

Topic 06: Dataset Library - Exploratory Database Operations

Overview

This topic introduces the **Dataset** library, a high-level database abstraction designed for **data exploration, scripting, and rapid prototyping**. While previous topics emphasized structured application development, Dataset takes a different approach: making database operations as simple as working with Python dictionaries. Built on top of **SQLAlchemy** (a powerful ORM) and **Alembic** (a migration tool), Dataset hides complexity to prioritize developer convenience and speed.

What is Dataset?

Dataset is a Python library that provides the simplest possible interface to relational databases.

Philosophy:

- **No schema required:** Just insert data, tables and columns are created automatically
- **Dictionary-based:** Work with rows as Python dicts
- **Minimal boilerplate:** Connect and start working immediately
- **Exploratory-friendly:** Perfect for data science, ETL, and quick scripts

Before (Peewee ORM):

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

db.create_tables([Pet])
pet = Pet(name="Suzy", age=3, owner="Greg", kind=kind)
pet.save()
```

After (Dataset):

```
db = dataset.connect('sqlite:///pets.db')
pets_table = db['pets']
pets_table.insert({'name': 'Suzy', 'age': 3, 'owner': 'Greg', 'kind_id':
1})
```

No models, no schema definitions, just dictionaries and method calls.

Dataset's Sweet Spot

Ideal use cases:

- ☒ **Data exploration:** Quick analysis of CSV/JSON data
- ☒ **ETL pipelines:** Extract, transform, load operations
- ☒ **Prototyping:** Rapid experimentation without schema planning
- ☒ **Scripts:** One-off data processing tasks
- ☒ **Jupyter notebooks:** Interactive data analysis
- ☒ **Web scraping:** Store scraped data quickly
- ☒ **Log analysis:** Parse and query log files

Not ideal for:

- ☐ **Production web applications:** Lacks relationship handling and validation
- ☐ **Complex schemas:** Foreign keys and constraints must be managed manually
- ☐ **Performance-critical apps:** Convenience comes with overhead
- ☐ **Type safety:** No model classes means no IDE autocomplete for fields
- ☐ **Large teams:** Lack of explicit schema can cause confusion

Why we use it in this topic:

- Demonstrates a different philosophy (convenience over structure)
- Shows how the same application can be built with different tools
- Illustrates trade-offs between explicitness and simplicity
- Prepares students for data science workflows

Core Concepts

1. Connection

```
db = dataset.connect('sqlite:///pets.db')
```

Connection string format:

- `sqlite:///filename.db` - SQLite file
- `postgresql://user:pass@localhost/dbname` - PostgreSQL
- `mysql://user:pass@localhost/dbname` - MySQL

Key features:

- Single line to connect
- No initialization or configuration needed
- Handles connection pooling automatically
- Can connect to any SQLAlchemy-supported database

2. Table Access

```
pets_table = db['pets']
kinds_table = db['kind']
```

Dictionary-style access:

- `db['table_name']` returns a table object
- Table is created automatically if it doesn't exist
- No need to define schema beforehand
- Same syntax for reading and writing

First use pattern:

```
# First time: table doesn't exist
pets_table = db['pets'] # Creates empty table

# Insert data: columns created on first insert
pets_table.insert({'name': 'Suzy', 'age': 3}) # Creates 'name' and 'age'
columns
```

3. Automatic Schema Management with Alembic

What is Alembic?

Alembic is a database migration tool that tracks and applies schema changes incrementally. Think of it as "version control for your database schema."

Traditional approach (manual migrations):

```
-- Version 1: Initial schema
CREATE TABLE pets (id INTEGER, name TEXT);

-- Version 2: Add age column
ALTER TABLE pets ADD COLUMN age INTEGER;

-- Version 3: Add owner column
ALTER TABLE pets ADD COLUMN owner TEXT;
```

Each change requires manual SQL, tracking which changes have been applied, and coordinating across environments.

Alembic approach:

- Tracks schema versions automatically
- Generates migration files when schema changes
- Applies changes incrementally (can upgrade or downgrade)
- Ensures all environments have consistent schema

How Dataset uses Alembic:

