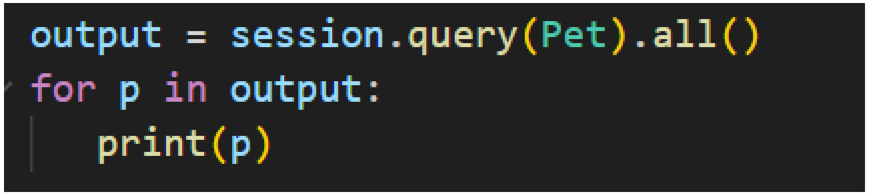


```
(1000) John Doe (F)
(1001) Jane Doe (M)
(1002) Bob Smith (M)
(1003) Brandy Smith (F)
(1004) Blue Ivy (F)
```

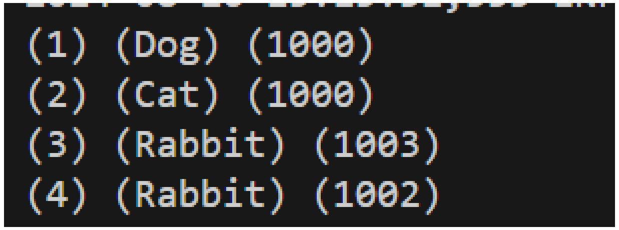
```
SELECT persons.`ID` AS `persons_ID`, persons.`FirstName` AS
`persons_FirstName`, persons.`LastName` AS `persons_LastName`,
persons.`Gender` AS `persons_Gender`, persons.`Expertise` AS `persons_Expertise`
FROM persons
```

Querying the Tables (2)

- List all the entries from the “pets” table.



```
output = session.query(Pet).all()
for p in output:
    print(p)
```

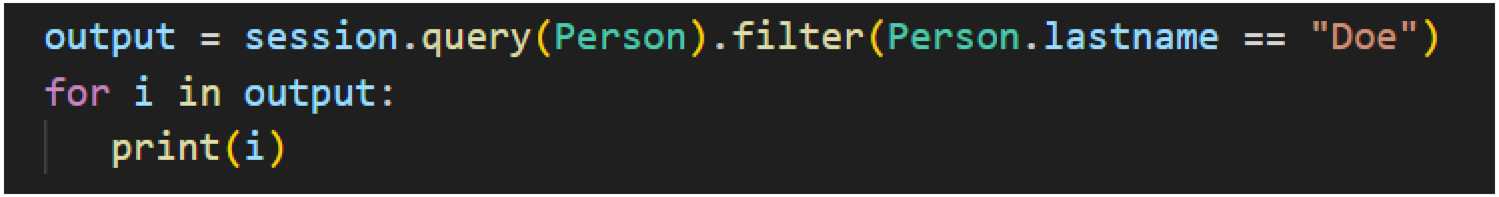


```
(1) (Dog) (1000)
(2) (Cat) (1000)
(3) (Rabbit) (1003)
(4) (Rabbit) (1002)
```

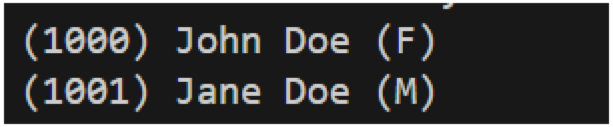
SELECT pets.`ID` AS `pets_ID`,
pets.`NAME` AS `pets_NAME`,
pets.`OWNER` AS `pets_OWNER`
FROM pets

Querying the Tables (3)

- List all persons that have “Doe” as their last name.



```
output = session.query(Person).filter(Person.lastname == "Doe")
for i in output:
    print(i)
```

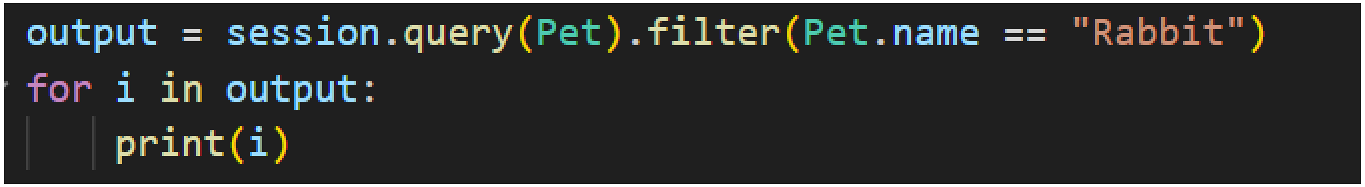


```
(1000) John Doe (F)
(1001) Jane Doe (M)
```

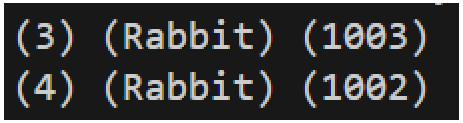
```
SELECT persons.`ID` AS `persons_ID`,
persons.`FirstName` AS `persons_FirstName`,
persons.`LastName` AS `persons_LastName`,
persons.`Gender` AS `persons_Gender`,
persons.`Expertise` AS `persons_Expertise`
FROM persons
WHERE persons.`LastName` = “Doe”
```

Querying the Tables (4)

- Search for all the pets in the database that have the name Rabbit



```
output = session.query(Pet).filter(Pet.name == "Rabbit")
for i in output:
    print(i)
```

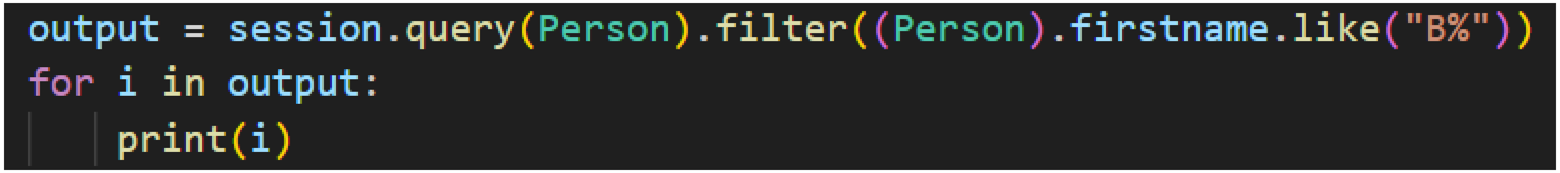


```
(3) (Rabbit) (1003)
(4) (Rabbit) (1002)
```

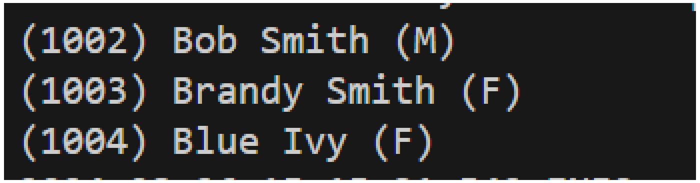
```
SELECT pets.`ID` AS `pets_ID`, pets.`NAME` AS
`pets_NAME`, pets.`OWNER` AS `pets_OWNER`
FROM pets
WHERE pets.`NAME` = “Rabbit”
```

Querying the Tables (5)

- Retrieve all the persons whose firstname starts with the letter “B”.



```python
output = session.query(Person).filter((Person).firstname.like("B%"))
for i in output:
    print(i)
```

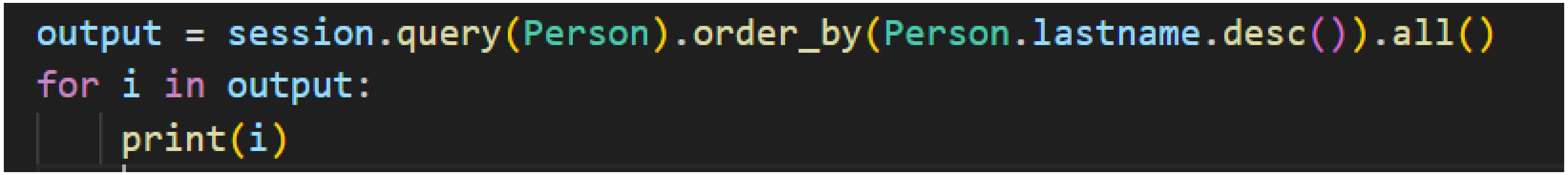


```
(1002) Bob Smith (M)
(1003) Brandy Smith (F)
(1004) Blue Ivy (F)
```

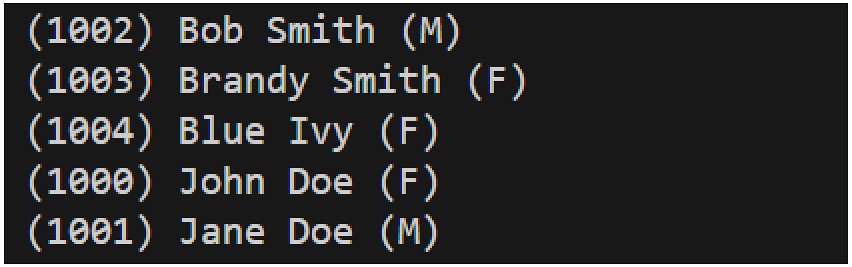
```sql
SELECT persons.`ID` AS `persons_ID`, persons.`FirstName` AS
`persons_FirstName`, persons.`LastName` AS `persons_LastName`,
persons.`Gender` AS `persons_Gender`, persons.`Expertise` AS
`persons_Expertise`
FROM persons
WHERE persons.`FirstName` LIKE “B%”
```

