Topic 05: ORM with Peewee

Overview

This topic introduces **Object-Relational Mapping (ORM)** using **Peewee**, a lightweight Python ORM. An ORM eliminates the need to write SQL by mapping database tables to Python classes and rows to objects. This represents a paradigm shift from manipulating data with strings (SQL) to working with objects using familiar Python syntax. While ORMs have trade-offs, they dramatically simplify common database operations and reduce boilerplate code.

What is an ORM?

Object-Relational Mapping (ORM) bridges the gap between object-oriented programming and relational databases.

Without ORM (Topic 04):

```
cursor.execute("""
    SELECT pet.id, pet.name, pet.age, pet.owner, kind.name as kind_name
    FROM pet
    JOIN kind ON pet.kind_id = kind.id
""")
pets = cursor.fetchall()
pets = [dict(pet) for pet in pets]
```

With ORM (Topic 05):

```
pets = Pet.select().join(Kind)
pets = list(pets)
```

What ORMs do:

- Map database tables to Python classes
- Map rows to object instances
- Generate SQL automatically
- Handle relationships between tables
- Provide Pythonic query syntax
- Manage database connections
- Create and update schemas

Why Peewee?

Peewee is chosen for this course because it's:

- **Simple**: Minimal API, easy to learn

- **Lightweight**: Small codebase, few dependencies
- **Expressive**: Pythonic syntax feels natural
- **Feature-complete**: Supports joins, transactions, migrations
- **Well-documented**: Clear examples and guides
- **SQLite-friendly**: Perfect for learning and small projects

Other popular Python ORMs:

- **SQLAlchemy**: More powerful, steeper learning curve
- **Django ORM**: Tied to Django framework
- **Tortoise ORM**: Async support for modern apps

Peewee hits the sweet spot for learning ORM concepts.

Model Definition - From Tables to Classes

Before: SQL Table Definition

```
create table kind (
    id integer primary key autoincrement,
    name text not null,
    food text,
    sound text
);
```

After: Peewee Model Class

```
class Kind(Model):
    kind_name = CharField()
    food = CharField()
    noise = CharField()

    class Meta:
        database = db
```

Key transformations:

- **Table -> Class**: `kind` table becomes `Kind` class
- **Column -> Attribute**: Each column becomes a class attribute
- **Data type -> Field type**: `text` becomes `CharField()`, `integer` becomes `IntegerField()`
- **Constraints**: Handled by field parameters

Field Types

Peewee provides field types that map to database column types:

Database Type	Peewee Field	Example

Database Type	Peewee Field	Example
text/varchar	`CharField()`	names, strings
integer	`IntegerField()`	ages, counts
real/float	`FloatField()`	prices, measurements
boolean	`BooleanField()`	flags, yes/no
date	`DateField()`	birth dates
datetime	`DateTimeField()`	timestamps
blob	`BlobField()`	binary data

Example:

```
class Pet(Model):
    name = CharField()           # text
    age = IntegerField()         # integer
    owner = CharField()          # text
    is_active = BooleanField()   # boolean (not in our example)
```

Foreign Key Relationships

Before: SQL Foreign Key

```
create table pet (
    id integer primary key autoincrement,
    name text not null,
    kind_id integer not null,
    foreign key (kind_id) references kind(id)
);
```

After: Peewee ForeignKeyField

```
class Pet(Model):
    name = CharField()
    age = IntegerField()
    owner = CharField()
    kind = ForeignKeyField(Kind, backref="pets")

    class Meta:
        database = db
```

Breaking it down:

- `kind = ForeignKeyField(Kind, ...)` - Creates relationship to Kind model
- `Kind` - The related model (parent table)
- `backref="pets"` - Creates reverse relationship (Kind.pets returns all pets of that kind)
- Automatically creates `kind_id` column in database
- Returns Kind object, not just ID

The backref magic:

```
# Forward relationship (pet -> kind)
pet = Pet.get_by_id(1)
print(pet.kind.kind_name)  # Access related Kind object

# Reverse relationship (kind -> pets)
kind = Kind.get_by_id(1)
for pet in kind.pets:  # Iterate through all pets of this kind
    print(pet.name)
```

Database Initialization

```
db = None  # Global database connection

def initialize(database_file):
    global db, Kind, Pet
    db = SqliteDatabase(database_file)

    Kind._meta.database = db
    Pet._meta.database = db

    db.connect()
    db.create_tables([Pet, Kind])
```

What this does:

1. **Create connection**: `SqliteDatabase(database_file)` connects to SQLite file
2. **Bind models**: Assign database to each model's metadata
3. **Connect**: Open the database connection
4. **Create tables**: Automatically generate table schemas from models

Key insight: `db.create_tables()` is idempotent - safe to call multiple times. It only creates tables that don't exist.