```
# First insert: creates table with these columns
pets_table.insert({'name': 'Suzy', 'age': 3})
# Table: pets (id, name, age)

# Later: insert with new field
pets_table.insert({'name': 'Buddy', 'age': 5, 'owner': 'Alice'})
# Alembic automatically adds 'owner' column!
# Table: pets (id, name, age, owner)
```

Behind the scenes:

1. Dataset detects new column ('owner')
2. Uses Alembic to generate an ALTER TABLE migration
3. Applies the migration automatically
4. Inserts the data with all columns

Benefits:

- **Schema evolves naturally** as data changes
- **No manual ALTER TABLE** commands needed
- **Safe migrations** with rollback support
- **Version tracking** of schema changes

The `.db-shm` and `.db-wal` files:

When using Dataset with SQLite, you'll see:

- `pets.db` - Main database file
- `pets.db-shm` - Shared memory file (for WAL mode)
- `pets.db-wal` - Write-Ahead Log (transaction log)

These support Alembic's migration system and improve concurrency.

4. Insert Operations

Single insert:

```
pets_table.insert({
    'name': 'Suzy',
    'age': 3,
    'owner': 'Greg',
    'kind_id': 1
})
```

Key features:

- Pass a dictionary with column names as keys

- Returns the ID of the inserted row
- Creates columns automatically if they don't exist
- Type inference from Python values

Bulk insert:

```
pets_table.insert_many([
    {'name': 'Suzy', 'age': 3, 'owner': 'Greg', 'kind_id': 1},
    {'name': 'Sandy', 'age': 2, 'owner': 'Steve', 'kind_id': 2},
    {'name': 'Dorothy', 'age': 1, 'owner': 'Elizabeth', 'kind_id': 3}
])
```

Efficient for loading large datasets:

```
import csv
pets_table = db['pets']
with open('pets.csv') as f:
    pets_table.insert_many(csv.DictReader(f))
```

5. Query Operations

Get all records:

```
pets = pets_table.all()
# Returns: iterator of dicts
for pet in pets:
    print(pet['name'], pet['age'])
```

Find with conditions:

```
# Find one
kind = kinds_table.find_one(id=1)

# Find many
dogs = pets_table.find(kind_id=1)

# Multiple conditions
greg_dogs = pets_table.find(owner='Greg', kind_id=1)
```

Return type:

- All queries return **iterators of dictionaries**
- Each row is a **dict** with column names as keys
- Convert to list if needed: `list(pets_table.all())`

6. Update Operations

```
pets_table.update({
    'id': 1,
    'name': 'Suzy Updated',
    'age': 4,
    'owner': 'Greg',
    'kind_id': 1
}, ['id']) # Keys to match on
```

Breaking it down:

- First argument: dictionary with new values (including key)
- Second argument: list of columns to use for matching
- Matches row where `id=1` and updates other fields

Update pattern:

1. Include the identifying field(s) in the dictionary
2. Specify which field(s) to match on in second argument
3. All other fields are updated

Alternative with upsert (update or insert):

```
pets_table.upsert({'id': 1, 'name': 'Suzy', 'age': 4}, ['id'])
# Updates if id=1 exists, inserts if not
```

7. Delete Operations

Delete by condition:

```
pets_table.delete(id=1)
# Deletes all rows where id=1

pets_table.delete(kind_id=2)
# Deletes all cats (kind_id=2)
```

Delete all:

```
pets_table.delete() # Removes all rows (use with caution!)
```

Working Without Relationships

Dataset doesn't have built-in relationship support like ORMs. You must **manually join** data:

```
@app.route("/list")
def get_list():
    pets_table = db['pets']
    kinds_table = db['kind']

    pets = pets_table.all()
    pet_list = []

    # Manual join: look up each pet's kind
    for pet in pets:
        kind = kinds_table.find_one(id=pet['kind_id'])
        pet_list.append({
            'id': pet['id'],
            'name': pet['name'],
            'age': pet['age'],
            'owner': pet['owner'],
            'kind_name': kind['kind_name'],
            'food': kind['food'],
            'noise': kind['noise']
        })

    return render_template("list.html", pets=pet_list)
```

Why manual joins?

- Dataset prioritizes simplicity over features
- Relationship management adds complexity
- For simple cases, manual joins are fine
- For complex apps, use an ORM instead

Performance consideration: This approach has the **N+1 query problem**:

