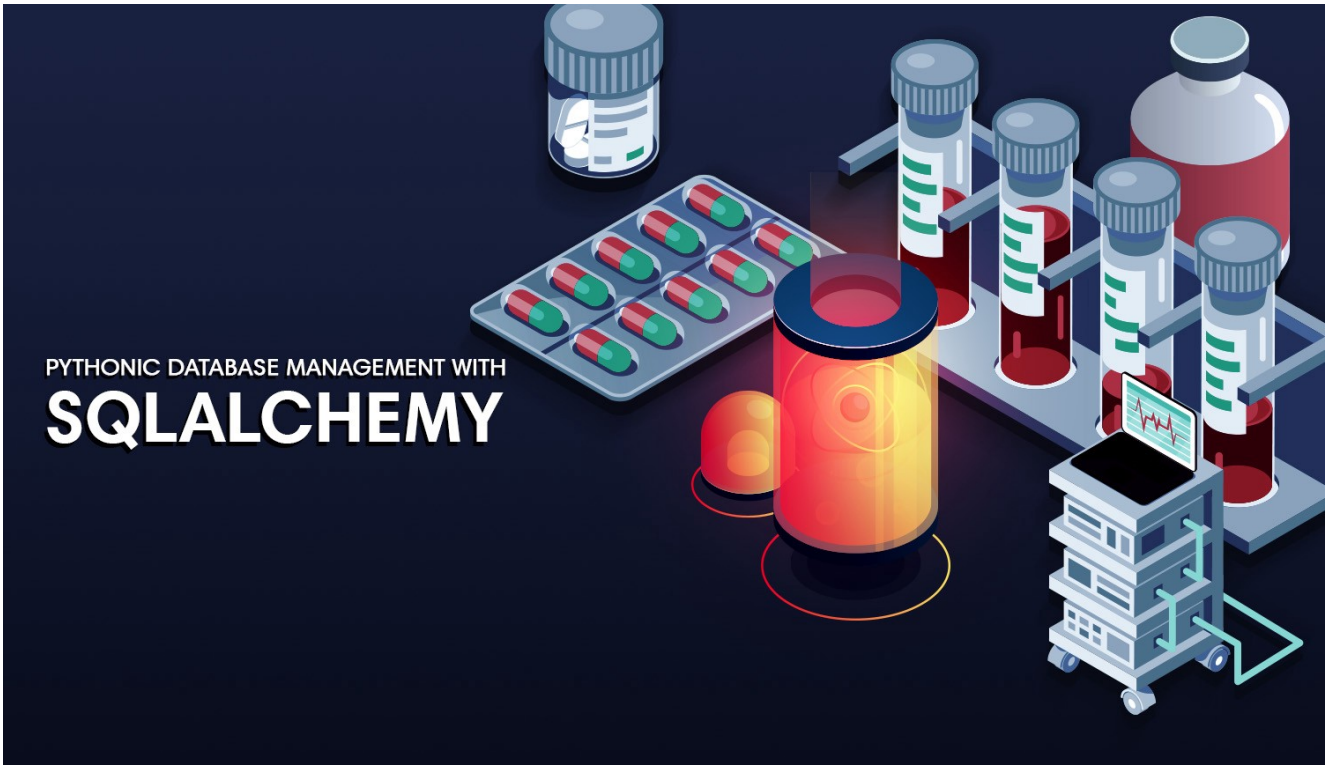
Databases in Python Made Easy with SQLAlchemy

Leverage the iconic SQLAlchemy Python library to effortlessly handle database connections and queries in software.



When we think about software interacting with databases directly, many envision a similar pattern of managing connections, explicitly committing queries, and similarly tiresome boilerplate. Python libraries like PyMySQL and Psycopg2 do a fine job fitting this paradigm and upholding the status quo of what we've come to accept. **SQLAlchemy** has become one of Python's most iconic libraries by abstracting the mundane and trivializing databases in software.