Querying the Tables (6)

- Sort the Person objects by last name in descending order



```python
output = session.query(Person).order_by(Person.lastname.desc()).all()
for i in output:
    print(i)
```

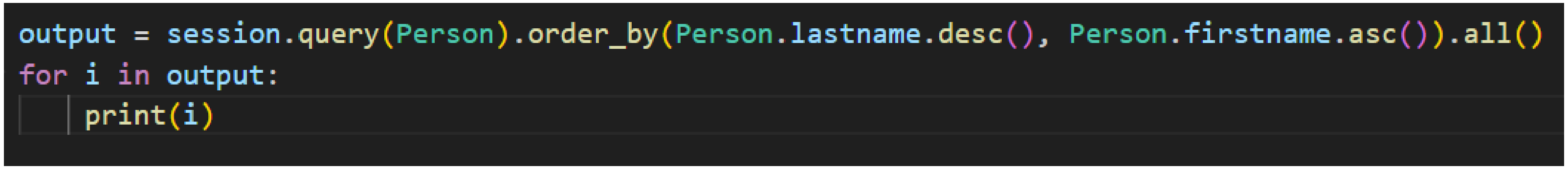


```
(1002) Bob Smith (M)
(1003) Brandy Smith (F)
(1004) Blue Ivy (F)
(1000) John Doe (F)
(1001) Jane Doe (M)
```

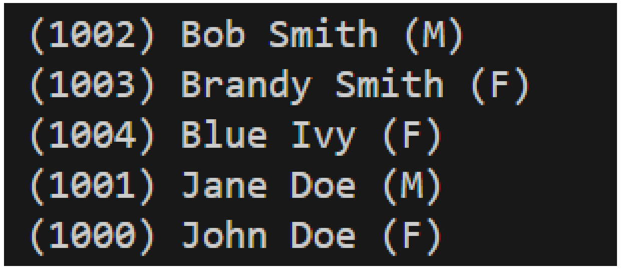
```sql
SELECT persons.`ID` AS `persons_ID`, persons.`FirstName` AS
`persons_FirstName`, persons.`LastName` AS `persons_LastName`,
persons.`Gender` AS `persons_Gender`, persons.`Expertise` AS
`persons_Expertise`
FROM persons ORDER BY persons.`LastName` DESC
```

Querying the Tables (7)

- Sort the Person objects by last name in descending order followed by first name in ascending order.



```
output = session.query(Person).order_by(Person.lastname.desc(), Person.firstname.asc()).all()
for i in output:
    print(i)
```

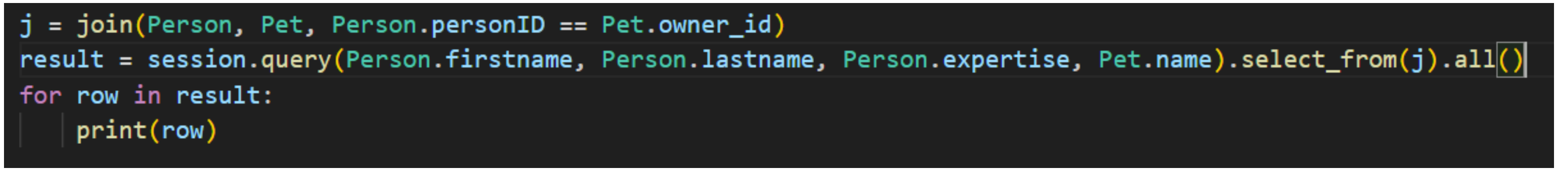


```
(1002) Bob Smith (M)
(1003) Brandy Smith (F)
(1004) Blue Ivy (F)
(1001) Jane Doe (M)
(1000) John Doe (F)
```

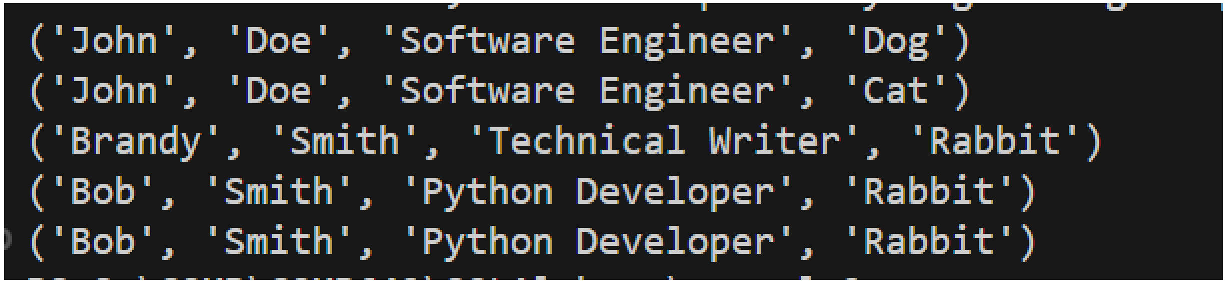
SELECT persons.`ID` AS `persons_ID`, persons.`FirstName` AS `persons_FirstName`, persons.`LastName` AS `persons_LastName`, persons.`Gender` AS `persons_Gender`, persons.`Expertise` AS `persons_Expertise`

Querying the Tables (8)

- Join the Person and Pet tables and select data from both



```python
j = join(Person, Pet, Person.personID == Pet.owner_id)
result = session.query(Person.firstname, Person.lastname, Person.expertise, Pet.name).select_from(j).all()
for row in result:
    print(row)
```



```
('John', 'Doe', 'Software Engineer', 'Dog')
('John', 'Doe', 'Software Engineer', 'Cat')
('Brandy', 'Smith', 'Technical Writer', 'Rabbit')
('Bob', 'Smith', 'Python Developer', 'Rabbit')
('Bob', 'Smith', 'Python Developer', 'Rabbit')
```

```sql
SELECT persons.`FirstName` AS `persons_FirstName`, persons.`LastName`
AS `persons_LastName`, persons.`Expertise` AS `persons_Expertise`, pets.`NAME`
AS `pets_NAME`
FROM persons INNER JOIN pets ON persons.`ID` = pets.`OWNER`
```

Inheritance with SQLAlchemy ORM

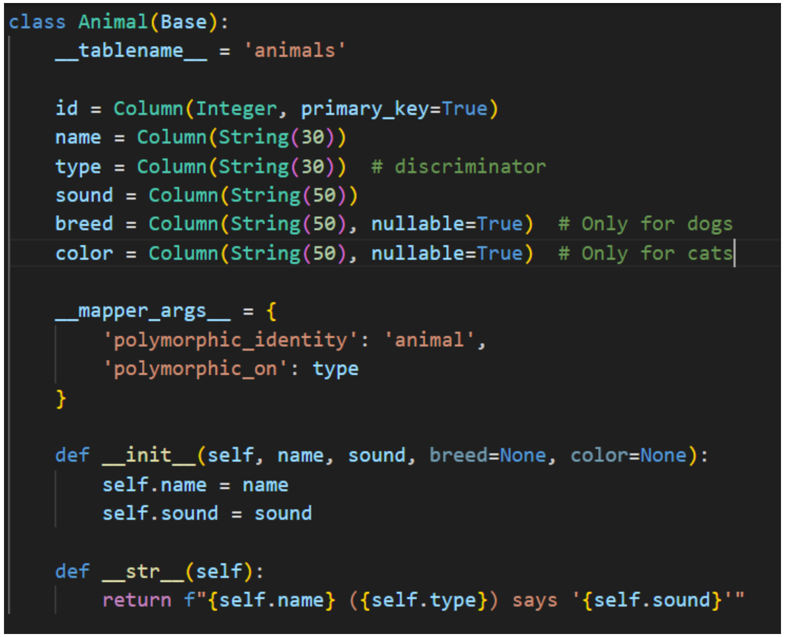
Choosing the right inheritance strategy depends on factors such as performance requirements, query complexity, and how frequently the schema will change.

- Single Table Inheritance (STI)
- Joined Table Inheritance (JTI)
- Concrete Table Inheritance (CTI)

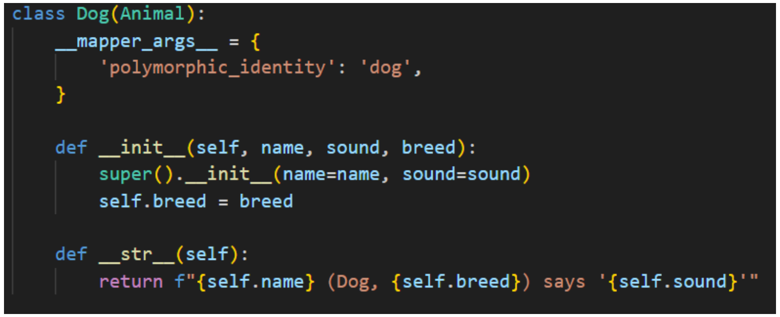
Single Table Inheritance (1)

- A single database table is used to represent an entire class hierarchy.
- All classes classes in the hierarchy are mapped to the same table.
- A special column called a "discriminator" column is used to differentiate between the different subclasses.
- Easy to implement and manage with fewer joins required (simple).
- Query performance is better since all data in one table.
- Columns for subclass specific attributes may be null (wasted space).
- As table grows with more subclasses, queries can become complex and less efficient.

