AA 2022-23

Università degli Studi di Cagliari

Short tutorial on SQLAlchemy

Data base and data analytics

Corso di Laurea IADA

Informatica Applicata e Data Analytics

Prof. Giuliano Armano

Summary of the lesson

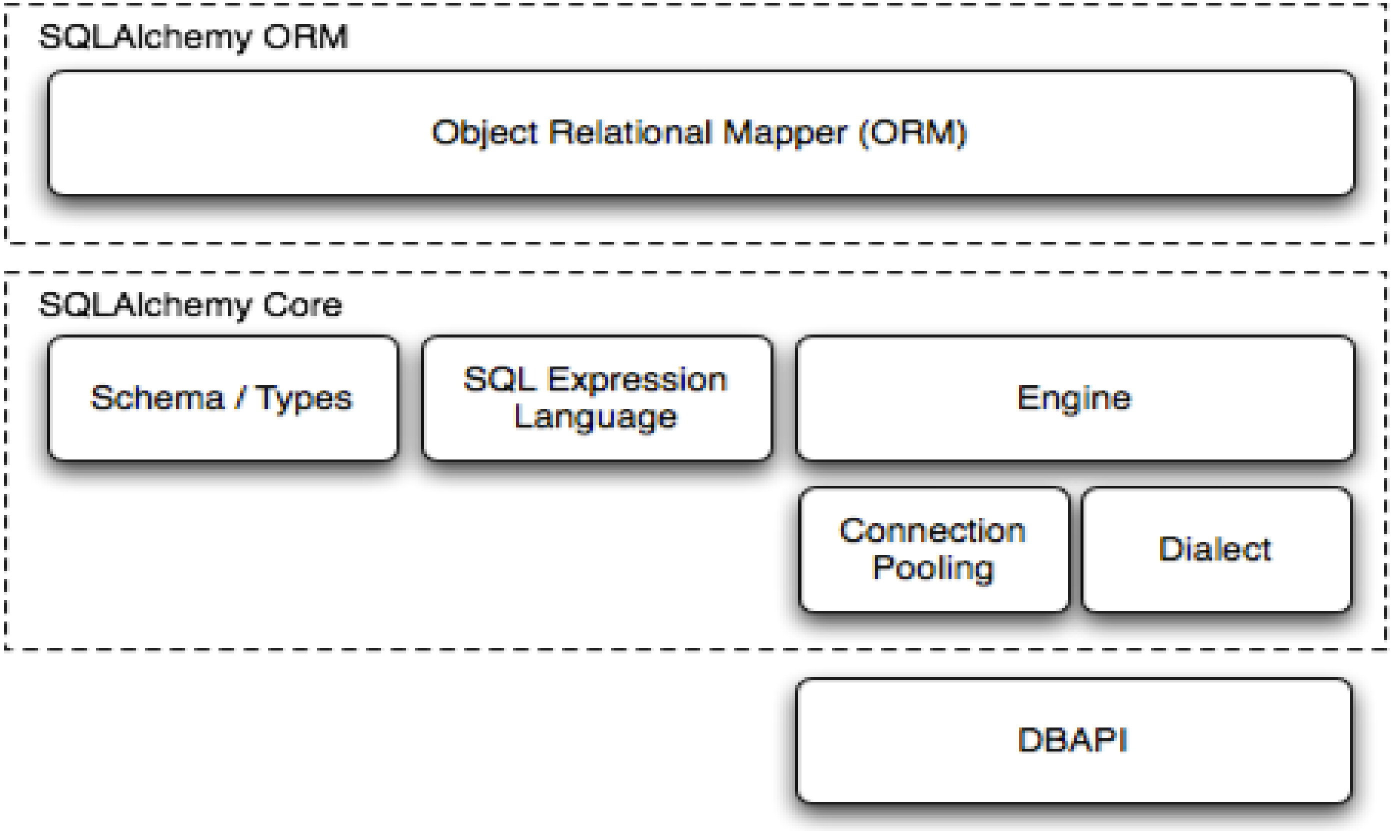
- The SQLAlchemy layered architecture
- Engines, connections and sessions
- SQLAlchemy as dispatcher of SQL commands
- The SQLAlchemy Core
- The SQLAlchemy Object Relational Mapper (ORM)

SQLALchemy

- The SQLAlchemy layered architecture
- Engines, connections and sessions
- SQLAlchemy as dispatcher of SQL commands
- The SQLAlchemy Core
- The SQLAlchemy Object Relational Mapper (ORM)

The SQLAlchemy layered architecture

- SQLAlchemy is an open-source SQL toolkit and object-relational mapper (ORM) for Python



Source: https://docs.sqlalchemy.org/en/14/intro.html

The SQLAlchemy layered architecture

- SQLAlchemy wraps the SQL DBAPI with two software layers:
  - SQLAlchemy Core

    The Core embeds a number of subsystems (including the engine) and provides classes and functions devised to perform queries to the DBAPI
  - SQLAlchemy ORM

    The ORM has been implemented according to the data mapper pattern, which facilitates the task of associating user-defined classes with database tables

A data mapper is a software layer that performs bidirectional transfers of data between a persistent data store and a running process, while keeping them independent of each other (Martin Fowler, David Rice: Patterns of Enterprise Application Architecture, Addison-Wesley, Boston, 2003.).