It is also home to the most piss poor documentation for a Python library I've ever had the misfortune of navigating.

SQLAlchemy's API offers an objectively better, simpler, and faster way to work with relational databases. This is useless to those attempting to make sense of the esoteric jargon masquerading as SQLAlchemy's documentation that slaps you across the face and says, *"I don't give a shit about you."*

I'm here to say I *do* give a shit about you and would like nothing more than to express these feelings in the form of an SQLAlchemy tutorial.

SQL or ORM?

SQLAlchemy can serve two purposes: making SQL interactions easier, and serving as a full-blown ORM. Even if you aren't interested in implementing an ORM (or don't know what an ORM is), SQLAlchemy is invaluable as a tool to simplify connections to SQL databases and performing raw queries. I thought that was an original statement as I wrote it, but it turns out SQLAlchemy describes itself this way:

> *SQLAlchemy consists of two distinct components, known as the **Core** and the **ORM**. The Core is itself a fully featured SQL abstraction toolkit, providing a smooth layer of abstraction over a wide variety of DBAPI implementations and behaviors, as well as a SQL*

Expression Language which allows expression of the SQL language via generative Python expressions.

> *SQLAlchemy consists of two distinct components, known as the **Core** and the **ORM**. The Core is itself a fully featured SQL abstraction toolkit, providing a smooth layer of abstraction over a wide variety of DBAPI implementations and behaviors, as well as a SQL Expression Language which allows expression of the SQL language via generative Python expressions.*

Connecting to a Database

SQLAlchemy gives us a few options for managing database connections, but they all begin with the concept of an **engine**. An "engine" is a Python object representing a database. Engines are created one time upfront by passing in connection info for a target database.

Once an engine is created, we can swiftly work with our database at any time by interacting with the engine object. Passing a SQL statement to an engine directly lets SQLAlchemy handle opening a new connection, executing, and immediately closing the connection. In the case of sophisticated web apps sustaining a high volume of data transfer, it's preferable to persistent continuous database connections via a **session**. We'll get to that in the next post.

The syntax for creating an engine is refreshingly simple.

`create_engine()` requires one positional argument, which is a string representing the connection information to connect to a database:

```
"""Create database connection."""
from sqlalchemy import create_engineengine = crea
    "mysql+pymysql://user:password@host:3600/data
)
```

That string being passed into `create_engine()` is a connection URI. Assuming you have a database to work with, we'll figure out what your URI should look like together.

Database Connection URIs

Dissecting a URI into its parts looks like this:

```
[DB_TYPE]+[DB_CONNECTOR]://[USERNAME]:[PASSWORD]@
```

[DB_TYPE]: Specifies the kind (dialect) of database we're connecting to. SQLAlchemy can interface with all mainstream flavors of relational databases. Depending on which database you're connecting to, replace `[DB_TYPE]` with the matching dialect:

- **MySQL**: `mysql`
- **PostgreSQL**: `postgresql`
- **SQLite**: `sqlite`

- **Oracle** (ugh): `oracle`
- **Microsoft** SQL (slightly less exasperated "ugh"): `mssql`

[DB_CONNECTOR]: To manage your database connections, SQLAlchemy leverages whichever Python database connection library you chose to use. If you aren't sure what this means, here are the libraries I'd recommend per dialect:

- **MySQL**: pymysql, mysqldb
- **PostgreSQL**: psycopg2, pg8000
- **SQLite**: (none needed)
- **Oracle**: cx_oracle
- **Microsoft SQL**: pymssql, pyodbc

The variables that come after should all look familiar; these refer to your target database's URL, a database user, that user's password, etc.

