SQLALCHEMY - CHEAT SHEET

Introduzione e agenda

Che cos'è SQLAlchemy

SQLAlchemy è un framework che fornisce:

- Astrazione per l'interazione col DB e la creazione di statement SQL ("core")
- Funzionalità ("ORM")

Agenda

- Introduzione, installazione
- Architettura: core/ORM
- Engine, Session, Connection
- Definizione dei modelli
- Operazioni CRUD di base
- Esempi di vari tipi di select

Installazione

Installazione di SQLAlchemy

```
pip install sqlalchemy
```

Nella semplicità del comando di installazione è "nascosta" la natura da un lato doppia (core/orm) dall'altra parte inscindibile che suggerirà alcune strategie d'uso.

Installazione dei driver del DB

```
# MySQL
pip install pymysql

# Oracle
pip install cx-oracle

# Postgres
pip install psycopg2
```

Tutorial e documentazione

SQLAlchemy 2 ha - finalmente - un tutorial.

Home page del tutorial https://docs.sqlalchemy.org/en/20/tutorial/index.html

Altre pagine di interesse:

Engine e connessioni https://docs.sqlalchemy.org/en/20/core/connections.html#result-set-api

Pattern di relazioni https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html

Introduzione a SQLAlchemy Core https://docs.sqlalchemy.org/en/20/core/

Approccio "code first" vs "data first"

Senza voler nulla togliere a nessuno dei due approcci, qui viene presentato un approccio di tipo "data first" perchè nell'esperienza di chi scrive è la situazione più comune nella quale ci si viene a trovare nella realtà lavorativa di tutti i giorni'

L'approccio data first durerebbe fino alla prima release...

Modi di funzionamento: Core e ORM

Core

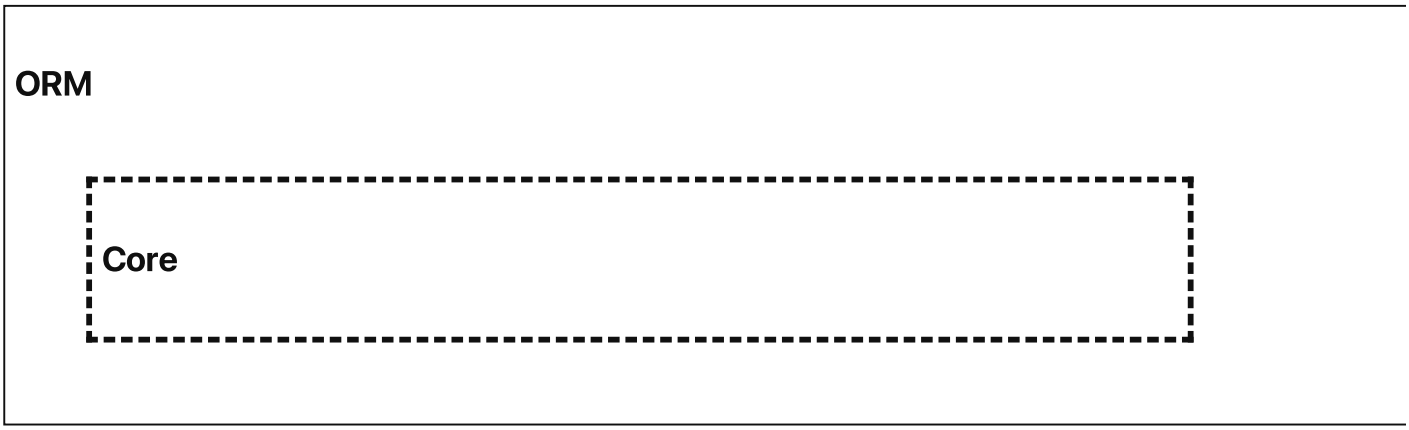
Dal tutorial:

> "SQLAlchemy Core is the foundational architecture for SQLAlchemy as a “database toolkit”. The library provides tools for managing connectivity to a database, interacting with database queries and results, and programmatic construction of SQL statements."