```
pets = pets_table.all() # 1 query
for pet in pets:
    kind = kinds_table.find_one(id=pet['kind_id']) # N queries (one per pet)
```

For 100 pets, this executes 101 queries. ORMs with joins execute just 1 query.

Referential Integrity - Application Level

Dataset doesn't enforce foreign key constraints automatically. You must handle this in application code:

```
@app.route("/kind/delete/<id>")
def delete_kind(id):
    kinds_table = db['kind']
    pets_table = db['pets']
```

```
# Check if any pets reference this kind
if pets_table.find_one(kind_id=id):
    error_message = "Cannot delete kind as it is associated with
pets."
    kinds = kinds_table.all()
    return render_template("kind_list.html", kinds=kinds,
error_message=error_message)

# Safe to delete
kinds_table.delete(id=id)
return redirect(url_for('list_kinds'))
```

Why application-level?

- Dataset doesn't manage schema constraints
- Foreign keys can be defined in SQL (as in `create_db.sql`), but Dataset doesn't enforce them
- Must validate manually to prevent orphaned data

Contrast with Topic 04:

```
# Topic 04: Database enforces constraints
def delete_kind(id):
    cursor.execute("delete from kind where id = ?", (id,))
    # Raises exception if pets reference this kind

# Topic 06: Application must check
def delete_kind(id):
    if pets_table.find_one(kind_id=id):
        return error_message
    kinds_table.delete(id=id)
```

Database Initialization

Using SQL script:

```
sqlite3 pets.db < create_db.sql
```

The SQL file (`create_db.sql`) defines schema with constraints:

```
CREATE TABLE kind (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    kind_name TEXT NOT NULL,
    food TEXT NOT NULL,
    noise TEXT NOT NULL
);

CREATE TABLE pets (
```

```
id INTEGER PRIMARY KEY AUTOINCREMENT,  
name TEXT NOT NULL,  
age INTEGER NOT NULL CHECK (age >= 0),  
kind_id INTEGER NOT NULL,  
owner TEXT NOT NULL,  
FOREIGN KEY (kind_id) REFERENCES kind(id)  
    ON DELETE RESTRICT  
    ON UPDATE CASCADE  
);
```

Why use SQL initialization with Dataset?

- Defines proper primary keys and foreign keys
- Sets NOT NULL constraints
- Creates tables with proper types
- Ensures consistent schema across environments

Without initialization, Dataset creates simple tables:

```
# No SQL setup - Dataset creates basic schema  
pets_table.insert({'name': 'Suzy', 'age': 3, 'kind_id': 1})  
# Creates: pets (id, name, age, kind_id) with basic types, no constraints
```

CRUD Operations Pattern

Create

```
@app.route("/create", methods=['POST'])  
def get_post_create():  
    data = dict(request.form)  
    pets_table = db['pets']  
    pets_table.insert({  
        'name': data['name'],  
        'age': int(data['age']),  
        'owner': data['owner'],  
        'kind_id': int(data['kind_id'])  
    })  
    return redirect(url_for('get_list'))
```

Key points:

- Convert form data to appropriate types
- Insert returns row ID (not captured here)
- No commit needed (automatic)

Read

```

@app.route("/list")
def get_list():
    pets_table = db['pets']
    kinds_table = db['kind']

    pets = pets_table.all()
    pet_list = []

    for pet in pets:
        kind = kinds_table.find_one(id=pet['kind_id'])
        pet_list.append({
            'id': pet['id'],
            'name': pet['name'],
            'kind_name': kind['kind_name'],
            # ... combine data
        })

    return render_template("list.html", pets=pet_list)

```

Update

```

@app.route("/update/<id>", methods=['POST'])
def get_update(id):
    data = dict(request.form)
    pets_table = db['pets']
    pets_table.update({
        'id': id,
        'name': data['name'],
        'age': int(data['age']),
        'owner': data['owner'],
        'kind_id': int(data['kind_id'])
    }, ['id'])
    return redirect(url_for('get_list'))

```

Delete