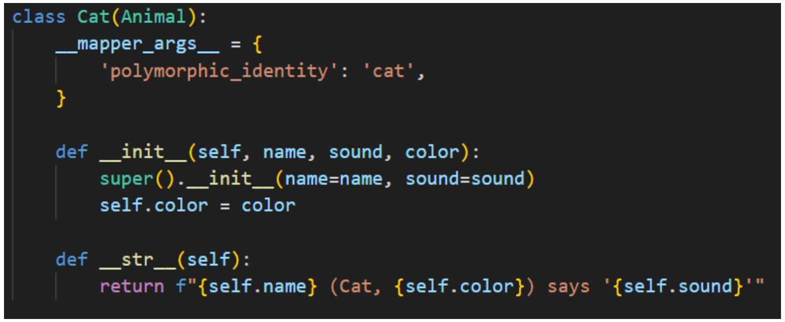
Single Table Inheritance (2)



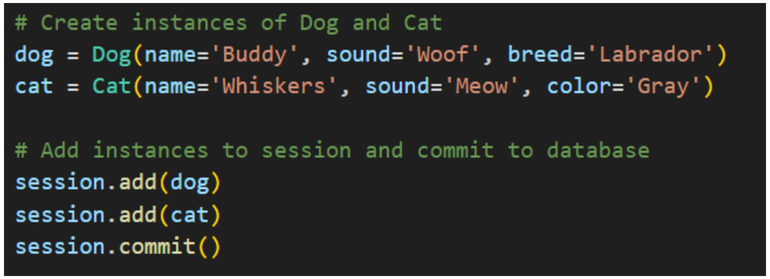
```
class Animal(Base):
    __tablename__ = 'animals'

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    type = Column(String(30))  # discriminator
    sound = Column(String(50))
    breed = Column(String(50), nullable=True)  # Only for dogs
    color = Column(String(50), nullable=True)  # Only for cats

    __mapper_args__ = {
        'polymorphic_identity': 'animal',
        'polymorphic_on': type
    }

    def __init__(self, name, sound, breed=None, color=None):
        self.name = name
        self.sound = sound

    def __str__(self):
        return f"{self.name} ({self.type}) says '{self.sound}'"
```



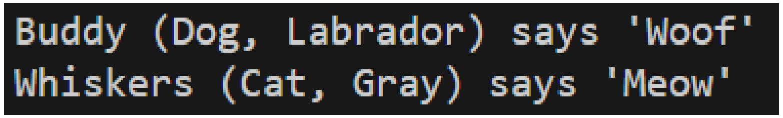
```
class Dog(Animal):
    __mapper_args__ = {
        'polymorphic_identity': 'dog',
    }

    def __init__(self, name, sound, breed):
        super().__init__(name=name, sound=sound)
        self.breed = breed

    def __str__(self):
        return f"{self.name} (Dog, {self.breed}) says '{self.sound}'"
```



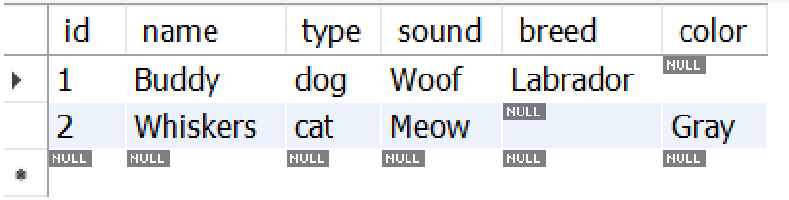
```
class Cat(Animal):
    __mapper_args__ = {
        'polymorphic_identity': 'cat',
    }

    def __init__(self, name, sound, color):
        super().__init__(name=name, sound=sound)
        self.color = color

    def __str__(self):
        return f"{self.name} (Cat, {self.color}) says '{self.sound}'"
```



```
# Create instances of Dog and Cat
dog = Dog(name='Buddy', sound='Woof', breed='Labrador')
cat = Cat(name='Whiskers', sound='Meow', color='Gray')

# Add instances to session and commit to database
session.add(dog)
session.add(cat)
session.commit()
```



```
Buddy (Dog, Labrador) says 'Woof'
Whiskers (Cat, Gray) says 'Meow'
```

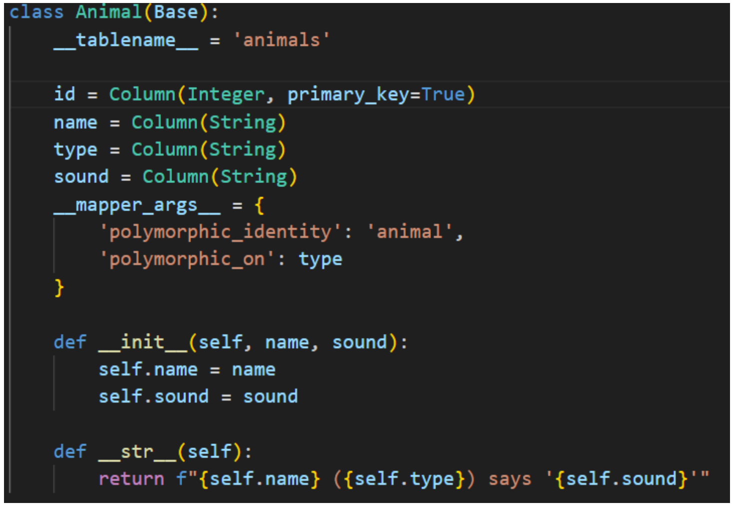


	id	name	type	sound	breed	color
▶	1	Buddy	dog	Woof	Labrador	NULL
	2	Whiskers	cat	Meow	NULL	Gray
✱	NULL	NULL	NULL	NULL	NULL	NULL

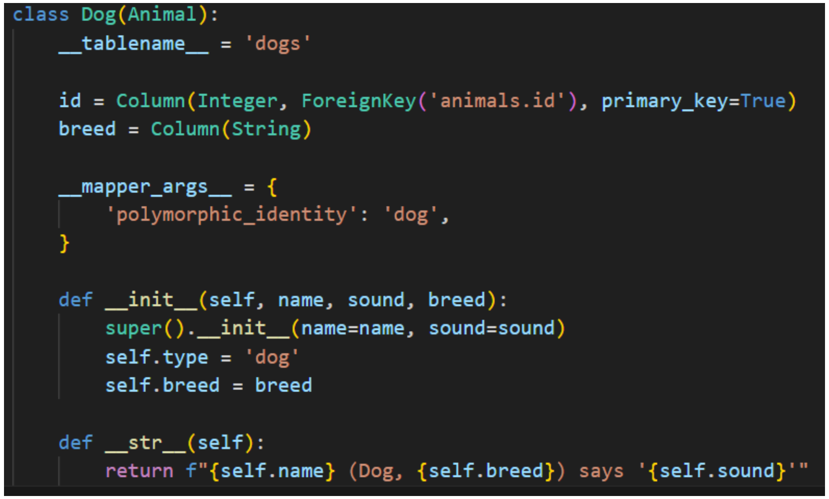
Joined Table Inheritance (1)

- Each class in the hierarchy is mapped to a separate table, with relationship between them.
- Base table contains columns for the base class properties.
- Each subclass has its own table that contains additional properties specific to the subclass and a foreign key references the base table.
- Avoid null columns and redundancy (normalised).
- Easier to add new subclass without altering existing tables (flexible).
- Joins are needed to retrieve data, which can complicate query and potentially affect performance.
- Results in higher number of tables in the database.

