

# SQLAlchemy

- [Sqlalchemy](#)
- [ORM режим](#)
- [Core режим](#)
- [Пример проекта](#)

# Sqlalchemy

## Установка

Ядро

```
pip install sqlalchemy
```

Драйвер для postgres, mysql

```
pip install psycopg2
```

```
pip install psycopg2-binary
```

```
pip install pymysql
```

## Подключение

```
from sqlalchemy import create_engine  
engine = create_engine('postgresql+psycopg2://username:password@localhost:5432/mydb')  
engine = create_engine('mysql+pymysql://cookiec:chip@mysql01.com/cookies', pool_recycle=3600)  
engine = create_engine('sqlite:///cookies.db')  
engine2 = create_engine('sqlite:///memory:')
```

Доп. ключи через запятую:

|                 |   |
|-----------------|---|
| echo            | булево. Лог запросов. По-умолчанию False.   |
| encoding        | строка. По-умолчанию utf-8  |
| isolation_level | уровень изоляции  |
| pool_recycle    | число секунд для переподключения, желательно выставлять 3600. При работе с mysql соединение может быть активным до 4 часов. |

Создание engine не создает фактического соединения с БД.

```
connection = engine.connect()
```

Сырой запрос:

```
result = connection.execute("select * from orders").fetchall()
```

## Структура фреймворка

Таблицы -> MetaData -> Engine -> Dialect -> DB

**MetaData:** объект, в котором таблицы, индексы,...

- Может получать информацию о существующих сейчас сущностях в БД
- Может хранить шаблоны именования индексов и ограничений (constraint). Т е перед началом проекта добавляется настройка именования, затем при создании/удалении все ок. Это словарь ограничение:шаблон

| Ограничение или индекс | Описание             |
|------------------------|----------------------|
| ix                     | обычный индекс       |
| uq                     | уникальный индекс    |
| ck                     | ограничение проверки |
| fk                     | foreign-ключ         |
| pk                     | primary-ключ         |

```
from sqlalchemy import MetaData
convention = {
    'all_column_names': lambda constraint, table: '_'.join([
        column.name for column in constraint.columns.values()
    ]),
    'ix': 'ix__%(table_name)s__%(all_column_names)s',
    'uq': 'uq__%(table_name)s__%(all_column_names)s',
    'ck': 'ck__%(table_name)s__%(all_column_names)s',
    'fk': ('fk__%(table_name)s__%(all_column_names)s' '%(referred_table_name)s'),
    'pk': 'pk__%(table_name)s'
}
metadata_obj = MetaData(naming_convention=convention)
```

- Должен быть инициализирован до обращения к нему в таблицах
- Изначально пустой объект, можно или вручную занести данные, или получить из базы.
- Получение информации об одной таблице по имени
  - Нельзя получить одновременно две таблицы. Либо одна, либо вся база
  - Нельзя (и одна таблица, и база) получить ограничения (CONSTRAINT), комментарии, триггеры, значения по умолчанию. Но можно вручную добавить данные. Но похоже проще импортировать описание.

```
from sqlalchemy import ForeignKeyConstraint
album.append_constraint(ForeignKeyConstraint(['ArtistId'], ['artist.ArtistId']))
```

- при получении базы, имена таблиц с большой и маленькой буквы. Т е удваивается количество объектов.

```

engine = create_engine(...)
metadata = MetaData()
cookie_tbl = Table('cookies', metadata, autoload_with=engine)
s = select(cookie_tbl)
with engine.connect() as conn:
    m = conn.execute(s)
    print(m.keys())

```

- Получение информации обо всей базе

```

from sqlalchemy import create_engine, MetaData

engine = create_engine(...)
metadata = MetaData()
metadata.reflect(bind=engine)
for table in metadata.sorted_tables:
    print(table.name)
#Потом получить объект таблицы:
mytable = metadata.tables['mytable']