ORM

Dal tutorial:

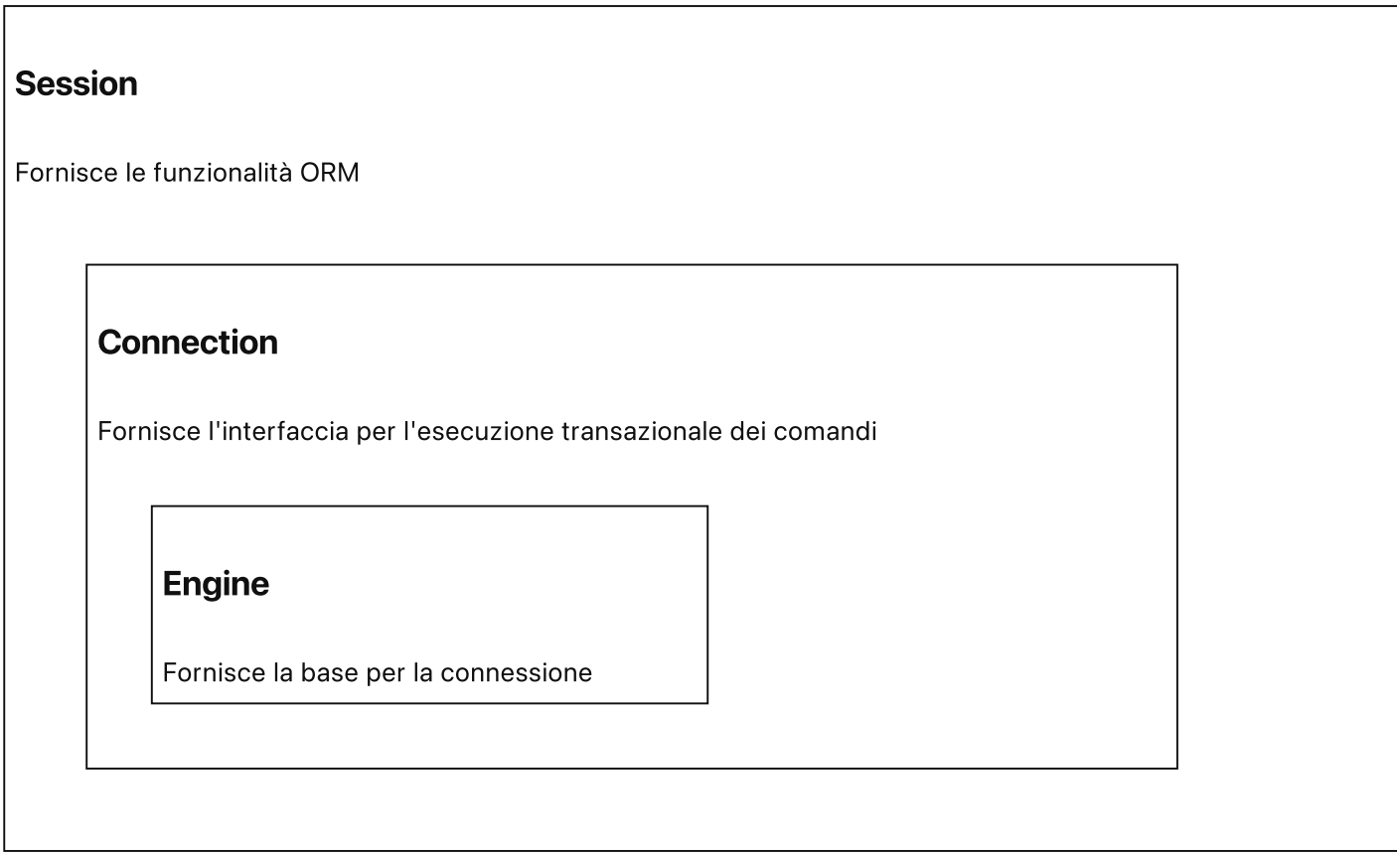
> "SQLAlchemy ORM builds upon the Core to provide optional object relational mapping capabilities. The ORM provides an additional configuration layer allowing user-defined Python classes to be mapped to database tables and other constructs, as well as an object persistence mechanism known as the Session. It then extends the Core-level SQL Expression Language to allow SQL queries to be composed and invoked in terms of user-defined objects."



Engine, Connection e Session

Alla base della connettività con il database ci sono tre oggetti fondamentali:

- Engine: Fornisce la base per la connessione al db
- Connection: Fornisce l'interfaccia per l'esecuzione transazionale dei comandi
- Session: Fornisce le funzionalità ORM (ed esegue gli statement SQL via Connection)



Differenza tra Connection e Session https://stackoverflow.com/questions/34322471/sqlalchemy-engine-connection-and-session-difference

Percorso consigliato da SQLAlchemy

Leggendo la documentazione si evince che SQLAlchemy permette ed ha permesso nel tempo una molteplicità di modalità operative; Con la versione 2.0 tuttavia emerge che la via consigliata di lavorare è un mix di feature core ed ORM.