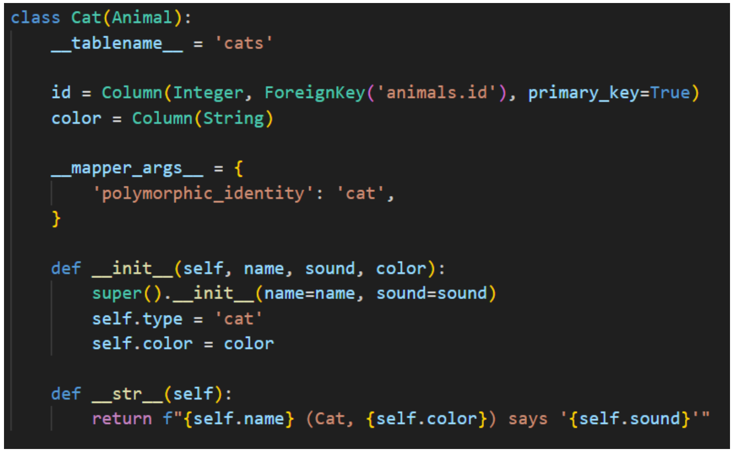
Joined Table Inheritance (2)



```python
class Animal(Base):
    __tablename__ = 'animals'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    type = Column(String)
    sound = Column(String)
    __mapper_args__ = {
        'polymorphic_identity': 'animal',
        'polymorphic_on': type
    }

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

    def __str__(self):
        return f"{self.name} ({self.type}) says '{self.sound}'"
```

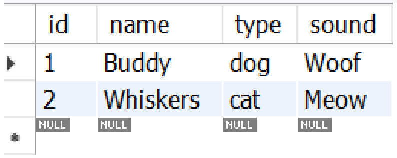


```python
class Dog(Animal):
    __tablename__ = 'dogs'

    id = Column(Integer, ForeignKey('animals.id'), primary_key=True)
    breed = Column(String)

    __mapper_args__ = {
        'polymorphic_identity': 'dog',
    }

    def __init__(self, name, sound, breed):
        super().__init__(name=name, sound=sound)
        self.type = 'dog'
        self.breed = breed

    def __str__(self):
        return f"{self.name} (Dog, {self.breed}) says '{self.sound}'"
```



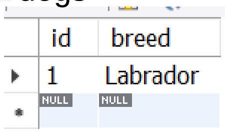
```python
class Cat(Animal):
    __tablename__ = 'cats'

    id = Column(Integer, ForeignKey('animals.id'), primary_key=True)
    color = Column(String)

    __mapper_args__ = {
        'polymorphic_identity': 'cat',
    }

    def __init__(self, name, sound, color):
        super().__init__(name=name, sound=sound)
        self.type = 'cat'
        self.color = color

    def __str__(self):
        return f"{self.name} (Cat, {self.color}) says '{self.sound}'"
```

animals



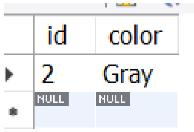
id	name	type	sound
1	Buddy	dog	Woof
2	Whiskers	cat	Meow
NULL	NULL	NULL	NULL

dogs

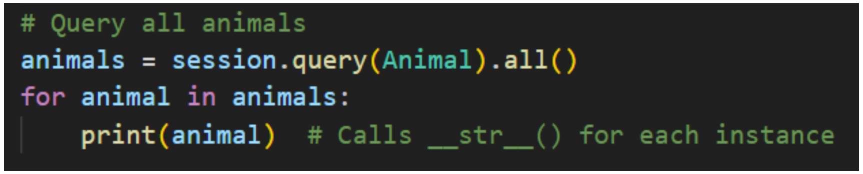


id	breed
1	Labrador
NULL	NULL

cats



id	color
2	Gray
NULL	NULL

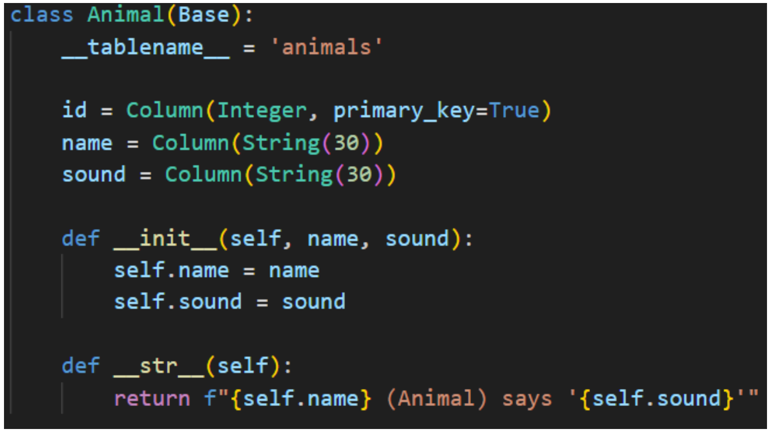


```python
# Query all animals
animals = session.query(Animal).all()
for animal in animals:
    print(animal)  # Calls __str__() for each instance
```

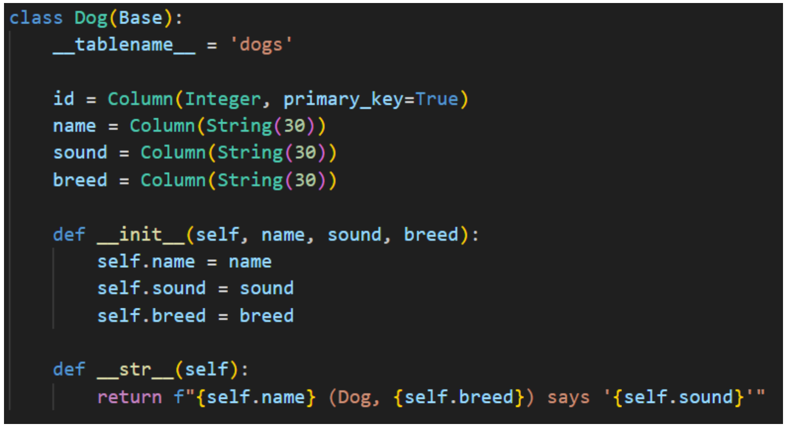
Concrete Table Inheritance (1)

- Each class in the hierarchy is mapped to its own table, which includes properties of both the base class and the subclass.
- Each subclass table contains all the fields for that subclass, including those inherited from the base class.
- No joins are required to retrieve all properties as each table is self-contained (simple queries).
- Can be efficient for queries where only one subclass type is needed.
- Common attributes from the base are duplicated across table (data redundancy).
- Schema changes can be challenging since each subclass tables must be updated separately.

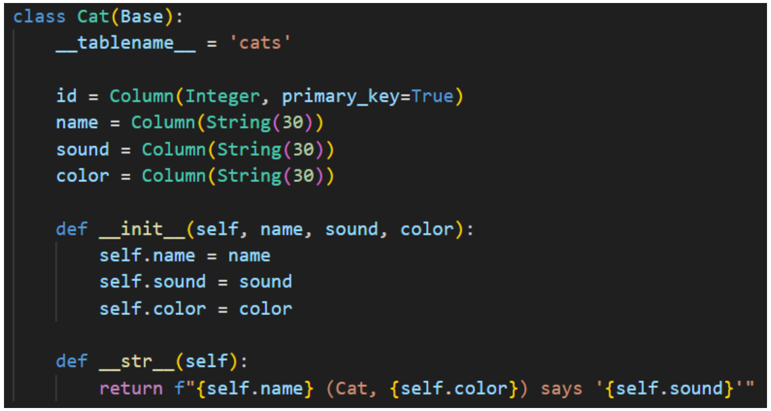
Concrete Table Inheritance (2)



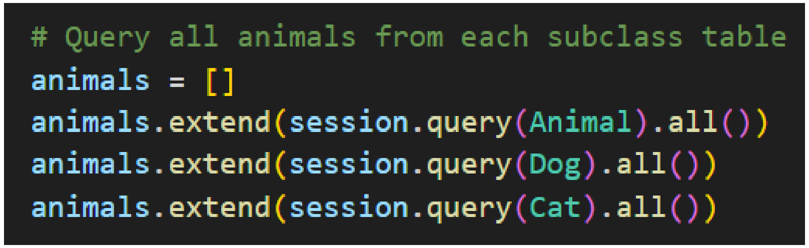
```python
class Animal(Base):
    __tablename__ = 'animals'

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    sound = Column(String(30))

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

    def __str__(self):
        return f"{self.name} (Animal) says '{self.sound}'"
```



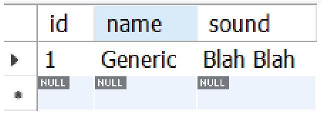
```python
class Dog(Base):
    __tablename__ = 'dogs'

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    sound = Column(String(30))
    breed = Column(String(30))

    def __init__(self, name, sound, breed):
        self.name = name
        self.sound = sound
        self.breed = breed

    def __str__(self):
        return f"{self.name} (Dog, {self.breed}) says '{self.sound}'"
```



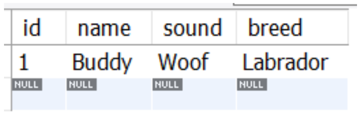
```python
class Cat(Base):
    __tablename__ = 'cats'

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    sound = Column(String(30))
    color = Column(String(30))

    def __init__(self, name, sound, color):
        self.name = name
        self.sound = sound
        self.color = color

    def __str__(self):
        return f"{self.name} (Cat, {self.color}) says '{self.sound}'"
```