With that out of the way, we can install SQLAlchemy along with your connector:

```
$ pip install sqlalchemy pymysql
```

Additional Engine Configuration

Besides a connection URI, engines can be configured to accommodate specific needs (or preferences) of your particular database setup. I personally require

connections to happen over SSL; this requires a PEM key to be provided for extra security, which we're able to pass via the `connect_args` keyword argument when creating our engine:

```
"""Create database connection."""
from sqlalchemy import create_engine
from config import SQLALCHEMY_DATABASE_PEMengine
    "mysql+pymysql://user:password@host:3600/data
    connect_args={"ssl": {"key": SQLALCHEMY_DATAB
    echo=True,
)
```

If you're the curious type, you can also pass `echo=True` to your engine, which will print *all* SQL statements being executed to your console as they happen (including connections, table creation, etc.). This is great for getting insight into what's happening for newcomers but quickly gets annoying and spammy.

Executing Queries

We can run ad hoc queries on an engine object directly by using `engine.execute("SELECT * FROM mytable")`. When we call `execute()` on an engine, SQLAlchemy handles:

- Opening a connection to our database.
- Executing raw SQL on our database and returning the results.

- Immediately closes the database connection used to run this query to avoid hanging connections.

This way of interacting with a database is called ***explicit connectionless execution***. This type of operation is "explicit" in the sense that queries are *automatically committed* upon execution (contrast this to conventional Python libraries, which abide by PEP-249, which do not run queries unless followed by a `commit()`).

This is great when we want to run ad hoc queries on demand (ie: nearly all cases that are not user-facing applications). We can fetch or mutate the data we want without worrying about the extra stuff.

SELECT Data

I've set up a database of dummy data for this example. To demonstrate how easy it is to work with SQLAlchemy, I'm going to start by fetching some data from a table called **nyc_jobs**:

```
...results = engine.execute(
    "SELECT job_id, agency, business_title, \
    salary_range_from, salary_range_to \
    FROM nyc_jobs ORDER BY RAND();"
)
print(results)
```

I've passed a simple SQL query to `engine.execute()`

which SELECTs a few columns from my **nyc_jobs** table (returned in random order). Let's check the output of that:

```
<sqlalchemy.engine.result.ResultProxy object at (
```

Oh God, what the hell is that?! What's a `ResultProxy`? Was this all too good to be true?!

Parsing Query Results

`ResultProxy` is a useful data structure summarizing the result of our query and wrapping the results themselves. The `ResultProxy` object makes it easy to get a high-level look at how our query succeeded by giving us access to attributes, like `rowcount`:

```
...results = engine.execute(
    "SELECT job_id, agency, business_title, \
    salary_range_from, salary_range_to \
    FROM nyc_jobs ORDER BY RAND();"
)print(f"Selected {results.rowcount} rows.")
```

Which gives us...

```
Selected 3123 rows.
```

...But enough horsing around. Let's see what our data looks like by calling `fetchall()` on our `ResultProxy`:

```
...results = engine.execute(
    "SELECT job_id, agency, business_title, \
    salary_range_from, salary_range_to \
    FROM nyc_jobs ORDER BY RAND();"
)print(f"Selected {results.rowcount} rows.")
for row in results.fetchall():
    print(row)
```

Now we're talking:

```
Selected 3123 rows.
(399274, 'NYC HOUSING AUTHORITY', 'Vice Presidenᵗ
(399276, 'BOROUGH PRESIDENT-QUEENS', 'Director oᶠ
(399276, 'BOROUGH PRESIDENT-QUEENS', 'Director oᶠ
(399300, 'NYC HOUSING AUTHORITY', 'VICE PRESIDENᵀ
(399300, 'NYC HOUSING AUTHORITY', 'VICE PRESIDENᵀ
(399349, 'CONSUMER AFFAIRS', 'Deputy Director of
(399349, 'CONSUMER AFFAIRS', 'Deputy Director of
(399355, 'DEPT OF INFO TECH & TELECOMM', 'Assistᵃ
(399357, 'DEPARTMENT OF BUILDINGS', 'Deputy Commᵢ
...
```