No more manual SQL for schema creation!

CRUD Operations - The ORM Way

Create (Insert)

Before (SQL):

```
cursor.execute(
    """insert into kind(name, food, sound) values (?,?,?)""",
    (data["name"], data["food"], data["sound"])
)
connection.commit()
```

After (Peewee):

```
kind = Kind(kind_name="dog", food="dog_food", noise="bark")
kind.save()
```

Alternative syntax:

```
kind = Kind.create(kind_name="dog", food="dog_food", noise="bark")
```

Key differences:

- Create object like any Python class
- Call `.save()` to persist to database
- Or use `.create()` to instantiate and save in one step
- No SQL strings
- No manual commit (handled automatically)

Creating with relationships:

```
kind = Kind(kind_name="dog", food="dog_food", noise="bark")
kind.save()

pet = Pet(name="Dorothy", age=10, owner="Greg", kind=kind)
pet.save()
```

Notice: Pass the `kind` object, not `kind.id`. Peewee handles the foreign key automatically.

Read (Select)

Get all records:

Before (SQL):

```
cursor.execute("select * from kind")
kinds = cursor.fetchall()
kinds = [dict(kind) for kind in kinds]
```

After (Peewee):

```
def get_kinds():
    kinds = Kind.select()
    return list(kinds)
```

Breaking it down:

- `Kind.select()` - Returns a query object (like `SELECT * FROM kind`)
- `list(kinds)` - Executes query and converts to list
- Each element is a `Kind` object, not a dict or tuple

Get by ID:

Before (SQL):

```
cursor.execute("select * from pet where id = ?", (id,))
pet = cursor.fetchone()
```

After (Peewee):

```
def get_pet_by_id(id):
    pet = Pet.get_or_none(Pet.id == id)
    return pet
```

Alternative methods:

```
# Raises exception if not found
pet = Pet.get_by_id(1)

# Returns None if not found (safer)
pet = Pet.get_or_none(Pet.id == 1)

# Using where clause
pet = Pet.select().where(Pet.id == 1).first()
```

Query with conditions:

```
# Get all dogs
dogs = Pet.select().where(Pet.kind.kind_name == "dog")

# Get pets older than 5
old_pets = Pet.select().where(Pet.age > 5)
```

```
# Get pets by owner
greg_pets = Pet.select().where(Pet.owner == "Greg")
```

Pythonic query syntax:

- `==` for equality
- `>`, `<`, `>=`, `<=` for comparisons
- `&` for AND
- `|` for OR

Joins

Before (SQL):

```
cursor.execute("""
    SELECT pet.id, pet.name, pet.age, pet.owner,
           kind.name as kind_name, kind.food, kind.sound
    FROM pet
    JOIN kind ON pet.kind_id = kind.id
""")
```

After (Peewee):

```
def get_pets():
    pets = Pet.select().join(Kind)
    return list(pets)
```

Accessing joined data:

```
pets = Pet.select().join(Kind)
for pet in pets:
    print(pet.name)           # From Pet table
    print(pet.kind.kind_name) # From Kind table (via relationship)
    print(pet.kind.food)      # Automatic join traversal
```

The magic:

- `pet.kind` automatically accesses the related Kind object
- No manual join syntax needed
- Peewee uses the ForeignKeyField definition to generate the JOIN
- Works in templates and code identically

Update

Before (SQL):

```
cursor.execute(
    """update pet set name=?, age=?, owner=? where id=?""",
    (data["name"], data["age"], data["owner"], id)
)
connection.commit()
```

After (Peewee):

```
pet = Pet.get_by_id(id)
pet.name = "New Name"
pet.age = 12
pet.owner = "Alice"
pet.save()
```

Or update in one go:

```
Pet.update(name="New Name", age=12).where(Pet.id == id).execute()
```

Update multiple records:

```
# Make all pets 1 year older
Pet.update(age=Pet.age + 1).execute()

# Change owner for specific pets
Pet.update(owner="Bob").where(Pet.owner == "Alice").execute()
```

Delete

Before (SQL):

```
cursor.execute("delete from pet where id = ?", (id,))
connection.commit()
```

After (Peewee):

```
pet = Pet.get_by_id(id)
pet.delete_instance()
```

Or delete by query:

```
Pet.delete().where(Pet.id == id).execute()
```

Delete multiple records:

```
# Delete all cats
Pet.delete().where(Pet.kind.kind_name == "cat").execute()

# Delete all pets older than 15
Pet.delete().where(Pet.age > 15).execute()
```

Working with Objects vs Dictionaries

In previous topics, database functions returned dictionaries:

```
pets = get_pets()
# Returns: [{'id': 1, 'name': 'Dorothy', 'age': 10, ...}, ...]
print(pets[0]['name'])
```