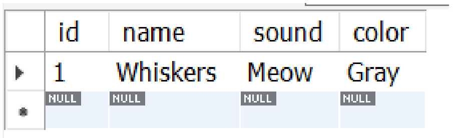
```python
# Query all animals from each subclass table
animals = []
animals.extend(session.query(Animal).all())
animals.extend(session.query(Dog).all())
animals.extend(session.query(Cat).all())
```



	id	name	sound
▸	1	Generic	Blah Blah
*	NULL	NULL	NULL

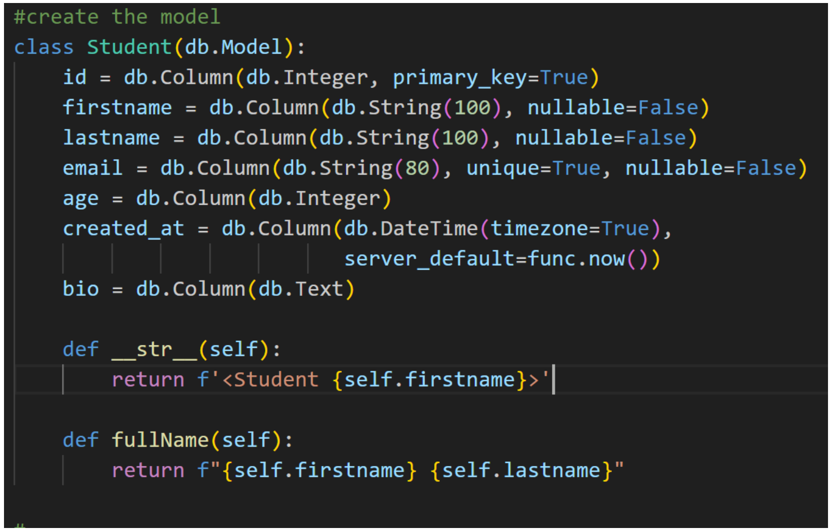


id	name	sound	breed
1	Buddy	Woof	Labrador
NULL	NULL	NULL	NULL

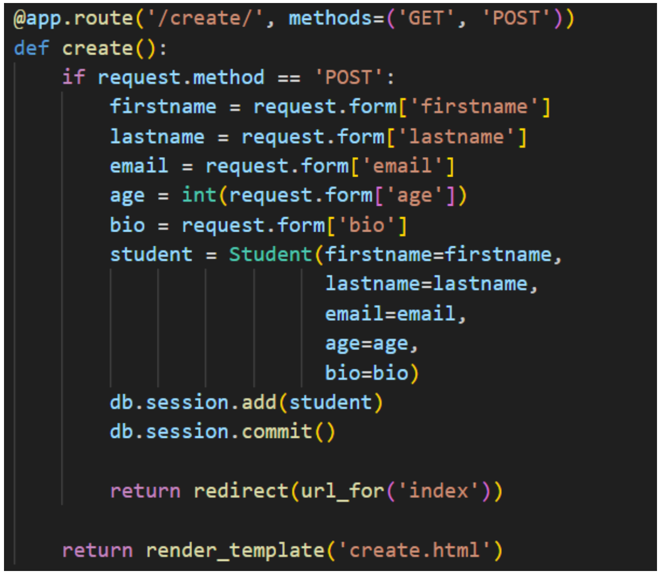


	id	name	sound	color
▸	1	Whiskers	Meow	Gray
*	NULL	NULL	NULL	NULL

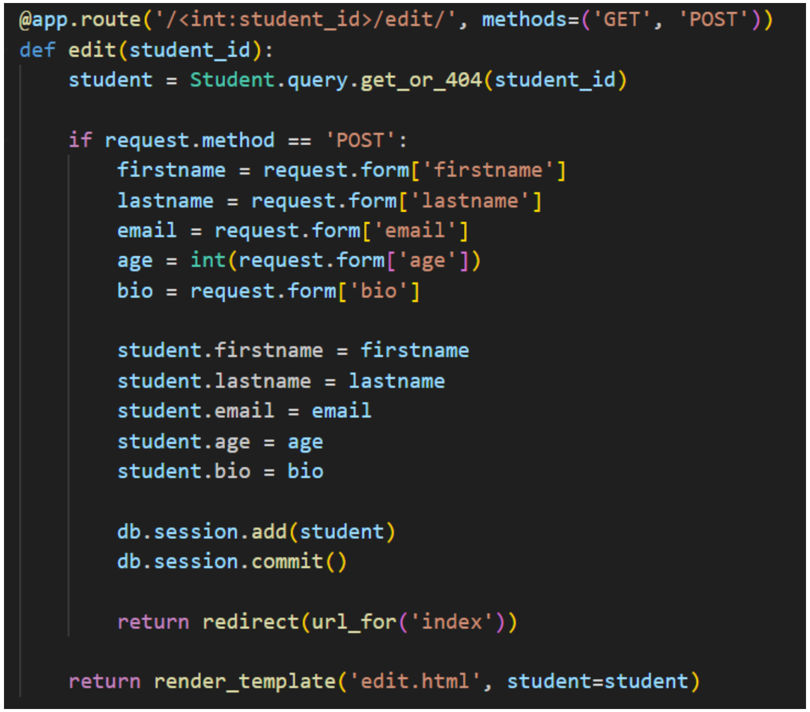
Put it Together (1)



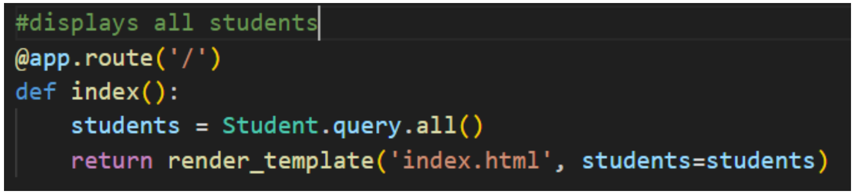
```python
#create the model
class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    firstname = db.Column(db.String(100), nullable=False)
    lastname = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(80), unique=True, nullable=False)
    age = db.Column(db.Integer)
    created_at = db.Column(db.DateTime(timezone=True),
                           server_default=func.now())
    bio = db.Column(db.Text)

    def __str__(self):
        return f'<Student {self.firstname}>'

    def fullName(self):
        return f"{self.firstname} {self.lastname}"
```



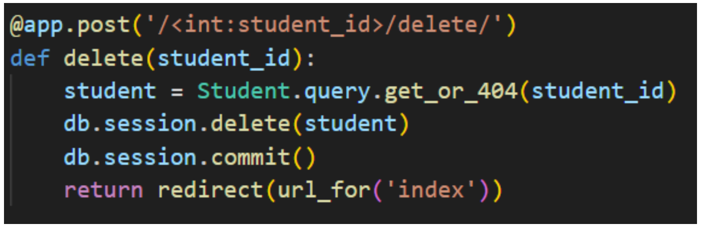
```python
@app.route('/create/', methods=('GET', 'POST'))
def create():
    if request.method == 'POST':
        firstname = request.form['firstname']
        lastname = request.form['lastname']
        email = request.form['email']
        age = int(request.form['age'])
        bio = request.form['bio']
        student = Student(firstname=firstname,
                          lastname=lastname,
                          email=email,
                          age=age,
                          bio=bio)
        db.session.add(student)
        db.session.commit()

        return redirect(url_for('index'))

    return render_template('create.html')
```



```python
@app.route('/<int:student_id>/edit/', methods=('GET', 'POST'))
def edit(student_id):
    student = Student.query.get_or_404(student_id)

    if request.method == 'POST':
        firstname = request.form['firstname']
        lastname = request.form['lastname']
        email = request.form['email']
        age = int(request.form['age'])
        bio = request.form['bio']

        student.firstname = firstname
        student.lastname = lastname
        student.email = email
        student.age = age
        student.bio = bio

        db.session.add(student)
        db.session.commit()

        return redirect(url_for('index'))

    return render_template('edit.html', student=student)
```

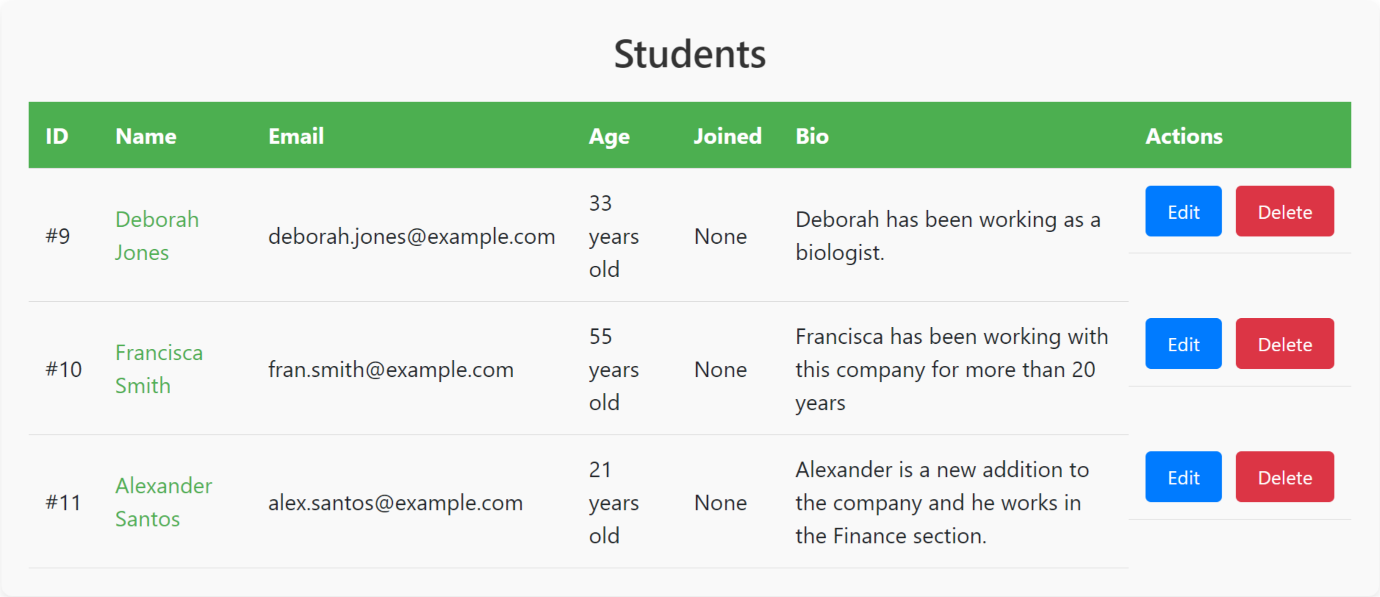


```python
#displays all students
@app.route('/')
def index():
    students = Student.query.all()
    return render_template('index.html', students=students)
```