Hey, those look like rows containing job listings! These "rows" look to be a list of tuples at first glance, but SQLAlchemy has one more trick up its sleeve:

```
...for row in results.fetchall():
    print(row['business_title'])
```

Whoa, did I just try to fetch a key from a tuple with

`row['business_title']`? Have I lost my mind? As it turns out, our "rows" aren't tuples at all; they're data structures called `RowProxy`:

```
Selected 3123 rows.
COMPUTER SPECIALIST (SOFTWARE)
Resident Engineer
Resident Engineer
Supervising Demolition Inspector
Supervising Demolition Inspector
Senior Inspector
Senior Inspector
Director for Pre-Development Planning
Director for Pre-Development Planning
...
```

business_title is one of our table's column names. By looping over `row['business_title']`, we're seeing every job title in our database; just like looping through a list of dictionaries. In fact, let's do just that: let's return the result of our query as a list of dictionaries:

```
...rows = [dict(row) for row in results.fetchall(
print(rows)
```

Now you have a format that will surely suit your needs regardless of what you're trying to achieve:

```
[
...
```

```
{
  "agency": "DEPT OF INFO TECH & TELECOMM",
  "business_title": "Information Security Analy
  "job_id": 390003,
  "salary_range_from": 75000,
  "salary_range_to": 100000
},
{
  "agency": "HRA/DEPT OF SOCIAL SERVICES",
  "business_title": "JAVA DEVELOPER",
  "job_id": 384199,
  "salary_range_from": 75000,
  "salary_range_to": 180000
},
{
  "agency": "DEPT OF HEALTH/MENTAL HYGIENE",
  "business_title": "Early Childhood Education
  "job_id": 386535,
  "salary_range_from": 64507,
  "salary_range_to": 80118
},
{
  "agency": "DEPT OF ENVIRONMENT PROTECTION",
  "business_title": "Air Pollution Inspector",
  "job_id": 401509,
  "salary_range_from": 43014,
  "salary_range_to": 43014
},
{
  "agency": "DEPT OF DESIGN & CONSTRUCTION",
  "business_title": "Safety Code Compliance Aud
  "job_id": 385796,
  "salary_range_from": 54409,
  "salary_range_to": 73124
```

```
},
{
  "agency": "DEPT OF ENVIRONMENT PROTECTION",
  "business_title": "Procurement Specialist",
  "job_id": 378936,
  "salary_range_from": 48399,
  "salary_range_to": 78211
},
{
  "agency": "DEPT OF DESIGN & CONSTRUCTION",
  "business_title": "College Aide",
  "job_id": 401628,
  "salary_range_from": 16,
  "salary_range_to": 20
},
{
  "agency": "DEPT OF ENVIRONMENT PROTECTION",
  "business_title": "Engineering Technician I",
  "job_id": 348589,
  "salary_range_from": 36239,
  "salary_range_to": 41675
},
{
  "agency": "DEPT OF DESIGN & CONSTRUCTION",
  "business_title": "Executive Assistant",
  "job_id": 395553,
  "salary_range_from": 38851,
  "salary_range_to": 60990
},
{
  "agency": "OFFICE OF THE COMPTROLLER",
  "business_title": "Special Assistant (Corpora
  "job_id": 384671,
  "salary_range_from": 50000,
```

```
    "salary_range_to": 60000
  },
  {
    "agency": "LAW DEPARTMENT",
    "business_title": "Secretary to Division Chie
    "job_id": 315372,
    "salary_range_from": 43319,
    "salary_range_to": 54913
  }
...
]
```


UPDATE Rows

We can run UPDATE queries the same way we ran our SELECT query, with the caveat that there is such thing as a *bad* UPDATE query. Let's make sure we avoid those.