In particolare:

- Per l'accesso alle operazioni di database viene consogliato di utilizzare sempre la Session (ORM)
- Per la definizione delle tabelle viene consigliata la sintassi ORM in quanto più completa di funzionalità
- Per l'esecuzione delle operazioni sul DB sono disponibili entrambe le strade; tuttavia se non ci sono particolari necessità di utilizzare le finzionalità ORM viene suggerito di eseguire glli statement in modalità e stile core.

Creazione della connessione e sessione

Per creare una connessione o una sessione si segue grossomodo lo schema logico spiegato sopra.

I passaggi logici risultano quindi essere:

1. Creazione di un oggetto Engine

```
from sqlalchemy import create_engine

database:str = "mysql+pymysql://root:password@localhost:3306/sandbox?
charset=utf8mb4"
engine = create_engine(database)
```

2/a. Creazione di un oggetto Connection

Per completezza si mostra l'istanziazione di un a Connection anche se la raccomandazione di SQLAlchemy è quella di utilizzare un oggetto di tipo Session anche se i utilizzano le sole funzionalità ORM

```
with engine.connect() as connection:
    ...
```

2/b. Creazione di un oggetto Session

L'utilizzo dell'oggetto Session è la via raccomandata anche nel caso in cui si vogliano scrivere gli statement in modalità "core"

Gli step logici sono:

1. Stringa di connessione -> Engine
2. Session maker
3. Uso della scoped_session (context manager)

Demo: `demo_ecommerce/connection/connection.py`

```
from sqlalchemy.orm import Session
from sqlalchemy.orm.session import sessionmaker

# Creare un session_maker
# https://medium.com/@keyyleiva/easy-database-handling-with-sqlalchemys-
sessionmaker-
659f718a86d5#:~:text=The%20SessionMaker%20is%20a%20tool,changes%2C%20and%20executi
ng%20database%20queries.

Session = sessionmaker(bind=engine)

# Creare un oggetto session

session = Session()
```

3. Utilizzo dell'oggetto connection o session come context manager

Il modo suggerito è quello di usare la session come context manager

```
with Session() as session:
    ...
```

Definizione tabelle

Alla base delle funzionalità di SQLAlchemy ci sono i concetti di tabella e colonna, utilizzati sia dalla parte Core che dalla parte ORM.

L'oggetto MetaData

E' la collezione di oggetti che rappresentano il database (tabelle, colonne...)

E' un concetto condiviso tra la parte core ed ORM

```
from sqlalchemy import MetaData

metadata = MetaData()
```

Come per quasi tutti gli aspetti di SQLAlchemy, le tabelle possono essere definite con due "stili": core ed ORM

Definizione delle tabelle in stile Core

Viene illustrato per completezza un esempio di creazione tabelle in stile Core, anche se è consigliato comunque di utilizzare l'ORM

https://docs.sqlalchemy.org/en/20/tutorial/metadata.html#setting-up-metadata-with-table-objects

Si veda anche ```tutorial/core/models.py``````

Esempio di tabella

```
from sqlalchemy import Column, Integer, String, MetaData

from sqlalchemy import Table

db_metadata = MetaData()

customer = Table(
    "customer",
    db_metadata,
    Column("id", Integer, primary_key=True),
    Column("name", String(50), nullable=False),
    Column("address", String(100), nullable=True),
)
```

Vincoli e relazioni stile Core

https://docs.sqlalchemy.org/en/20/core/constraints.html

Chiave esterna

```
Column("item_id", ForeignKey(item.c.id), nullable=False),
```

Unicità

```
Column("colx", Integer, unique=True),
...
UniqueConstraint("col_a", "col_b", name="my_unique_constraint"),
```

Vincoli vari

Si noti come sia possibile definire il vincolo al di fuori della tabella, esattamente come in SQL, con il grosso vantaggio di poter riferirsi direttamente agli oggetti colonna e non esprimere espressioni testuali (non parsate che rischiano di rompersi a runtime)

Tuttavia anrebbero usati col contagocce.

```
CheckConstraint("col2 > col3 + 5", name="check1"),

...

CheckConstraint(
    price_list.c.price > 0,
    name="check_price",
    table=price_list,
)
```