The SQLAlchemy layered architecture

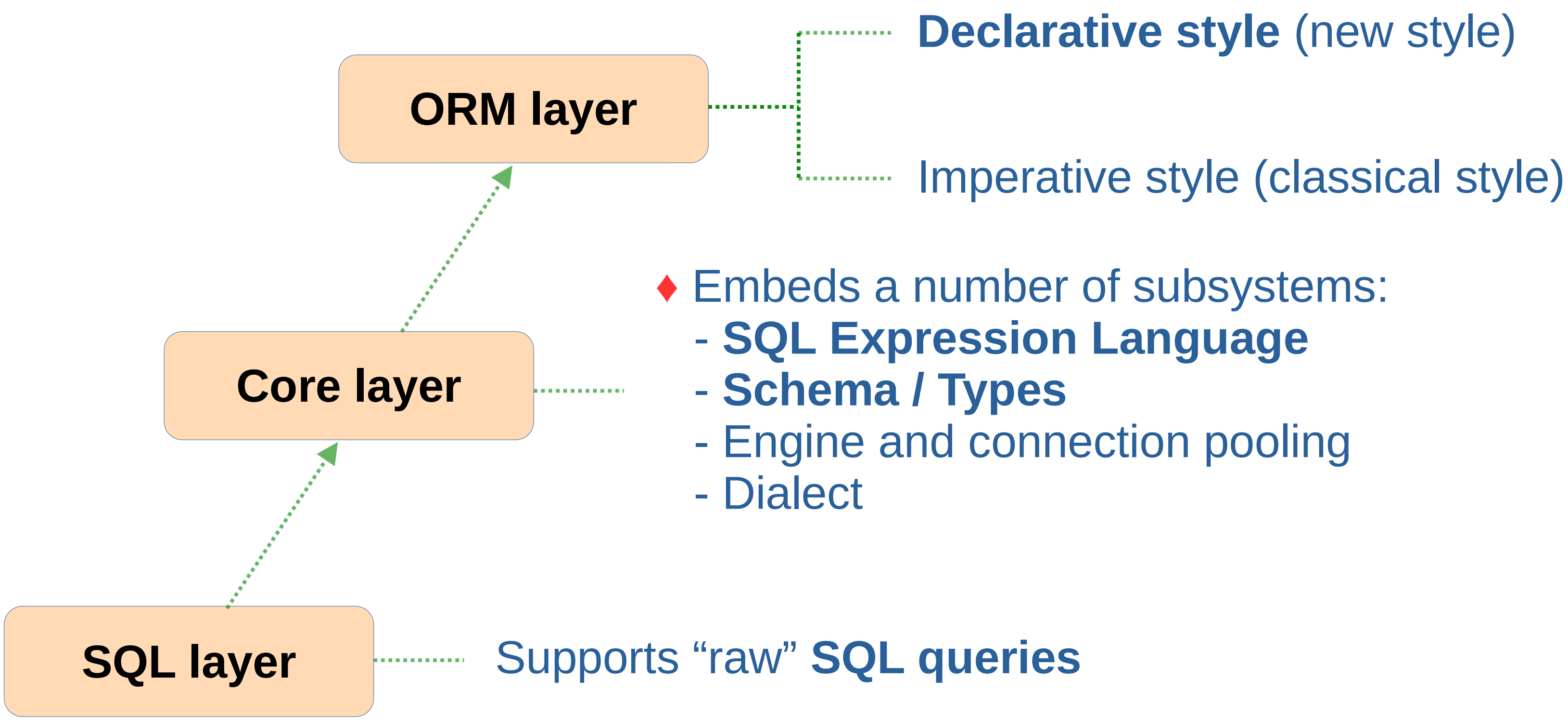
- There are basically three ways to enact an interaction between SQLAlchemy and the DBAPI:
  1) Standard SQL commands
  2) Core facilities (through the SQL Expression Language and schema definition language)
  3) ORM facilities (by mapping user-defined classes to DB tables)

The above ordering is followed in this tutorial

ORM user-defined classes can be associated to database tables using an imperative or declarative approach. The latter is the method suggested in the latest releases of the library.

The SQLAlchemy layered architecture

- SQLAlchemy abstraction layers at a glance



ORM layer

Declarative style (new style)

Imperative style (classical style)

Core layer

♦ Embeds a number of subsystems:
- **SQL Expression Language**
- **Schema / Types**
- Engine and connection pooling
- Dialect

SQL layer

Supports “raw” **SQL queries**

The SQLAlchemy layered architecture

- SQLAlchemy can handle several SQL dialects, i.e.:
  - Firebird
  - Microsoft SQL Server
  - MySQL
  - Oracle
  - PostgreSQL
  - SQLite
  - Sybase

SQLALchemy

- The SQLAlchemy layered architecture
- Engines, connections and sessions
- SQLAlchemy as dispatcher of SQL commands
- The SQLAlchemy Core
- The SQLAlchemy Object Relational Mapper (ORM)

Engines, connections and sessions

- To communicate with the DBAPI, SQLAlchemy provides the following classes:
  - Engine and Connection

    These classes are typically used to perform a) raw SQL queries or b) queries mediated by the SQLAlchemy Core
  - Session

    This class should be used with the ORM

Engines, connections and sessions

- The Engine class
  - An engine is the front door towards the DBAPI, as it can manage several connections
  - An Engine object is created by calling sqlalchemy.create_engine()
  - An Engine object can be used to execute requests to the DBAPI (through the method ‘execute’)

NB The call to create_engine should be made once for each database URL and held globally for the lifetime of a single running process.

