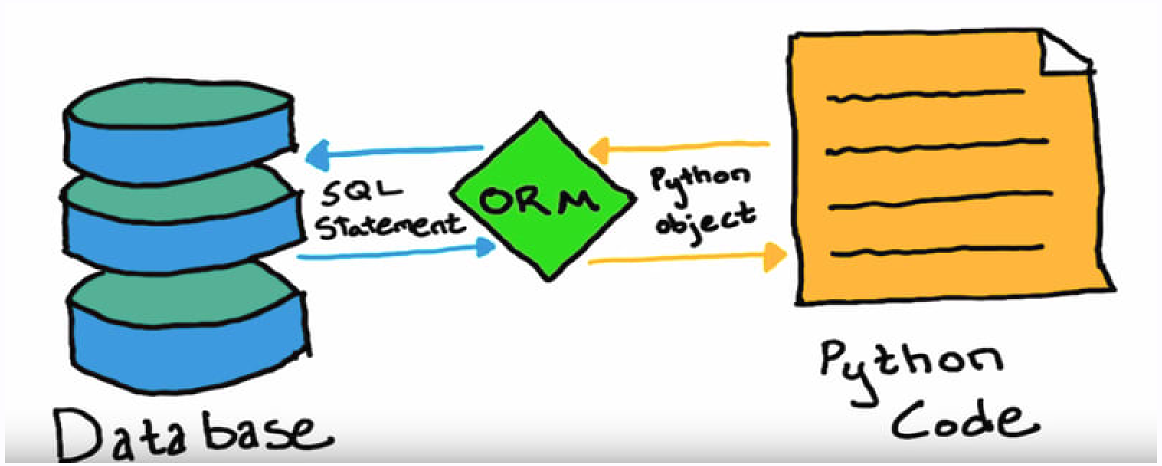
Intro to SQLAlchemy

ORMs in python



whois: `Bauke Brenninkmeijer`

- MSc in CS and Data Science @Nijmegen
- Data Scientist @ABNAMRO since 2019
  - 1.5 years in Data Management
  - 2 years in Global Markets
- @baukebrenninkmeijer

Outline

1. Introduction
2. SQLAlchemy
3. PyODBC
4. Comparison of features and advantages
5. Examples
6. conclusion

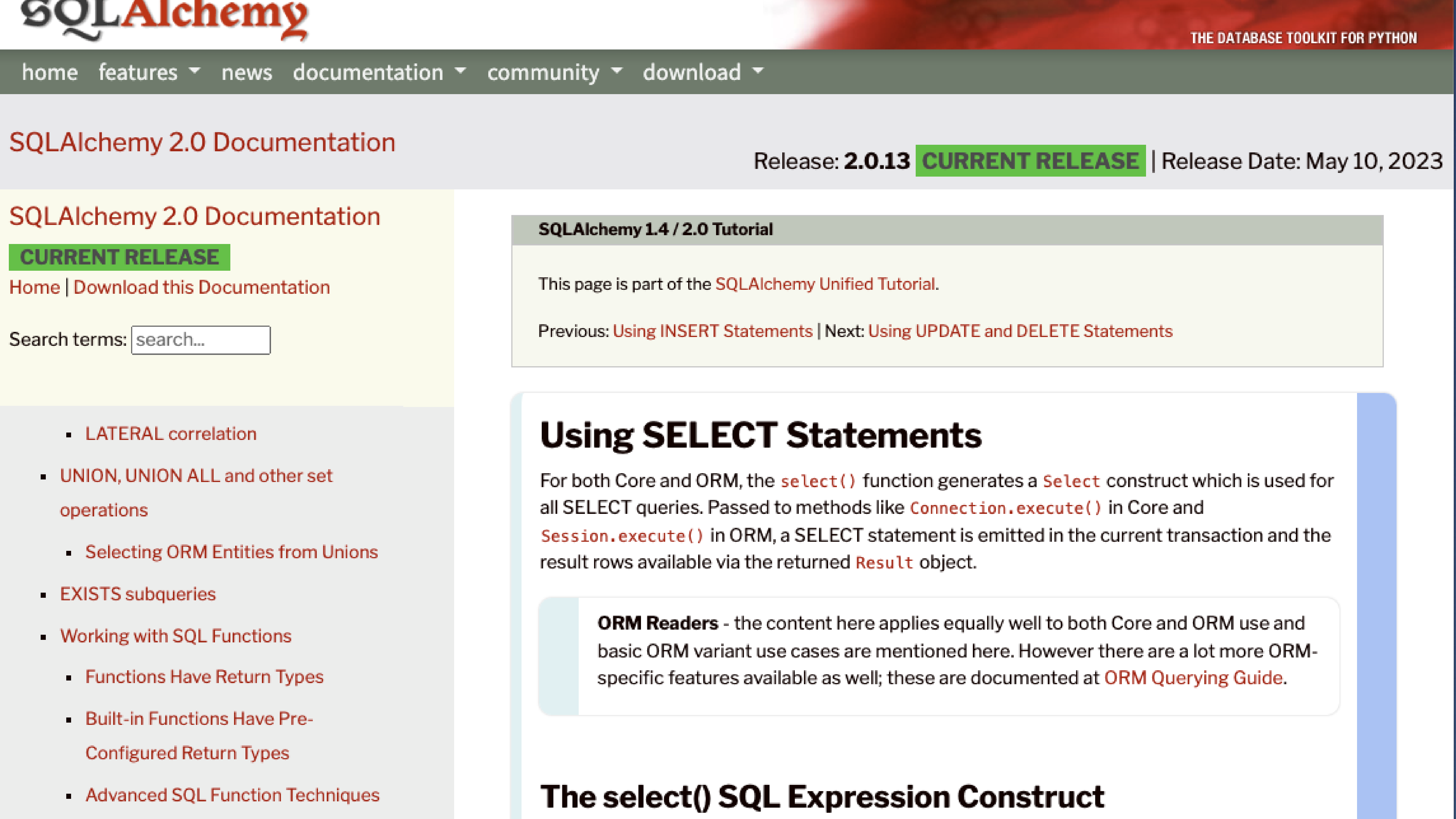
Introduction

SQLAlchemy: Python SQL toolkit and ORM that gives full SQL capabilities to python

PyODBC: Python module that allows connecting to any ODBC database

We will compare and contrast the features, advantages and disadvantages.