```

- Получение информации обо всей базе через ORM + Automap

```

from sqlalchemy.ext.automap import automap_base
from sqlalchemy import create_engine

Base = automap_base()
engine = create_engine('sqlite:///Chinook_Sqlite.sqlite')
Base.prepare(engine, reflect=True)
# данные о классах загружены.
#Например для получения списка классов:
Base.classes.keys()
Artist = Base.classes.Artist # создание классов
#Внешние связи - в свойстве <related_object>_collection
artist = session.query(Artist).first()
for album in artist.album_collection:
    print('{} - {}'.format(artist.Name, album.Title))

```

**Engine:** Скрывает пул подключений и диалект.

**Dialect:** Скрывает детали реализации в конкретной базе

**Core:** SQL в чистом виде

**ORM:** абстракции

## Работа с данными

### INSERT

Вариант 1:

```
ins = cookies.insert().values(  
    cookie_name="chocolate chip",  
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",  
    cookie_sku="CC01",  
    quantity="12",  
    unit_cost="0.50"  
)  
  
result = connection.execute(ins)  
print(result.inserted_primary_key)
```

Вариант 2:

```
from sqlalchemy import insert  
ins = insert(cookies).values(  
    cookie_name="chocolate chip",  
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",  
    cookie_sku="CC01",  
    quantity="12",  
    unit_cost="0.50"  
)
```

Вариант 3

```
ins = cookies.insert()  
result = connection.execute(  
    ins,  
    cookie_name='dark chocolate chip',  
    cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',  
    cookie_sku='CC02',  
    quantity='1',  
    unit_cost='0.75'  
)  
result.inserted_primary_key
```

Вариант 4

```
inventory_list = [  
    {
```

```
'cookie_name': 'peanut butter',
'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html',
'cookie_sku': 'PB01',
'quantity': '24',
'unit_cost': '0.25'
},
{
'cookie_name': 'oatmeal raisin',
'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html',
'cookie_sku': 'EWW01',
'quantity': '100',
'unit_cost': '1.00'
}
]
result = connection.execute(ins, inventory_list)
```

## SELECT

### Вариант 1

```
from sqlalchemy.sql import select
s = select([cookies])
rp = connection.execute(s)
results = rp.fetchall()
```

### Вариант 2

```
s = cookies.select()
rp = connection.execute(s)
results = rp.fetchall()
```

### Вариант 3

```
s = cookies.select()
rp = connection.execute(s)
for record in rp:
    print(record.cookie_name)
```

## SELECT определенных столбцов

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
```

## ORDERING

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
```

## Прямая сортировка

```
s = s.order_by(cookies.c.quantity)
```

## Обратная сортировка

```
s = s.order_by(desc(cookies.c.quantity))
```

```
rp = connection.execute(s)
```

## LIMITING

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
s = s.order_by(cookies.c.quantity)
s = s.limit(2)
rp = connection.execute(s)
```

## Встроенные функции SQL

Сумма: func.sum

```
from sqlalchemy.sql import func
s = select([func.sum(cookies.c.quantity)])
rp = connection.execute(s)
print(rp.scalar())
```

Количество: func.count

```
s = select([func.count(cookies.c.cookie_name)])
rp = connection.execute(s)
record = rp.first()
print(record.keys()) # ключи могут быть разные.
print(record.count_1)
```

Название свойства: label

```
s = select([func.count(cookies.c.cookie_name).label('inventory_count')])
rp = connection.execute(s)
record = rp.first()
print(record.keys())
print(record.inventory_count)
```

## WHERE

Их можно комбинировать, используется AND

```
s = select([cookies]).where(cookies.c.cookie_name == 'chocolate chip')
rp = connection.execute(s)
```

Варианты модификаторов:

- `between(cleft, cright)` Значение столбца между `cleft` и `cright`
- `concat(column_two)` Объединение `column` и `column_two`
- `distinct()` Находит только уникальные значения в столбце
- `in_([list])` Только если значения столбца в списке
- `is_(None)` Проверка на пустые значения
- `contains(string)` Значение столбца содержит строку
- `endswith(string)` Оканчивается строкой, зависит от больших букв
- `like(string)` зависит от больших букв
- `startswith(string)` зависит от больших букв
- `ilike(string)` не зависит от больших букв
- Есть отрицательные модификаторы `not<method>`, исключение - метод `isnot()`

### Пример для LIKE