```

@app.route("/delete/<id>")
def get_delete(id):
    pets_table = db['pets']
    pets_table.delete(id=id)
    return redirect(url_for('get_list'))

```

Comparison Across Topics

Feature	Topic 04 (Raw SQL)	Topic 05 (Peewee ORM)	Topic 06 (Dataset)
---------	--------------------	-----------------------	--------------------

Feature	Topic 04 (Raw SQL)	Topic 05 (Peewee ORM)	Topic 06 (Dataset)
Schema definition	Manual SQL	Model classes	Automatic/SQL
Inserts	SQL with params	Object.save()	dict insert
Queries	SQL strings	Model.select()	table.find()
Returns	Tuples/dicts	Model instances	Dicts
Joins	SQL JOIN	.join() method	Manual lookup
Relationships	Foreign keys	ForeignKeyField	Manual validation
Type safety	None	Model attributes	None
Migrations	Manual ALTER	Manual/Peewee migrations	Automatic (Alembic)
Boilerplate	Medium	High	Minimal
Learning curve	SQL knowledge	ORM concepts	Easiest
Best for	Control & performance	Structured apps	Rapid prototyping

Data Science Workflow Example

Typical Dataset use case:

```
import dataset
import requests
import csv

# Connect
db = dataset.connect('sqlite:///data_analysis.db')
table = db['api_data']

# Fetch data from API
response = requests.get('https://api.example.com/data')
data = response.json()

# Store in database
table.insert_many(data)

# Analyze
results = table.find(category='important')
for row in results:
    print(row['value'])

# Export
with open('results.csv', 'w') as f:
    writer = csv.DictWriter(f, fieldnames=['id', 'value'])
    writer.writeheader()
    writer.writerows(table.all())
```

Why Dataset shines here:

- No schema planning needed
- Quick iteration on data structure
- Easy import/export
- Minimal code

Advantages of Dataset

1. Zero Configuration

```
db = dataset.connect('sqlite:///pets.db')
pets = db['pets']
pets.insert({'name': 'Suzy'})
# Done! Table created, data inserted.
```

2. Schema Evolution

```
# Day 1: Simple schema
pets.insert({'name': 'Suzy', 'age': 3})

# Day 2: Add new field seamlessly
pets.insert({'name': 'Buddy', 'age': 5, 'breed': 'Labrador'})
# 'breed' column added automatically!
```

3. Dictionary Interface

```
pet = pets_table.find_one(id=1)
print(pet['name']) # Dictionary access
# No model classes to define
```

4. Rapid Prototyping

```
# From idea to working code in seconds
db = dataset.connect('sqlite:///prototype.db')
users = db['users']
users.insert({'email': 'test@example.com', 'name': 'Test User'})
```

5. Database Portability

```
# Switch databases by changing connection string
# db = dataset.connect('sqlite:///pets.db')
```

```
db = dataset.connect('postgresql://localhost/pets')  
# Code stays identical
```

Disadvantages of Dataset

1. No Relationship Support

```
# Must manually join  
for pet in pets_table.all():  
    kind = kinds_table.find_one(id=pet['kind_id'])  
    # Tedious for complex relationships
```

2. N+1 Query Problem

```
# Inefficient: queries database for each pet  
for pet in pets_table.all(): # 1 query  
    kind = kinds_table.find_one(id=pet['kind_id']) # N queries
```

3. Limited Validation

```
# Can insert invalid data  
pets_table.insert({'name': 'Suzy', 'age': 'not a number'})  
# No automatic type checking or validation
```

4. No IDE Autocomplete

```
pet = pets_table.find_one(id=1)  
pet['nme'] # Typo! But no error until runtime  
# No autocomplete for field names
```

5. Schema Drift Risk

```
# Typo creates new column  
pets_table.insert({'name': 'Suzy', 'aeg': 3}) # Creates 'aeg' column!  
# Hard to catch in large codebases
```

6. Manual Constraint Enforcement