With Peewee, functions return model instances:

```
pets = get_pets()
# Returns: [<Pet instance>, <Pet instance>, ...]
print(pets[0].name)
```

Benefits of objects:

- Attribute access instead of string keys (`pet.name` vs `pet['name']`)
- Methods available on objects
- Type hints and IDE autocomplete work better
- Can add custom methods to models

For templates, Peewee objects work seamlessly:

```
{% for pet in pets %}
    <td>{{ pet.name }}</td>
    <td>{{ pet.kind.kind_name }}</td>
    <td>{{ pet.age }}</td>
{% endfor %}
```

Notice: `pet.kind.kind_name` traverses the relationship automatically.

Testing with Peewee

```
def test_initialize():
    print("test initialize...")
    initialize("test_pets.db")
    assert db != None
```

Creating test data:

```
def test_get_pets():
    print("test get_pets...")
    kind = Kind(kind_name="dog", food="dog_food", noise="bark")
    kind.save()

    pet = Pet(name="Dorothy", age=10, owner="Greg", kind=kind)
    pet.save()

    pets = get_pets()
    assert type(pets) is list
    assert type(pets[0]) is Pet
    assert pets[0].name == "Dorothy"
```

Key testing patterns:

- Create objects with constructors
- Use `.save()` to persist
- Query returns object instances
- Assert against object attributes

Advantages of Using an ORM

1. No SQL Strings

```
# Before: SQL string with placeholders
cursor.execute("select * from pet where age > ?", (5,))

# After: Python expressions
Pet.select().where(Pet.age > 5)
```

2. Type Safety

```
# SQL: Easy to make typos, no IDE help
cursor.execute("select * from pets where nane = ?", (name,))  # Typo!

# ORM: IDE autocomplete, syntax errors caught immediately
Pet.select().where(Pet.name == name)  # IDE suggests .name
```

3. Automatic Joins

```
# SQL: Manual join syntax
cursor.execute("""
    SELECT pet.*, kind.name
    FROM pet
    JOIN kind ON pet.kind_id = kind.id
""")

# ORM: Relationships handled automatically
pet = Pet.get_by_id(1)
print(pet.kind.kind_name)  # Join implicit
```

4. Schema Generation

```
# SQL: Write CREATE TABLE statements
cursor.execute("""
    create table pet (
        id integer primary key autoincrement,
        name text not null,
        ...
    )
""")

# ORM: Models define schema
class Pet(Model):
    name = CharField()
    ...

db.create_tables([Pet])  # Table created automatically
```

5. DRY Principle (Don't Repeat Yourself)

```
# SQL: Repeat column lists in queries
cursor.execute("insert into pet(name, age, owner) values (?,?,?)", ...)
cursor.execute("update pet set name=?, age=?, owner=? where id=?", ...)

# ORM: Define structure once in model
pet.name = "New"
pet.save()  # Knows what columns to update
```

6. Database Portability

```
# Switch from SQLite to PostgreSQL:
# db = SqliteDatabase('pets.db')
```

```
db = PostgresqlDatabase('pets', user='postgres', password='secret')

# Models and queries stay the same!
```

7. Relationship Navigation

```
# SQL: Manual queries for related data
cursor.execute("select * from kind where id = ?", (pet_kind_id,))

# ORM: Navigate relationships like object attributes
pet.kind.kind_name
kind.pets  # All pets of this kind (backref)
```

Disadvantages and Trade-offs

1. Learning Curve

- Must learn ORM syntax in addition to SQL
- Abstractions hide what's actually happening
- Debugging requires understanding both ORM and SQL

2. Performance Overhead

```
# ORM generates SQL automatically, may not be optimal
pets = Pet.select().join(Kind)  # Might not use ideal indexes

# Raw SQL can be optimized precisely
cursor.execute("select * from pet where id in (1,2,3)")  # Efficient
```

3. Complex Queries Can Be Awkward

```
# Complex SQL with subqueries, CTEs, window functions
# May be clearer in raw SQL than ORM equivalent

# Sometimes need to fall back to raw SQL:
Pet.raw("select * from pet where complex_condition")
```

4. Abstraction Leaks

```
# ORM tries to hide database details, but:
# - Still need to understand foreign keys
# - Still need to think about N+1 queries
# - Still need to handle transactions
# - Database-specific features not always available
```

5. Magic Can Be Confusing

```
# What SQL does this generate?
pets = Pet.select().join(Kind).where(Pet.age > 5)

# Without understanding, hard to debug slow queries
```

6. Less Control

```
# Can't always control exact SQL generated
# May generate inefficient queries
# Harder to use advanced database features
```

When to Use an ORM vs Raw SQL

Use an ORM when:

- Building standard CRUD applications
- Schema changes frequently during development
- Team is more comfortable with Python than SQL
- Need to support multiple databases
- Relationships between entities are straightforward
- Rapid development is priority

Use raw SQL when:

- Performance is critical
- Complex queries with aggregations/subqueries
- Database-specific features needed
- Working with existing complex schema
- Bulk operations on large datasets
- You're comfortable with SQL and want precise control

Best practice: Use ORM for 90% of operations, raw SQL for performance-critical or complex queries.