Chiave primaria

La versione esplicita (`PrimaryKeyConstraint("id")`) in fase di creazione effettua il controllo della correttezza del nome colonna per cui risulta *safe* (basta un test unitario o un conftest che fa il create_all per tenere sotto controllo la cosa); ha il vantaggio di poter dare il nome al constraint.

```
Column("id", Integer, primary_key=True),

...

item = Table(
    "item",
    db_metadata,
    Column("id", Integer),
    Column("description", String(50), nullable=False),
    PrimaryKeyConstraint("id", name"my_primary_key"),
)
```

Creazione delle tabelle in modalità core

```
engine = create_engine(database)

db_metadata.create_all()
```

Definizione delle tabelle in stile ORM

Demo: `demo_ecommerce/models/models.py`

https://docs.sqlalchemy.org/en/20/tutorial/metadata.html#using-orm-declarative-forms-to-define-table-metadata

Le definizioni in stile ORM sono quelle raccomandate perché

- Sono in stile più "pythonico"
- Migliore compatibilità con i type checkers (mypy)
- Disponibilità sia delle funzionalità ORM che Core, essendo una Façade su Table

```python
class BaseModel(DeclarativeBase):
    pass


db_orm_metadata = BaseModel.metadata

# Sqlalchemy 2.x

class Customer(BaseModel):
    __tablename__ = "customer"

    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    name: Mapped[str] = mapped_column(String(50), nullable=False)
    address: Mapped[str] = mapped_column(String(200), nullable=True)
```

```python

# Sqlalchemy 1.4

class Customer(BaseModel):
    __tablename__ = "customer"

    id: int = sa.Column(sa.Integer, primary_key=True)
    name: str = sa.Column(sa.String(50), nullable=False)
    address: str = sa.Column(sa.String(200), nullable=True)
```

Vincoli e relazioni in stile ORM

https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html

Le relazioni vengono definite tutte attraverso l'insieme di Foreign Key e la funzione relationship().

```
class Customer(BaseModel):
    __tablename__ = "customer"

    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    name: Mapped[str] = mapped_column(String(50), nullable=False)
    address: Mapped[str] = mapped_column(String(200), nullable=True)
    invoices: Mapped[List[Invoice]] = relationship(back_populates="customer")

class Invoice(BaseModel):
    __tablename__ = "invoice"
    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    customer_id: Mapped[int] = mapped_column(ForeignKey(Customer.id))
    customer: Mapped[Customer] = relationship(back_populates="invoices")
```

Creazione delle tabelle in modalità orm

```
engine = create_engine(database)

BaseModel.metadata.create_all(engine)
```

Ambiente sandbox

demo_ecommerce/database/start-db.sh

```
# Once:
#     docker pull mysql
#
# Enter
#     mysql sandbox --password=password
#     mysql sandbox --user utente --password=password


docker run \
    --name mysql-sandbox \
    -e MYSQL_ROOT_PASSWORD=password \
    -e MYSQL_DATABASE=sandbox \
    -v ./init.d:/docker-entrypoint-initdb.d \
    -p 3306:3306 \
    -it -d mysql:latest
```

```
CREATE USER 'utente' identified by 'password';

GRANT ALL PRIVILEGES ON *.* TO 'utente';

FLUSH PRIVILEGES;
```

Costruzione ed esecuzione statement SQL

Esempi pratici di esecuzione statement SQL di manipolazione dati e query

Insert

Insert Core Style

```
INSERT INTO customer (name, address) VALUES ('Ettore', 'Via dei Tigli')
```

```
from sqlalchemy import insert
session.execute(insert(Customer).values(name="Ettore", address="Via dei Tigli"))

session.commit()
```

Insert ORM Style

dml_01_insert_core.py

```
record = Customer(name="Ettore", address="Via dei Tigli")
session.add(record)

session.commit()
```

Select (base)

https://docs.sqlalchemy.org/en/20/tutorial/data_select.html#selecting-orm-entities-and-columns

Differenza tra core ed ORM nel risultato:

> When executing a statement like the above using the ORM Session.execute() method, there is an important difference when we select from a full entity such as User, as opposed to user_table, which is that the entity itself is returned as a single element within each row.