ODBC: Open Database Connectivity



SQLAlchemy 2.0 Documentation

Release: **2.0.13** **CURRENT RELEASE** | Release Date: May 10, 2023

SQLAlchemy 2.0 Documentation

CURRENT RELEASE

Home | Download this Documentation

Search terms: search...

SQLAlchemy 1.4 / 2.0 Tutorial

This page is part of the SQLAlchemy Unified Tutorial.

Previous: Using INSERT Statements | Next: Using UPDATE and DELETE Statements

Using SELECT Statements

For both Core and ORM, the `select()` function generates a `Select` construct which is used for all SELECT queries. Passed to methods like `Connection.execute()` in Core and `Session.execute()` in ORM, a SELECT statement is emitted in the current transaction and the result rows available via the returned `Result` object.

> **ORM Readers** - the content here applies equally well to both Core and ORM use and basic ORM variant use cases are mentioned here. However there are a lot more ORM-specific features available as well; these are documented at ORM Querying Guide.

The select() SQL Expression Construct

Database support

- **SQLAlchemy** supports multiple databases and backends, including
  - MySQL
  - PostgreSQL
  - SQLite
  - Can use PyODBC to connect to almost anything
- **PyODBC** supports any database with DB API 2.0, which also includes:
  - SQL Sever
  - Access
  - Excel

What is an ORM?

- **O**bject **R**elational **M**apping
- Method to align code and database structures
- Allow to interact with databases in code, rather than raw SQL.

ORM for us

- **SQLAlchemy** has ORM that allows definition of classes that represent database tables, manipulating data using python syntax.
- **PyODBC** does not have any ORM features. You can use other ORMs on top of PyODBC, such as Django, PeeWee, and SQLAlchemy.

Query Construction

- **SQLAlchemy**
  - Supports constructing SQL queries using Python expressions and operators.
  - Textual SQL is also supported.
- **PyODBC**
  - requires raw SQL statements as string, which are passed to the cursor object.
  - Placeholders and parameters are needed to dynamic usage and avoiding SQL injection attacks.

3 APIs

SQLAlchemy actually has 3 different APIs

- **SQLAlchemy 2.0 style** (used today)
- SQLAlchemy ORM 1.x style
- SQLAlchemy Core 1.x style

What it is not

SQLAlchemy is not an analytics interface.

Large aggregations and complex groupbys are **not supported**.

It's main functions are **retrieval, insert, update and deleting** of data.

Creating tables

```python
# SQLAlchemy
# imports...
engine = create_engine('sqlite:///test.db', echo=True)
Base = declarative_base()

# Define a class that represents the users table
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

metadata.create_all(engine)
```

Creating tables

```
# PyODBC
import pyodbc

conn = pyodbc.connect('DRIVER={SQLite3};DATABASE=test.db')
cursor = conn.cursor()
cursor.execute('''
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER
    )
''')
conn.commit()
conn.close()
```

Inserting data

Python typehints ✨

```python
# SQLAlchemy
Session = sessionmaker(bind=engine)
session = Session()

# Insert some data into the table using the User class
session.add_all([
    User(name='Alice', age=25),
    User(name='Bob', age=30),
    User(name='Charlie', age=35)
])
session.commit()
```

Inserting data

Just raw text. Hard to programmatically extend reliable.

No SQL injection prevention.

```
# PyODBC
cursor.execute('''
    INSERT INTO users (name, age) VALUES
    ('Alice', 25),
    ('Bob', 30),
    ('Charlie', 35)
''')
```

Retrieving data

```
from sqlalchemy import select
session.scalars(
    select(User)
).all()
```

```
[User(id=1, name=Alice, age=25),
 User(id=2, name=Bob, age=30),
 User(id=3, name=Charlie, age=35)]
```

Retrieving data

```python
# PyODBC
cursor.execute('SELECT * FROM users')

# Fetch and print the rows from the query result
rows = cursor.fetchall()
for row in rows:
    print(row)
```

```
(1, 'Alice', 25)
(2, 'Bob', 30)
(3, 'Charlie', 35)
```

More advanced

with table `User` and `Address`

```
result = session.execute(
    select(User.name, Address.email_address)
    .join(User.addresses)
    .order_by(User.id, Address.id)
)
```

WHERE

```
session.scalars(select(User).where(User.age > 28)).all()
```

```
[User(id=2, name=Bob, age=30),
 User(id=3, name=Charlie, age=35)]
```

Granular inserts

```
new_user = User(name='dennis', age=58)

session.add(
    new_user
)
session.commit()
```

Granular updates

```python
from sqlalchemy import update
stmt = (
    update(User)
    .where(User.name == "Alice")
    .values(name="Alice the Third von Baumgarten")
)
session.execute(stmt)
session.commit()
```

Delete

```python
from sqlalchemy import delete
stmt = delete(User).where(User.name.in_(["Bob"]))
session.execute(stmt)
```

Questions