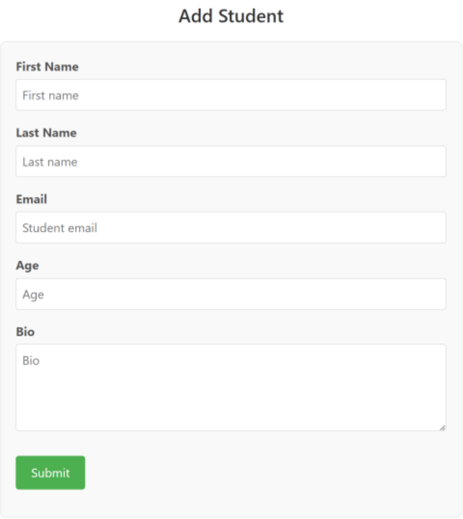
```python
@app.post('/<int:student_id>/delete/')
def delete(student_id):
    student = Student.query.get_or_404(student_id)
    db.session.delete(student)
    db.session.commit()
    return redirect(url_for('index'))
```

Put it Together (2)



Students

ID	Name	Email	Age	Joined	Bio	Actions
#9	Deborah Jones	deborah.jones@example.com	33 years old	None	Deborah has been working as a biologist.	Edit Delete
#10	Francisca Smith	fran.smith@example.com	55 years old	None	Francisca has been working with this company for more than 20 years	Edit Delete
#11	Alexander Santos	alex.santos@example.com	21 years old	None	Alexander is a new addition to the company and he works in the Finance section.	Edit Delete



Add Student

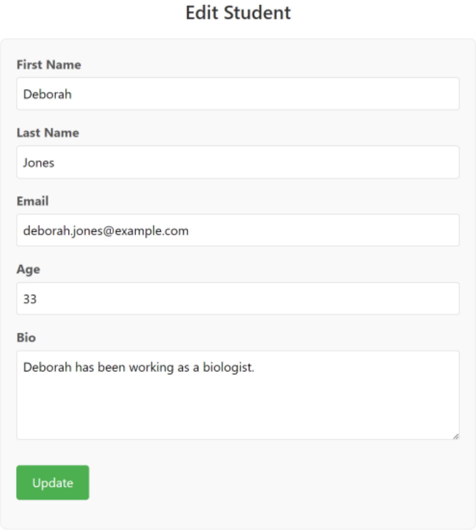
First Name
First name

Last Name
Last name

Email
Student email

Age
Age

Bio
Bio



Edit Student

First Name
Deborah

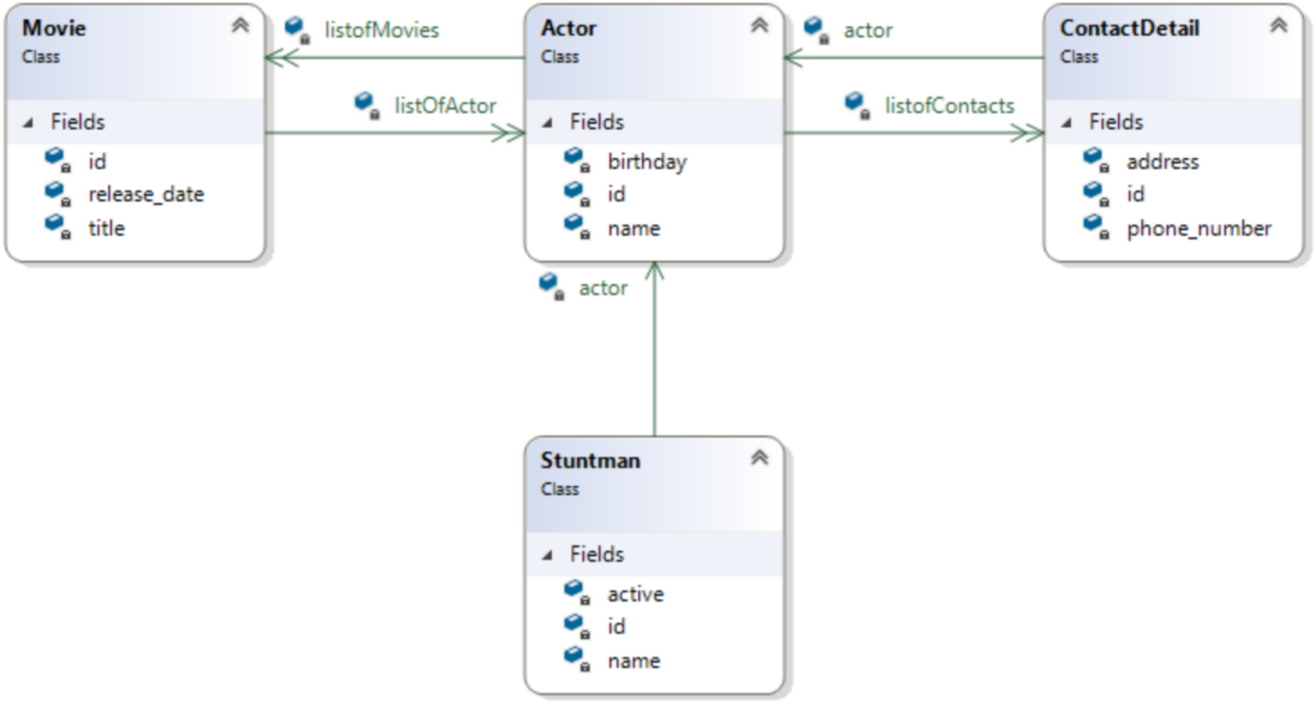
Last Name
Jones

Email
deborah.jones@example.com

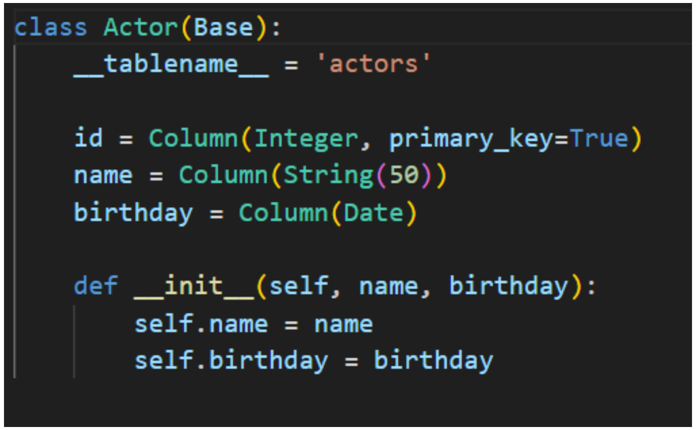
Age
33

Bio
Deborah has been working as a biologist.

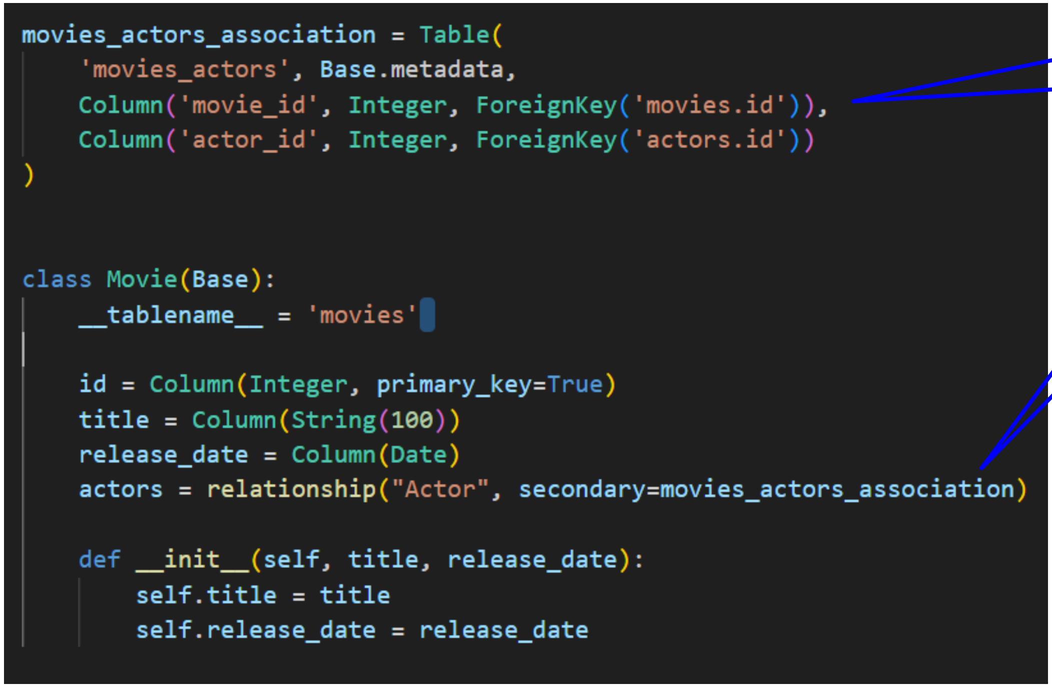
Put it Together (3)