Peewee's Sweet Spot

Peewee is ideal for:

- ☑ Small to medium applications
- ☑ Prototyping and learning
- ☑ SQLite-based projects
- ☑ Microservices with simple data models
- ☑ Scripts and utilities

Peewee may not be ideal for:

- [!] Large enterprise applications (consider SQLAlchemy)
- [!] Complex legacy schemas with many special cases
- [!] Heavy async requirements (consider Tortoise ORM)
- [!] Tight Django integration (use Django ORM)

Comparing Approaches

Aspect	Topic 04 (Raw SQL)	Topic 05 (ORM)
Query syntax	SQL strings	Python expressions
Return type	Dicts/tuples	Model instances
Schema definition	CREATE TABLE SQL	Model classes
Joins	Manual JOIN syntax	Relationship navigation
Type safety	None	IDE autocomplete
Learning curve	SQL knowledge	ORM + SQL concepts
Performance	Full control	Automatic, may not be optimal
Portability	Database-specific	Database-agnostic

Migration Path from Topic 04

Commented-out code in database.py shows the old SQL approach:

```
# def create_pet(data):
#     cursor = connection.cursor()
#     cursor.execute(
#         """insert into pet(name, age, kind_id, owner) values
(?,?,?,?)""",
#         (data["name"], data["age"], data["kind_id"], data["owner"]),
#     )
#     connection.commit()
```

This helps students compare approaches directly and understand the transformation.

Working with Existing Databases

Peewee can work with existing databases:

```
# Generate models from existing database
python -m pwiz -e sqlite pets.db > models.py
```

This introspects the database and creates model classes automatically.

Running the Application

1. Install Peewee

```
pip install peewee
```

2. Run Tests

```
python database.py
```

This creates `test_pets.db` with schema and test data.

3. Explore Interactively

```
from database import *

initialize("pets.db")

# Create a kind
kind = Kind(kind_name="dog", food="dog_food", noise="bark")
kind.save()

# Create a pet
pet = Pet(name="Buddy", age=5, owner="Alice", kind=kind)
pet.save()

# Query
all_pets = Pet.select()
for pet in all_pets:
    print(f"{pet.name} is a {pet.kind.kind_name}")
```

Key Concepts and Terminology

- **ORM**: Object-Relational Mapping - bridges objects and databases
- **Model**: Python class representing a database table
- **Field**: Class attribute representing a table column
- **Instance**: Object representing a database row
- **ForeignKeyField**: Creates relationships between models
- **Backref**: Reverse relationship for accessing related objects
- **Query**: Expression that generates SQL when executed
- **Select**: Query to retrieve records (SELECT)
- **Save**: Persist object changes to database (INSERT/UPDATE)
- **Delete**: Remove records from database (DELETE)

Best Practices with Peewee

1. ☑ Define models clearly with appropriate field types
2. ☑ Use `get_or_none()` instead of `get_by_id()` to avoid exceptions
3. ☑ Leverage relationships with ForeignKeyField
4. ☑ Use backrefs for reverse relationships
5. ☑ Call `db.create_tables()` in initialization
6. ☑ Test with a separate test database
7. ☑ Use transactions for multiple operations
8. ☑ Add indexes for frequently queried fields
9. ☑ Keep models focused (single responsibility)
10. ☑ Fall back to raw SQL when needed (Peewee supports it)

Key Takeaways

1. **ORMs map tables to classes** and rows to objects
2. **Peewee eliminates SQL strings** with Pythonic syntax
3. **Models define both structure and behavior** in one place
4. **Relationships are navigated like object attributes** (pet.kind.kind_name)
5. **Schema generation is automatic** from model definitions
6. **ORMs trade control for convenience** - understand the trade-offs
7. **Type safety improves** with IDE support and Python expressions
8. **Learning curve exists** but pays off in productivity
9. **Not all problems need an ORM** - raw SQL still has its place
10. **Peewee is perfect for learning** ORM concepts before tackling bigger ORMs

Topic 05 introduces the ORM paradigm that dominates modern web development. While Peewee is simple, the concepts apply to all ORMs (SQLAlchemy, Django ORM, etc.). Understanding both raw SQL (Topics 1-4) and ORM approaches gives you the flexibility to choose the right tool for each situation.