Engines, connections and sessions

- Example of using the engine

```
from sqlalchemy import create_engine, text
url = *** a config string to be passed to the engine ***
engine = create_engine(url)
# print the version of PostgreSQL
version = engine.execute(text('select version()'))
print(f'Version of PostgreSQL: {version}')
```

Engines, connections and sessions

- The most general form of the URL string used to create the engine is:

```
dialect[+driver]://user[:password]@host[:port]/dbname
```

- Example
  - `SQL dialect and driver = postgresql, psycopg2`
  - `User and password = 'dida', 'didapass'`
  - `Host and port = localhost, 5432 # 5432 = default port`
  - `Database name = 'didattica'`

  - `URL to be used for connecting to the database:`

```
postgres+psycopg2://dida:didapass@localhost/didattica
```

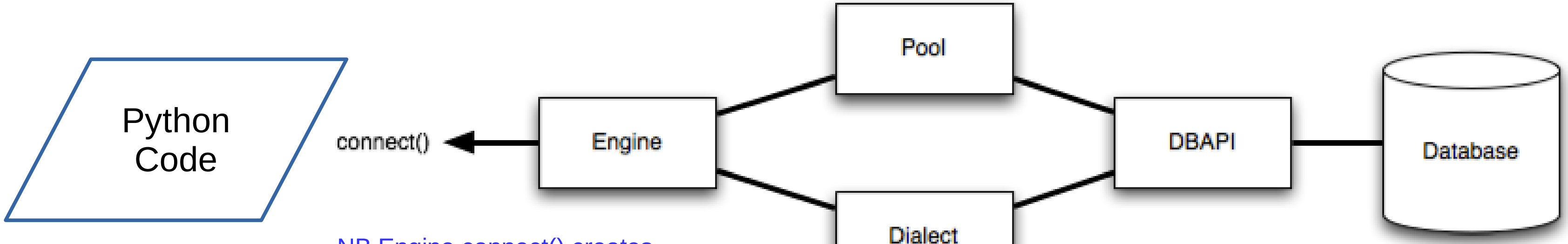
PS The default port (i.e., 5432) is typically not specified, whereas driver and password may be required.

Engines, connections and sessions

- The Connection class
  - The Connection class is entrusted with establishing a DBAPI connection
  - A Connection object is created by calling Engine.connect() and provides services for executing SQL statements in “raw” or Core format, as well as transaction control
  - It is highly preferable to use connections (rather than the engine) to interact with the DBAPI engine

Engines, connections and sessions

- Wrapping the DBAPI through Engine.connect()



NB Engine.connect() creates
a Connection object

Typical use of connections:

```
engine = create_engine(url) # typically created once in an application
with engine.connect() as connection:
  # Interact with the DBAPI using the connection
  query = … set up a query in raw SQL or SQL Core Expr. Language …
  result = connection.execute(query)
  … etc …
```

NB The connection is automatically closed on exit from the 'with' statement.

Engines, connections and sessions

- Example of using connections

```
from sqlalchemy import create_engine, text
url = *** a config string to be passed to the engine ***
engine = create_engine(url)
# print the version of PostgreSQL
with engine.connect() as connection:
  version = connection.execute(text('select version()'))
print(f'Version of PostgreSQL: {version}')
```

Engines, connections and sessions

- The Session class
  - The current session represents a “holding zone” for all the objects that have been loaded or associated with it during its lifespan
  - ORM objects are maintained inside the current session within a structure (called identity map) devised to maintain unique copies of each object
  - A Session class customized for the application needs can be created by using the service sqlalchemy.orm.sessionmaker()

NB The call to sessionmaker should be made once for each database URL and held globally for the lifetime of a single running process.