Put it Together (4)



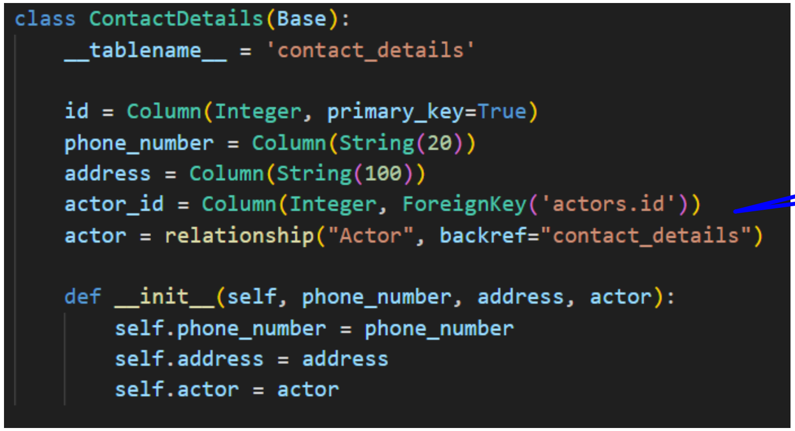
```python
class Actor(Base):
    __tablename__ = 'actors'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    birthday = Column(Date)

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



```python
movies_actors_association = Table(
    'movies_actors', Base.metadata,
    Column('movie_id', Integer, ForeignKey('movies.id')),
    Column('actor_id', Integer, ForeignKey('actors.id'))
)


class Movie(Base):
    __tablename__ = 'movies'

    id = Column(Integer, primary_key=True)
    title = Column(String(100))
    release_date = Column(Date)
    actors = relationship("Actor", secondary=movies_actors_association)

    def __init__(self, title, release_date):
        self.title = title
        self.release_date = release_date
```

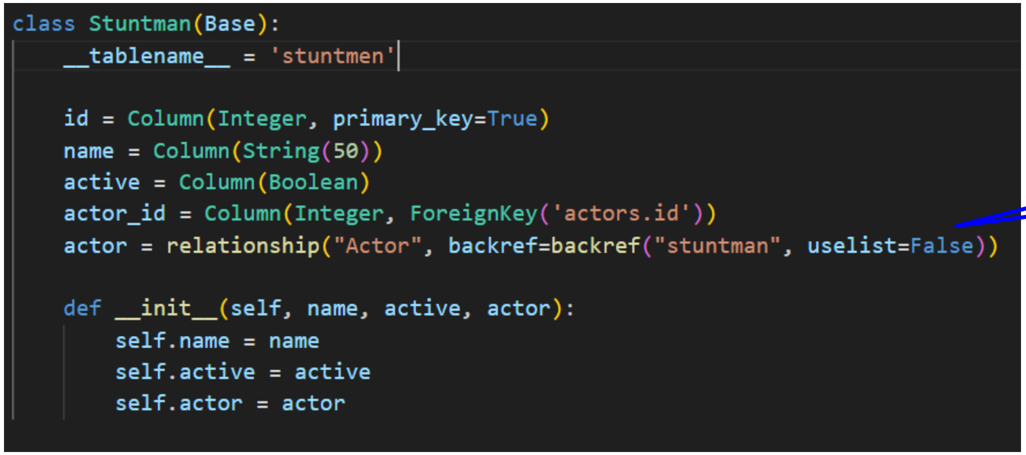
Many to many

Put it Together (5)



```python
class ContactDetails(Base):
    __tablename__ = 'contact_details'

    id = Column(Integer, primary_key=True)
    phone_number = Column(String(20))
    address = Column(String(100))
    actor_id = Column(Integer, ForeignKey('actors.id'))
    actor = relationship("Actor", backref="contact_details")

    def __init__(self, phone_number, address, actor):
        self.phone_number = phone_number
        self.address = address
        self.actor = actor
```

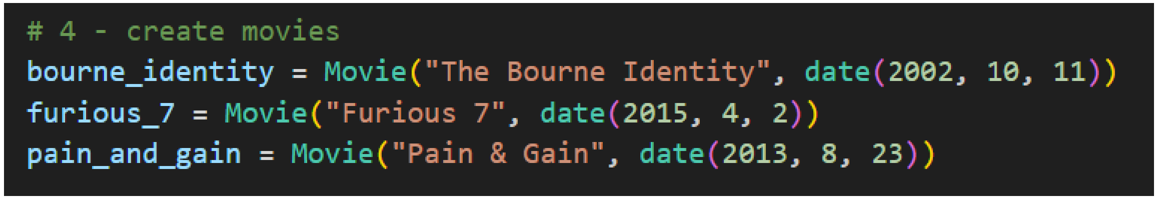
One to many



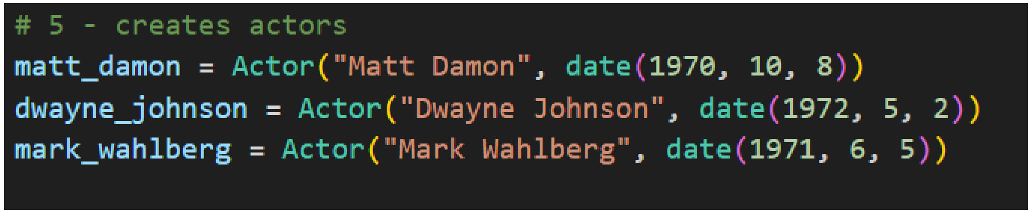
```python
class Stuntman(Base):
    __tablename__ = 'stuntmen'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    active = Column(Boolean)
    actor_id = Column(Integer, ForeignKey('actors.id'))
    actor = relationship("Actor", backref=backref("stuntman", uselist=False))

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

One to one

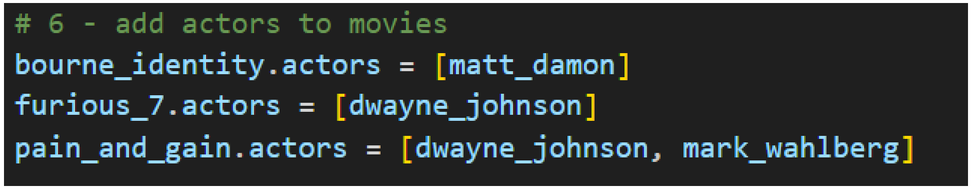
Put it Together (6)



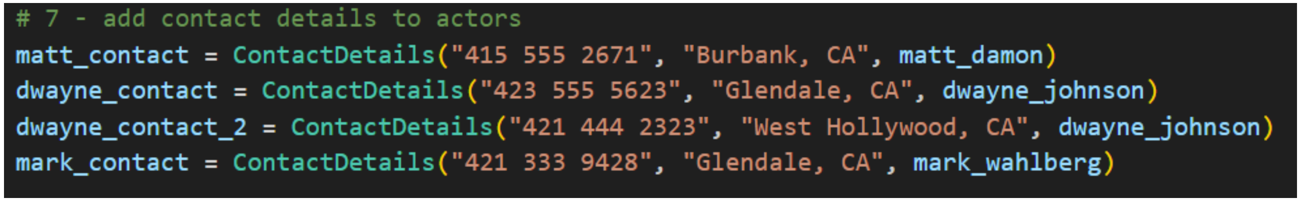
```python
# 4 - create movies
bourne_identity = Movie("The Bourne Identity", date(2002, 10, 11))
furious_7 = Movie("Furious 7", date(2015, 4, 2))
pain_and_gain = Movie("Pain & Gain", date(2013, 8, 23))
```



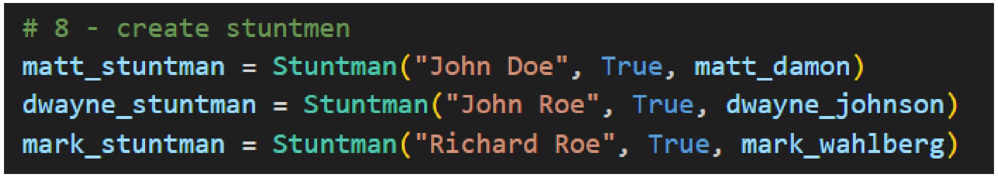
```python
# 5 - creates actors
matt_damon = Actor("Matt Damon", date(1970, 10, 8))
dwayne_johnson = Actor("Dwayne Johnson", date(1972, 5, 2))
mark_wahlberg = Actor("Mark Wahlberg", date(1971, 6, 5))
```



```python
# 6 - add actors to movies
bourne_identity.actors = [matt_damon]
furious_7.actors = [dwayne_johnson]
pain_and_gain.actors = [dwayne_johnson, mark_wahlberg]
```