```
s = select([cookies]).where(cookies.c.cookie_name.like('%chocolate%'))
rp = connection.execute(s)
for record in rp.fetchall():
    print(record.cookie_name)
```

Модификация полученных данных по шаблону

Вариант 1: к каждому значению столбца

```
s = select([cookies.c.cookie_name, 'SKU-' + cookies.c.cookie_sku]) - добавит строку 'SKU-'
```

Вариант 2: через функцию `cast`

```
from sqlalchemy import cast
s = select([cookies.c.cookie_name,
           cast((cookies.c.quantity * cookies.c.unit_cost),
               Numeric(12,2)).label('inv_cost')])
for row in connection.execute(s):
    print('{} - {}'.format(row.cookie_name, row.inv_cost))
```

### Логические функции

```
from sqlalchemy import and_, or_, not_
s = select([cookies]).where(
    and_(
        cookies.c.quantity > 23,
        cookies.c.unit_cost < 0.40
    )
)
```

### Извлечение данных



```
first_row = results[0]
first_row[1]
first_row.cookie_name
first_row[cookies.c.cookie_name]
```

Для `gr` есть следующие варианты, однако без `limit()` извлекаются все данные, затем одна строка:

- `first()` - Первая запись и закрытие соединения
- `fetchone()` - Одна запись и оставляет открытый курсор для последующих запросов
- `scalar()` - Одно значение если запрос возвращает одно значение в одной строке
- `gr.keys()` - список столбцов

## Обновление данных

```
from sqlalchemy import update
u = update(cookies).where(cookies.c.cookie_name == "chocolate chip")
u = u.values(quantity=(cookies.c.quantity + 120))
result = connection.execute(u)
```

## Удаление данных

```
from sqlalchemy import delete
u = delete(cookies).where(cookies.c.cookie_name == "dark chocolate chip")
result = connection.execute(u)
```

## JOIN

```
columns = [orders.c.order_id, users.c.username, users.c.phone,
           cookies.c.cookie_name, line_items.c.quantity,
           line_items.c.extended_cost]
cookiemon_orders = select(columns)
cookiemon_orders = cookiemon_orders.select_from(orders.join(users).join(
    line_items).join(cookies)).where(users.c.username ==
    'cookiemon')
result = connection.execute(cookiemon_orders).fetchall()
```

## OUTER JOIN

Для получения обратной статистики

```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username)
result = connection.execute(all_orders).fetchall()
```

Есть поддержка ALIAS

## Группировка данных

```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username)
result = connection.execute(all_orders).fetchall()
```

## Объединение запросов по условию

```
def get_orders_by_customer(cust_name, shipped=None, details=False):
    columns = [orders.c.order_id, users.c.username, users.c.phone]
    joins = users.join(orders)
    if details:
        columns.extend([cookies.c.cookie_name, line_items.c.quantity,
                        line_items.c.extended_cost])
        joins = joins.join(line_items).join(cookies)
    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins)

    cust_orders = cust_orders.where(users.c.username == cust_name)
    if shipped is not None:
        cust_orders = cust_orders.where(orders.c.shipped == shipped)
    result = connection.execute(cust_orders).fetchall()
    return result

get_orders_by_customer('cakeeater')
get_orders_by_customer('cakeeater', details=True)
get_orders_by_customer('cakeeater', shipped=True)
get_orders_by_customer('cakeeater', shipped=False)
get_orders_by_customer('cakeeater', shipped=False, details=True)
```

# ORM режим

Таблица это класс с требованиями:

- Потомок объекта, возвращаемого функцией `declarative_base`
- Включает `__tablename__` с именем таблицы
- Включает 1+ атрибутов, являющихся объектом `Column`
- При определении не включает имя столбца в конструкторе `Column`, имя столбца = имя атрибута
- 1+ атрибутов определяют первичный ключ
- `__table_args__` свойства таблицы (ограничения,...)