Engines, connections and sessions

- Example of using sessions

```
from sqlalchemy import create_engine
from sqlalchemy.orm import Session
url = *** same as before ***
engine = create_engine(url)
with Session(engine) as session:
  # do something with the ORM objects
  [...]
  session.commit() # commit the transaction...
```

SQLALchemy

- The SQLAlchemy layered architecture
- Engines, connections and sessions
- SQLAlchemy as dispatcher of SQL commands
- The SQLAlchemy Core
- The SQLAlchemy Object Relational Mapper (ORM)

SQLAlchemy as dispatcher of SQL commands

- Let’s see how “raw” SQL queries can be performed through SQLAlchemy
  - In this case, no in-memory representation of database tables or views occurs at the program level
  - In fact, the interaction with the DBAPI is obtained upon the creation of an engine and a connection, followed by the requests of executing standard SQL queries

SQLAlchemy as dispatcher of SQL commands

- Example of SQL query (using Engine+Connection)

```
from sqlalchemy import create_engine, text
driver = 'postgresql+psycopg2'
user, passwd = 'dida', 'didapass'
host, dbname = 'localhost', 'didattica'
url = f'{driver}://{user}:{passwd}@{host}/{dbname}'
engine = create_engine(url, echo=False)
with engine.connect() as connection:
  query = text("select * from segreteria.studenti")
  students = connection.execute(query)
for student in students: print(student)
```

SQLALchemy

- The SQLAlchemy layered architecture
- Engines, connections and sessions
- SQLAlchemy as dispatcher of SQL commands
- The SQLAlchemy Core
- The SQLAlchemy Object Relational Mapper (ORM)

The SQLAlchemy Core

- The Core features of SQLAlchemy allow to create Python objects that stand in foreground and represent tables and views of the underlying database
- An SQL table or view is represented by an object of class Table, whereas its attributes are represented as instances of the class Column

The SQLAlchemy Core

- Recall that source code aimed at implementing raw SQL queries or queries written in accordance with the Core programming style typically makes use of Engine and Connection objects

The SQLAlchemy Core

- Example of SQL query mediated by the Core

```
from sqlalchemy import create_engine, Table, MetaData
from sqlalchemy import select
user, passwd = 'dida', 'didapass'
dbname, schema = 'didattica', 'segreteria'
driver = 'postgresql+psycopg2'
url = f'{driver}://{user}:{passwd}@localhost/{dbname}'
engine = create_engine(url, echo=False)
metadata = MetaData(schema=schema, bind=engine)
students = Table("studenti", metadata, autoload=True)
with engine.connect() as connection:
  query = select(students)
  content = connection.execute(query)
for item in content: print(item)
```

The SQLAlchemy Core

- The MetaData class
  - A MetaData object stands behind any reflective operation performed by SQLAlchemy (e.g., loading the structure of a table defined in the database)
  - A MetaData object can (should) be bound to an Engine (or Connection) and to a schema
- Example

```
metadata = MetaData(schema='my_schema', bind=my_engine)
```

Reflection – In computer science, reflective programming or reflection is the ability of a running process to examine, introspect, and modify its own structure and behavior.

The SQLAlchemy Core

- The Table class
  - Any table or view in the database can be represented within the running process by an object of class Table
  - A Table object can be instantiated explicitly or loaded from the database thanks to the ‘autoload’ facility
  - Example

```
students = Table("studenti", metadata, autoload=True)
```

NB Note that the instantiation of a Table object always requires a MetaData object (for it embeds a collection of Table objects and their associated schema constructs).

The SQLAlchemy Core

- Some basic operations
  - Create a Table object from scratch
  - Create a Table object using the “autoload” facility
  - Insert tuples into a table
  - Update a table content
  - Select information from a table
  - Select information from multiple tables

Create a table object from scratch

Useful to create a database table with Python (DDL operation)

- Example

```
from sqlalchemy import create_engine, MetaData, Table, Column
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
columns = ( Column('matricola', Integer, primary_key=True), \
            Column('nome', VARCHAR(length=20)), \
            Column('cognome', VARCHAR(length=20)),
            Column('data_nascita', DATE()),
            Column('anno_corso', SMALLINT()
students = Table('studenti', metadata, *columns)
```

Note that a Table object is made up by columns (i.e., there is a “part-of” relation between a Table and its Column objects).

Create a table object using the “autoload” facility

Useful to interact with an existing database table with Python (DML operations)

- Example

```
from sqlalchemy import create_engine, MetaData, Table
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
students = Table('studenti', metadata, autoload=True)
print(repr(students)) # print out the table schema
```

NB The function ‘repr’ transforms an object into a string as the Python interpreter does (the conversion if often different from the one performed using ‘str’).

Insert tuples into a table

- Example

```
from sqlalchemy import create_engine, MetaData, Table
from sqlalchemy import insert
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
students = Table('studenti', metadata, autoload=True)
query = insert(students).values(
          matricola='45825',
          cognome='Secci', nome='Marco',
          data_nascita='1986-3-2', anno_corso=2 )
with engine.connect() as connection:
  connection.execute(query)
```

Update a table content

- Example

```
from sqlalchemy import create_engine, MetaData, Table
from sqlalchemy import update
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
students = Table('studenti', metadata, autoload=True)
query = update(students)\
          .where(students.c.matricola == '458250')\
          .values(matricola = '458350')
with engine.connect() as connection:
  connection.execute(query)
```

NB Note the syntax for accessing attributes when using the SQL Expression Language. One may also use: `students.columns.matricola`.