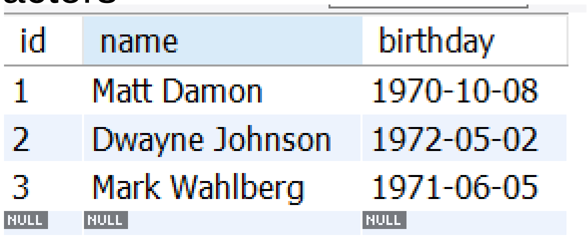
```python
# 7 - add contact details to actors
matt_contact = ContactDetails("415 555 2671", "Burbank, CA", matt_damon)
dwayne_contact = ContactDetails("423 555 5623", "Glendale, CA", dwayne_johnson)
dwayne_contact_2 = ContactDetails("421 444 2323", "West Hollywood, CA", dwayne_johnson)
mark_contact = ContactDetails("421 333 9428", "Glendale, CA", mark_wahlberg)
```



```python
# 8 - create stuntmen
matt_stuntman = Stuntman("John Doe", True, matt_damon)
dwayne_stuntman = Stuntman("John Roe", True, dwayne_johnson)
mark_stuntman = Stuntman("Richard Roe", True, mark_wahlberg)
```

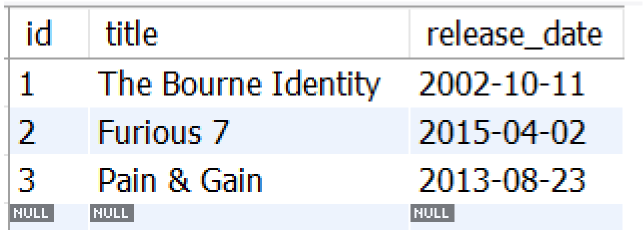
Put it Together (7)

actors



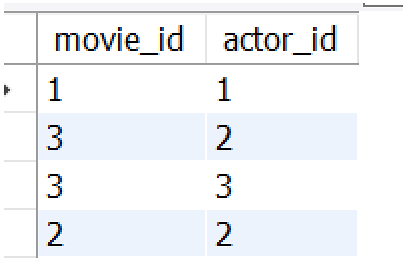
id	name	birthday
1	Matt Damon	1970-10-08
2	Dwayne Johnson	1972-05-02
3	Mark Wahlberg	1971-06-05
NULL	NULL	NULL

movies



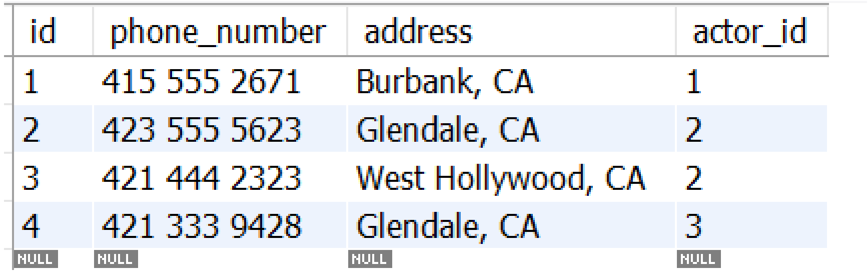
id	title	release_date
1	The Bourne Identity	2002-10-11
2	Furious 7	2015-04-02
3	Pain & Gain	2013-08-23
NULL	NULL	NULL

movies-actors



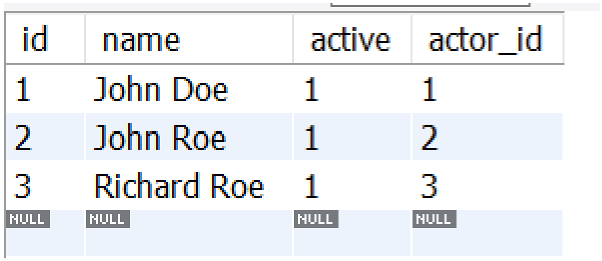
movie_id	actor_id
1	1
3	2
3	3
2	2

contact_details



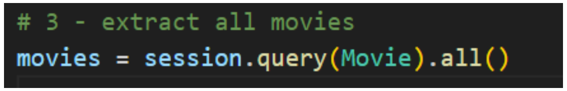
id	phone_number	address	actor_id
1	415 555 2671	Burbank, CA	1
2	423 555 5623	Glendale, CA	2
3	421 444 2323	West Hollywood, CA	2
4	421 333 9428	Glendale, CA	3
NULL	NULL	NULL	NULL

stuntmen

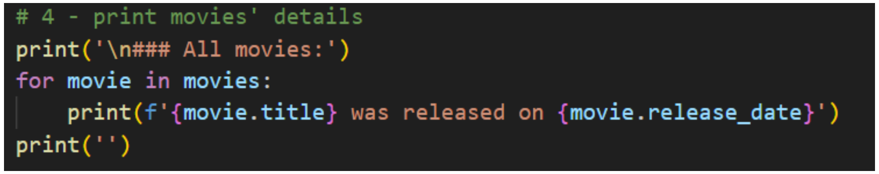


id	name	active	actor_id
1	John Doe	1	1
2	John Roe	1	2
3	Richard Roe	1	3
NULL	NULL	NULL	NULL

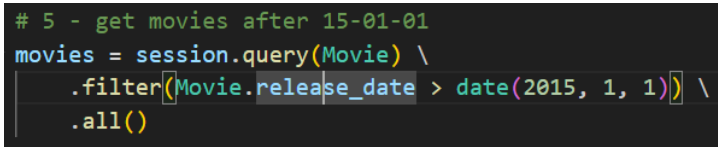
Put it Together (8)



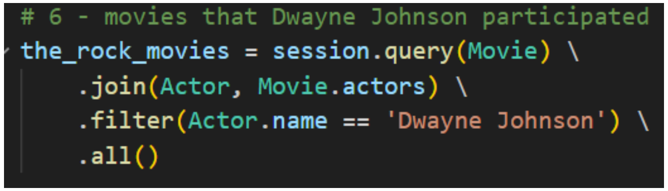
```python
# 3 - extract all movies
movies = session.query(Movie).all()
```



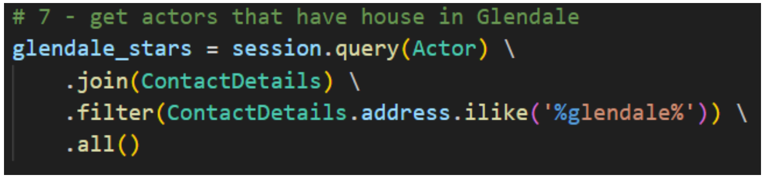
```python
# 4 - print movies' details
print('\n### All movies:')
for movie in movies:
    print(f'{movie.title} was released on {movie.release_date}')
print('')
```



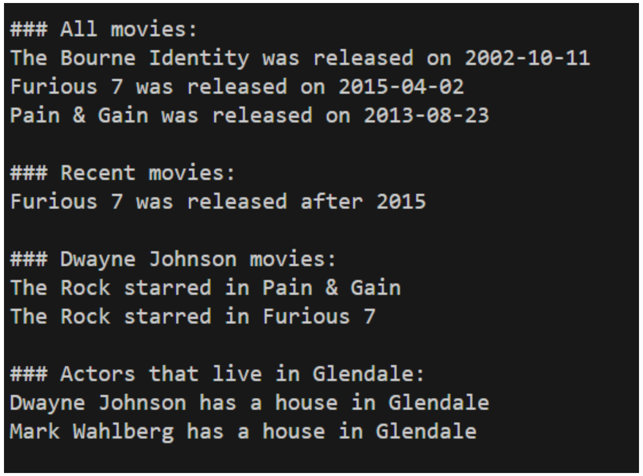
```python
# 5 - get movies after 15-01-01
movies = session.query(Movie) \
    .filter(Movie.release_date > date(2015, 1, 1)) \
    .all()
```



```python
# 6 - movies that Dwayne Johnson participated
the_rock_movies = session.query(Movie) \
    .join(Actor, Movie.actors) \
    .filter(Actor.name == 'Dwayne Johnson') \
    .all()
```



```python
# 7 - get actors that have house in Glendale
glendale_stars = session.query(Actor) \
    .join(ContactDetails) \
    .filter(ContactDetails.address.ilike('%glendale%')) \
    .all()
```



```
### All movies:
The Bourne Identity was released on 2002-10-11
Furious 7 was released on 2015-04-02
Pain & Gain was released on 2013-08-23

### Recent movies:
Furious 7 was released after 2015

### Dwayne Johnson movies:
The Rock starred in Pain & Gain
The Rock starred in Furious 7

### Actors that live in Glendale:
Dwayne Johnson has a house in Glendale
Mark Wahlberg has a house in Glendale
```