I'm picking a random row from my table to update. I'm going to change row **229837**, which looks like this in JSON form:


```
[
 {
   "job_id": 229837,
   "agency": "DEPT OF INFO TECH & TELECOMM",
   "business_title": "Senior Quality Oversight An
   "job_category": "Information Technology & Tele
   "salary_range_from": 58675,
   "salary_range_to": 125000,
   "salary_frequency": "Annual",
   "work_location": "75 Park Place New York Ny",
```

```
   "division": "General Counsel",
   "created_at": "2016-02-17T00:00:00.000",
   "updated_at": "2016-02-17T00:00:00.000"
 }
]
```

Let's write a potentially dangerous SQL query that introduces some problematic characters:

```
UPDATE
 nyc_jobs
SET
 business_title = 'Senior QA Scapegoat 🏆',
 job_category = 'Info <>!#%%Technology%%#^&%* & 
WHERE
 job_id = 229837;
```

It's a bad idea to attempt an UPDATE where the value contains operators that are sure to break the operation. Let's see what happens:

```
result = db.execute(
    "UPDATE nyc_jobs SET business_title = 'Senio
    job_category = 'Information <>!#%%Technology
    WHERE job_id = 229837;"
)
print(result.rowcount)
```

Running this pisses off *everything*:

```
2021-01-09 15:44:14,122 INFO sqlalchemy.engine.ba
2021-01-09 15:44:14,123 INFO sqlalchemy.engine.ba
2021-01-09 15:44:14,123 INFO sqlalchemy.engine.ba
Traceback (most recent call last):
  File "/Users/toddbirchard/projects/sqlalchemy-t
    init_script()
  File "/Users/toddbirchard/projects/sqlalchemy-t
    rows_updated = update_job_listing(db)
  File "/Users/toddbirchard/projects/sqlalchemy-t
    result = db.execute(
  File "/Users/toddbirchard/projects/sqlalchemy-t
    return connection.execute(statement, *multipa
  File "/Users/toddbirchard/projects/sqlalchemy-t
    return self._execute_text(object_, multiparam
  File "/Users/toddbirchard/projects/sqlalchemy-t
    ret = self._execute_context(
  File "/Users/toddbirchard/projects/sqlalchemy-t
    self._handle_dbapi_exception(
  File "/Users/toddbirchard/projects/sqlalchemy-t
    util.raise_(exc_info[1], with_traceback=exc_i
  File "/Users/toddbirchard/projects/sqlalchemy-t
    raise exception
  File "/Users/toddbirchard/projects/sqlalchemy-t
    self.dialect.do_execute(
  File "/Users/toddbirchard/projects/sqlalchemy-t
    cursor.execute(statement, parameters)
  File "/Users/toddbirchard/projects/sqlalchemy-t
    query = self.mogrify(query, args)
  File "/Users/toddbirchard/projects/sqlalchemy-t
    query = query % self._escape_args(args, conn)
TypeError: * wants int
```

Luckily for us, SQLAlchemy has a method called `text()` which escapes dangerous characters found in queries like these. ***Always*** wrap your queries in this method:

```
...
from sqlalchemy import textresult = db.execute(
    text(
        "UPDATE nyc_jobs SET business_title = 'Se
        job_category = 'Information <>!#%%Techno
        WHERE job_id = 229837;"
    )
)
print(result.rowcount)
```

We now receive no errors, and a count of `1` is returned to indicate that a single row was updated.

And So Much More

Unless you're a masochist, the SQLAlchemy workflow of running queries is objectively better than the boilerplate you'd be dealing with while using a vanilla DB connector. I can't begin to quantify the amount of time I've personally saved from this simplified workflow, which makes the following assertion seem outlandish: there is still so much more to gain from SQLAlchemy.

Anyway, one step at a time. I put together a working demo containing all the source code of what we covered in this tutorial on Github below. Join us next time!

hackersandslackers/sqlalchemy-tutorial

This repository contains the source code for a four-part tutorial series on SQLAlchemy: Databases in Python Made Easy...

Originally published at hackersandslackers.com