Select information from a table (without ‘where’)

- Example

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy import select
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
students = Table('studenti', metadata, autoload=True)
query = select(students) # select all students
with engine.connect() as connection:
  content = connection.execute(query)
for item in content: print(item)
```

Select information from a table (with ‘where’)

- Example

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy import select
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
students = Table('studenti', metadata, autoload=True)
query = select(students).where(students.c.anno_corso==1)
with engine.connect() as connection:
  content = connection.execute(query)
for item in content: print(item)
```

Select information from multiple tables

- Example (using join)

```
from sqlalchemy import create_engine, MetaData, join
url = *** same as before ***
engine = create_engine(url, echo=True)
metadata = MetaData(schema='segreteria', bind=engine)
exams = Table("esami", metadata, autoload=True)
students = Table("studenti", metadata, autoload=True)
es_view = students.join(exams,
            students.c.matricola == exams.c.studente)
query = select([students]).select_from(es_view)\
          .where(exams.c.corso == 'C03')
With engine.connect() as connection:
  content = connection.execute(query)
for item in content: print(item)
```

AND and OR in ‘where’ clauses

- Usually, where clauses contain and/or expressions
- The Core provides and_() and or_() functions, which enforce the semantics of “and” and “or” operators
- Example [for the and_() function]

```
query = select([students])\
          .where( and_( students.c.nome == 'Paola', \
                        students.c.cognome == 'Melis' ) )
```

- Alternative syntax with an infix operator (better)

```
query = select([students])\
          .where( ( students.c.nome == 'Paola') & \
                  ( students.c.cognome == 'Melis' ) )
```

The infix operator for the or_ clause is the pipe character (i.e., “|”).

ASC and DESC (in orderBy clauses)

- The asc() and desc() functions are used to specify which ordering must be followed to order the results of a query (as made by the SQL “order by” clause)
- Example [for the asc() function]

```
query = select([students])\
          .order_by(asc(students.c.cognome))
```

SQLALchemy

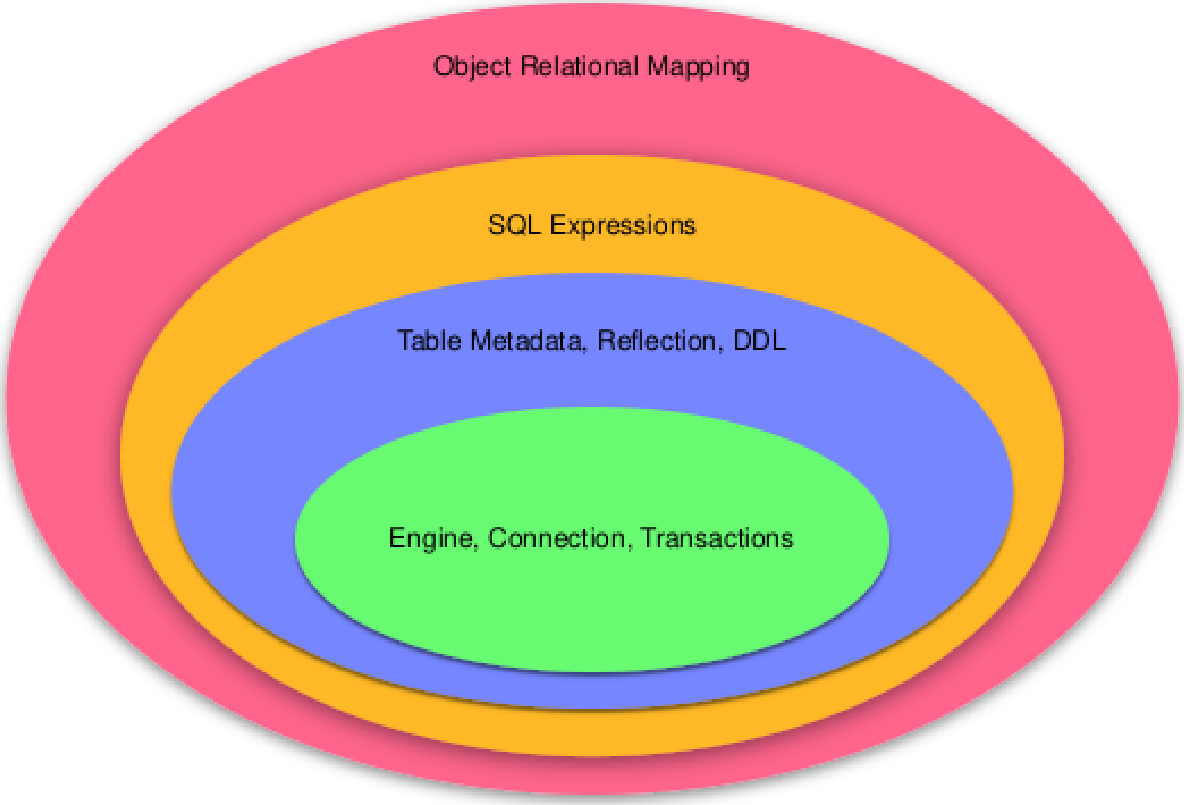
- The SQLAlchemy layered architecture
- Engines, connections and sessions
- SQLAlchemy as dispatcher of SQL commands
- The SQLAlchemy Core
- The SQLAlchemy Object Relational Mapper (ORM)