Select base Core Style

```
from sqlalchemy import select

query = select(Item).where(Item.code == "L001")
result = session.execute(query).all()

# result_orm == [(Item,)]
```

Select base ORM Style

```
result = session.query(Item).filter(Item.code == "L002").all()

# result == [Item]
```

Update

dml_02_update_core.py

Update Core Style

```
UPDATE customer SET address='Via dei Tigli, 2/D' WHERE customer.id = 1
```

```
from sqlalchemy import update

session.execute(
    update(Item).values(description="Lampadina standard").where(Item.code ==
"L002")
)
```

Update ORM Style

```
l2 = session.query(Item).where(Item.code=="L002").one_or_none()
l2.description = "Lampadina standard"
```

Delete

dml_03_delete_core.py

Delete Core Style

```
from sqlalchemy import delete

session.execute(
    delete(Item).where(Item.code == "P001")
)
```

Delete ORM Style

```
p1 = session.query(Item).where(Item.code=="P001").one_or_none()
session.delete(p1)
```

Select *"avanzate"* (ricette)

Di seguito una serie di esempi pratici di costruzioni di query che implementano varie tipologie di istruzioni SQL

[00] [01] Query di base, label, alias

demo_ecommerce/query/query_00_base_alias_core.py
demo_ecommerce/query/query_00_base_alias_orm.py

demo_ecommerce/query/query_01_simple_core.py demo_ecommerce/query/query_01_simple_orm.py

Label e alias:

```
detail = aliased(InvoiceDetail)

detail.item_code.label("articolo")
```

[02] Group by

demo_ecommerce/query/query_02_group_core.py demo_ecommerce/query/query_02_group_orm.py

- Campo di raggruppamento
- Funzioni di aggregazione
- Statement di raggruppamento

```
    query = select(
        InvoiceDetail.item_code, func.count(InvoiceDetail.id).label("occurrences")
    ).group_by(InvoiceDetail.item_code)
```

[03] Limit e Offset

demo_ecommerce/query/query_03_limit_offset.py

Limit e offset sono utili per la paginazione

```
    query = (
        select(Customer, Customer.id.label("duplicated_id"))
        .offset(5)
        .limit(3)
    )
```

[04] Estrazione risultato

demo_ecommerce/query/query_04_result_as_dict.py

Limit e offset sono utili per la paginazione

```python
def result_as_dict(record: Union[Row, DeclarativeBase]) -> Dict:
    if isinstance(record, DeclarativeBase):
        return model_as_dict(record=record)

    fields = record._mapping.keys()  # pylint: disable=protected-access

    elements = [
        (
            model_as_dict(getattr(record, field, None))  # type: ignore[arg-type]
            if isinstance(getattr(record, field, None), DeclarativeBase)
            else {field: getattr(record, field, None)}
        )
        for field in fields
    ]
    return reduce(lambda acc, cur: {**acc, **cur}, elements, {})
```

[05] Join

demo_ecommerce/query/query_05_join_core.py demo_ecommerce/query/query_05_join_orm.py

https://docs.sqlalchemy.org/en/20/tutorial/data_select.html#explicit-from-clauses-and-joins

```
"""
SELECT customer.id, customer.name, customer.address, invoice.id AS id_1,
invoice.customer_id
FROM customer INNER JOIN invoice ON customer.id = invoice.customer_id
"""


query = select(Customer, Invoice).join(
    Invoice, Customer.id == Invoice.customer_id
)

result = session.execute(query).all()
```

[06] Exists

demo_ecommerce/query/query_06_exists_core.py

https://docs.sqlalchemy.org/en/20/tutorial/data_select.html#explicit-from-clauses-and-joins

```
"""
SELECT customer.id, customer.name, customer.address
FROM customer
WHERE EXISTS (SELECT 1 AS anon_1
FROM invoice
WHERE customer.id = invoice.customer_id)
"""

subq_exists = (
    select(literal(1)).where((Customer.id == Invoice.customer_id)).exists()
)

query_exists_2 = select(Customer).where(subq_exists)
```

[07] Aliased

demo_ecommerce/query/query_07_aliased.py

https://docs.sqlalchemy.org/en/14/orm/query.html#sqlalchemy.orm.aliased

```
"""
(SELECT `FATTURE`.id, `FATTURE`.customer_id
FROM invoice AS `FATTURE`
 LIMIT 3) UNION ALL SELECT `TOTALI`.id, `TOTALI`.customer_id
FROM (SELECT sum(invoice.id) AS id, sum(invoice.customer_id) AS customer_id
FROM invoice) AS `TOTALI`
"""

fatture = aliased(Invoice, name="FATTURE")

totali = select(
    func.sum(Invoice.id).label("id"),
    func.sum(Invoice.customer_id).label("customer_id"),
).subquery()
totali_fatture = aliased(totali, name="TOTALI")

query = select(fatture).limit(3).union_all(select(totali_fatture))
```