```
# Must check constraints manually
if not data['name']:
    return "Name required"
if data['age'] < 0:
    return "Age must be positive"
# Database doesn't enforce NOT NULL or CHECK constraints (unless defined
in SQL)
```

When to Use Each Approach

Use Dataset when:

- Exploring unfamiliar datasets
- Building quick prototypes
- ETL/data pipeline scripts
- Schema is fluid and evolving
- Working alone or in notebooks
- Performance is not critical
- Simplicity is paramount

Use ORM (Peewee/SQLAlchemy) when:

- Building structured applications
- Schema is well-defined
- Need relationship handling
- Want type safety and validation
- Working in teams
- Need IDE support
- Performance matters moderately

Use Raw SQL when:

- Maximum performance required
- Complex queries with optimizations
- Working with existing complex schemas
- Need full control over SQL
- Database-specific features needed

Running the Application

1. Install Dataset

```
pip install dataset
```

This also installs SQLAlchemy and Alembic as dependencies.

2. Initialize Database

```
bash create_db.sh
# Or manually:
sqlite3 pets.db < create_db.sql
```

3. Run Application

```
flask --app app run --debug
```

4. Observe Schema Evolution

Try adding a new field:

```
pets_table = db['pets']
pets_table.insert({
    'name': 'Buddy',
    'age': 5,
    'owner': 'Alice',
    'kind_id': 1,
    'breed': 'Labrador' # New field!
})
```

Check the database:

```
sqlite3 pets.db ".schema pets"
# You'll see 'breed' column was added automatically
```

Key Concepts and Terminology

- **Dataset:** High-level database library for rapid development
- **SQLAlchemy:** Underlying ORM that Dataset builds on
- **Alembic:** Migration tool for schema changes
- **Migration:** Versioned change to database schema
- **Schema Evolution:** Automatic addition of columns as data changes
- **WAL Mode:** Write-Ahead Logging for SQLite transactions
- **N+1 Problem:** Performance issue from querying in loops
- **Schema Drift:** Unintended schema changes from typos/evolution
- **Dictionary Interface:** Working with rows as Python dicts

Best Practices with Dataset

1. ☒ Use for prototypes and data exploration
2. ☒ Initialize with SQL for proper constraints

3. ☒ Validate data in application code
4. ☒ Be careful with column names (typos create new columns)
5. ☒ Use `find_one()` instead of loops for lookups
6. ☒ Consider performance implications (N+1 queries)
7. ☒ Document expected schema for team projects
8. ☒ Export to CSV/JSON for sharing results
9. ☒ Transition to ORM for production applications
10. ☒ Combine with pandas for data analysis

Integration with Data Science Tools

With Pandas:

```
import dataset
import pandas as pd

db = dataset.connect('sqlite:///pets.db')
pets = list(db['pets'].all())

# Convert to DataFrame
df = pd.DataFrame(pets)
print(df.describe())
```

With Jupyter:

```
# In notebook
db = dataset.connect('sqlite:///pets.db')
pets_table = db['pets']

# Quick exploration
list(pets_table.all())[:5] # First 5 rows

# Filter
list(pets_table.find(age={'>=': 3}))
```

Key Takeaways

1. **Dataset prioritizes convenience over structure** - perfect for exploration
2. **Alembic enables automatic schema evolution** - columns added as needed
3. **Dictionary interface is simple** but lacks type safety
4. **No relationship support** - must manually join data
5. **Application-level validation required** - database doesn't enforce constraints
6. **N+1 queries are common** - watch for performance issues
7. **Best for prototyping and data science** - not production web apps
8. **Built on SQLAlchemy and Alembic** - can access underlying power if needed
9. **Trade explicitness for speed** - great for iteration, risky for production

10. **Know when to graduate to an ORM** - when structure and validation matter

Dataset represents the "quick and dirty" end of the database library spectrum. It's incredibly useful for its intended purpose (data exploration, ETL, scripting) but intentionally sacrifices features that structured applications need. Understanding where it fits in your toolbox helps you choose the right abstraction level for each task.

This topic completes the progression: from raw SQL (full control) -> ORM (structured objects) -> Dataset (maximum convenience). Each has its place in professional development.