The SQLAlchemy Object Relational Mapper

- Using the ORM data mapper, database tables and views are mapped to user-defined classes
- The ORM takes care of any issue concerning the dialog between the running process (written in Python) and the database engine (e.g., postgres/psql)
- No need to write code for accessing databases using an SQL syntax or using the SQLAlchemy Core

The SQLAlchemy Object Relational Mapper

- The SQLAlchemy abstraction layers according to its creator (Mike Bayer)


Object Relational Mapping
SQL Expressions
Table Metadata, Reflection, DDL
Engine, Connection, Transactions

From: https://github.com/zzzeek/sqla_tutorial/

The SQLAlchemy Object Relational Mapper

- Two different styles hold in SQLAlchemy for mapping user-defined classes with database tables:
  - **Declarative** (aka new mapping style)
  - **Imperative** (aka classical mapping style)

NB The declarative mapping style is used in these slides.

The SQLAlchemy Object Relational Mapper

- Declarative mapping (new mapping style)
  - The most common pattern is to first construct a base class using the DeclarativeBase superclass
  - The resulting base class will apply the declarative mapping process to all subclasses that will be derived from it

The SQLAlchemy Object Relational Mapper

- Example of declarative mapping

```
from sqlalchemy import Integer, MetaData
from sqlalchemy.types import DateTime
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(metadata=MetaData(schema="segreteria"))

class Studente(Base):
    __tablename__ = "studenti"
    matricola = Column(String(6),primary_key=True)
    cognome = Column(String(20))
    nome = Column(String(20))
    data_nascita= Column(DateTime)
    anno_corso = Column(Integer)
```

The SQLAlchemy Object Relational Mapper

- A mapping between a user-defined class and a database table can be direct or mediated
  - Direct mapping

    Direct link to the database (the Core is not used, at least in principle)
  - Mediated mapping

    Link to the database mediated by a Table (the Core stands in between the ORM and the database)

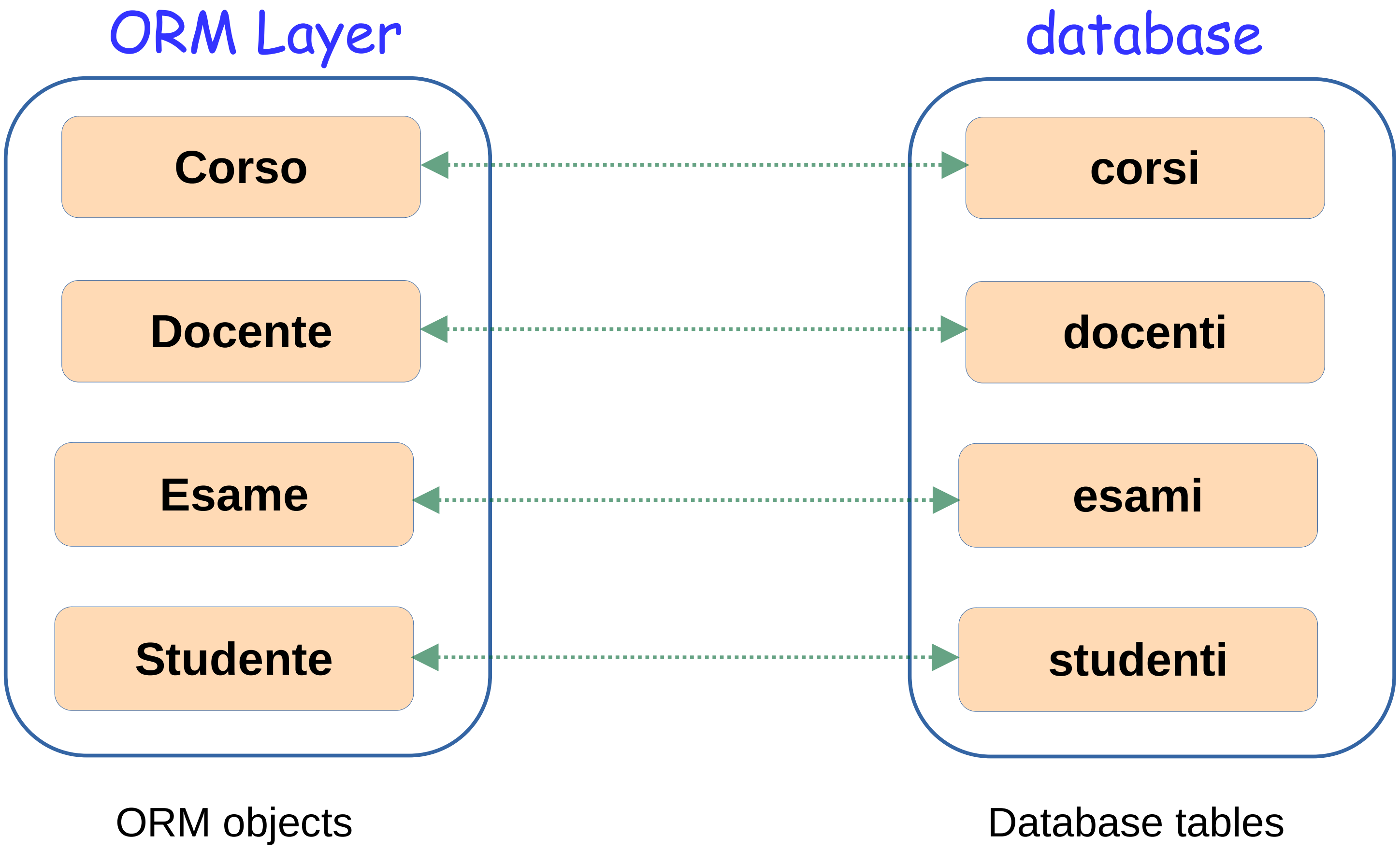
The SQLAlchemy Object Relational Mapper

- Direct mapping
  - To directly link a user-defined class (say Studente) to a database table (say 'studenti') a “__tablename__” assignment must be performed within Studenti
  - Example

```
from sqlalchemy import MetaData
from sqlalchemy.ext.declarative import declarative_base
metadata = MetaData(schema="segreteria")
Base = declarative_base(metadata=metadata)
class Studente(Base):
   __tablename__ = 'studenti' # name of a database table
   pass # more on Studente...
```

The SQLAlchemy Object Relational Mapper

- Interaction with database tables through __tablename__



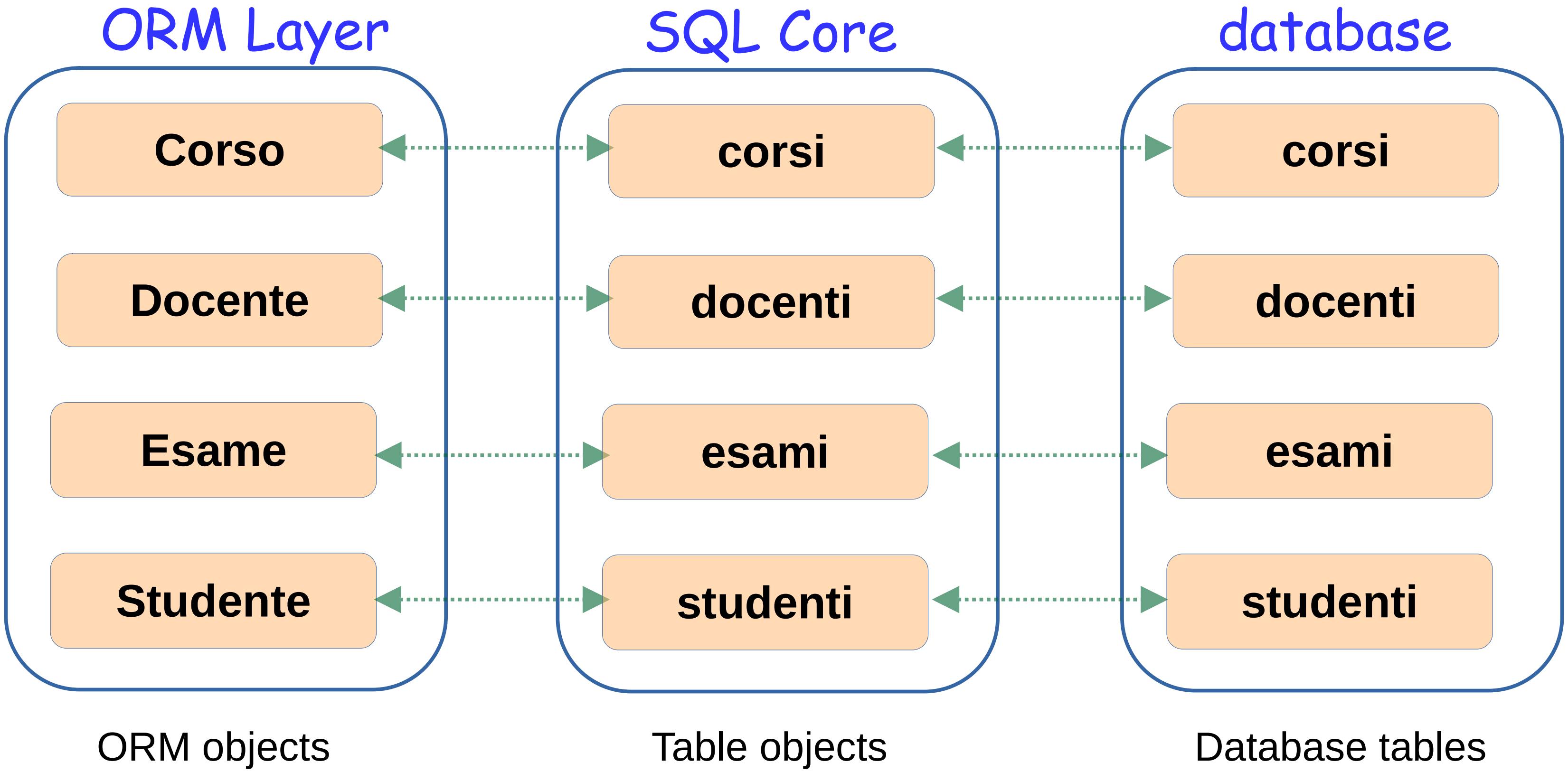
The SQLAlchemy Object Relational Mapper

- Mediated mapping
  - In this case we assume that a “skeleton” of the database table already exists in memory (e.g., the class studenti, instance of Table, which is linked to the database table ‘studenti’)
  - In this case the user-defined class Studente can be linked to to the database table studenti by referring to a Table
  - Example

```
from sqlalchemy import MetaData
from sqlalchemy.ext.declarative import declarative_base
metadata = MetaData(schema="segreteria")
Base = declarative_base(metadata=metadata)
class Studente(Base):
  __table__ = studenti # instance of Table
  pass # more on Studente...
```

The SQLAlchemy Object Relational Mapper

- Interaction with database tables through __table__



NB The function sqlalchemy.orm.mapper takes care of the interaction between ORM and Table objects. However, using declarative style, the mapping is automatically handled by SQLAlchemy.

The SQLAlchemy Object Relational Mapper

- As a matter of fact, SQL Core Tables are used in both cases (i.e., direct and mediated mapping)
- In fact, with direct mapping:
  - The table that stands behind the scenes at the SQL Core layer can be accessed by means of the __table__ slot
  - The mapper that handles the interaction between an ORM object and its corresponding Table object can be accessed by means of the __mapper__ slot

The SQLAlchemy Object Relational Mapper

- Source code written in accordance with the ORM data mapper typically makes use of Session objects
- This happens because the typical operations of commit and rollback are highly integrated with sessions
- Let’s see an example of declarative mapping at work...

The SQLAlchemy Object Relational Mapper

- Example

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy import Column, String, Integer, Date
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(metadata=MetaData(schema="segreteria"))

class Studente(Base):
  __tablename__ = "studenti"
  matricola = Column(Integer, primary_key=True)
  cognome = Column(String)
  nome = Column(String)
  data_nascita = Column(Date)
  anno_corso = Column(Integer)
  def __str__(self):
    slots = ('matricola','cognome','nome','data_nascita','anno_corso')
    return ", ".join([ f"{getattr(self,slot)}" for slot in slots ])
```

The SQLAlchemy Object Relational Mapper

- Example (test code)

```
if __name__ == '__main__':

  dialect, driver = "postgresql", "psycopg2"
  user, passwd = "dida", "didapass"
  host = "localhost" # using the default port 5432
  dbase = "didattica"

  url = f"{dialect}+{driver}://{user}:{passwd}@{host}/{dbase}"
  engine = create_engine(url, echo=False)

  Session = sessionmaker(bind=engine)

  with Session() as session:
    students = session.query(Studente).all()
    # print all student info
    print("\n*** Printing students (all attributes) ***\n")
    for k, s in enumerate(students): print(f"[{k:2d}] {s}")
```

Set global vars

Create the engine

Build the Session class

Get and print all student info

The SQLAlchemy Object Relational Mapper

- Defining binary relationships
  - Depending on the definition of database tables, some relationships may hold among them
  - At the database level, these relationships are modeled using foreign keys
  - These relationships can be:
    - One-to-one
    - One-to-many (and vice versa)
    - Many-to-many

The SQLAlchemy Object Relational Mapper

- Difference between the cited kinds of binary relationships
  - One-to-one
    - No need to model them at the program level
    - They are already modeled at the DB level by means of foreign keys
  - One-to-many (and vice versa)
    - Everything goes as if the table that stands on the “one” side is “populated” by several individuals
  - Many-to-many
    - Corresponds to a one-to-many relation on both sides

The SQLAlchemy Object Relational Mapper

- Example of one-to-many relationship

```
 from sqlalchemy import Column, String
 from sqlalchemy import ForeignKey
 from sqlalchemy.orm import relationship

 class Docente(Base):
   __tablename__ = 'docenti'
   cod_docente = Column(String, primary_key=True)
   cognome = Column(String, nullable=False)
   nome = Column(String, nullable=False)
   indirizzo = Column(String, default="")
   lista_corsi = relationship('Corso') # one-to-many relationship

class Corso(Base):
  __tablename__ = 'corsi'
  cod_corso = Column(String, primary_key=True)
  nome = Column(String, nullable=False)
  docente = Column(String, ForeignKey("docenti.cod_docente"))
```

The SQLAlchemy Object Relational Mapper

- Example of many-to-many relationship

```
from sqlalchemy import Column, Integer, String
from sqlalchemy import ForeignKey
from sqlalchemy.orm import relationship

class Studente(Base):
  __tablename__ = 'studenti'
  matricola = Column(String, primary_key=True)
  cognome = Column(String, nullable=False)
  nome = Column(String, nullable=False)
  lista_esami = relationship('Esame', back_populates='lista_studenti')

class Esame(Base):
  __tablename__ = 'esami'
  studente = Column(String,ForeignKey('studenti.matricola',primary_key=True)
  corso = Column(String,ForeignKey('corsi.cod_corso',primary_key=True)
  voto = Column(Integer, nullable=False)
  lista_studenti = relationship('Studente', back_populates='lista_esami')
```

The SQLAlchemy Object Relational Mapper

End with SQLAlchemy SLIDES