```
from sqlalchemy import Table, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Cookie(Base):
    __tablename__ = 'cookies'
    __table_args__ = (CheckConstraint('quantity >= 0', name='quantity_positive'),)
    cookie_id = Column(Integer(), primary_key=True)
    cookie_name = Column(String(50), index=True)
    quantity = Column(Integer())
```

## Создание таблиц

```
from sqlalchemy import create_engine
from dataclasses import Base

engine = create_engine(...)
Base.metadata.create_all(engine)
```

## Ограничения

```
__table_args__ = (ForeignKeyConstraint(['id'], ['other_table.id']), CheckConstraint(unit_cost >= 0.00,
name='unit_cost_positive'))
```

## Внешние связи

- Определяется столбец с ForeignKey
- Определяется дополнительный атрибут с relationship и необязательным backref
- При определении backref, relationship будет определен в указанном классе с указанным именем.

Один к одному:

```
cookie = relationship("Cookie", uselist=False)
```

Один ко многим:

```
user = relationship("User", backref=backref('orders'))
```

На себя (дерево) - неоднозначное решение.

```
class Employee(Base):
    __tablename__ = 'employees'
    id = Column(Integer(), primary_key=True)
    manager_id = Column(Integer(), ForeignKey('employees.id'))
    name = Column(String(255), nullable=False)
    manager = relationship("Employee", backref=backref('reports'), remote_side=[id])
```

## Сессии

В ORM режиме, сессия упаковывает

- соединение с БД через engine и предоставляет словарь объектов, загруженных через сессию или ассоциированных с сессией
- транзакции, которые открыты до коммита сессии

Это похожая на хэш-систему, состоящую из списка объектов, таблиц и ключей. Сессия создается через sessionmaker, один раз. Соединяется с базой в момент необходимости

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///memory:')
Session = sessionmaker(bind=engine)
session = Session()
```

## Состояния объекта в сессии:

|            |  |
|------------|--|
| Transient  | Объект не в сессии и не в БД                                   |
| Pending    | Объект добавлен в сессию add(), но не flush или commit         |
| Persistent | Объект в сессии имеет связанную запись в БД                    |
| Detached   | Объект больше не в сессии, но в БД есть соответствующая запись |
| Modified   | Объект изменен   |

## Просмотр состояния:

```
from sqlalchemy import inspect
insp = inspect(cc_cookie)
```

## Отключение объекта от сессии:

```
session.expunge(cc_cookie)
```

## Просмотр списка атрибутов и выяснение, что было модифицировано

```
for attr, attr_state in insp.attrs.items():
    if attr_state.history.has_changes():
        print('{ }: { }'.format(attr, attr_state.value))
        print('History: { }\n'.format(attr_state.history))
```

## Добавление данных

Создаем объект класса, добавляем в сессию и коммитим. Множественное добавление данных:

```
dcc = Cookie(...)
mcc = Cookie(...)
session.add(dcc)
session.add(mcc)
session.flush()
```

flush: не выполняет коммит и не завершает транзакцию, но получает id. Потом нужно в пределах сессии сделать commit. Commit в пределах чужой сессии не влияет. Дальше можно использовать объект.

Если дальше не нужно выполнять операции с объектами:

```
session.bulk_save_objects([dcc,mcc])
session.commit()
```

## Получение данных

|                       |  |
|-----------------------|--|
| <code>all()</code>    | все  |
| <code>first()</code>  | возвращает одну запись если она единственная и закрывает соединение          |
| <code>one()</code>    | возвращает одну запись и оставляет соединение. Аккуратно!                    |
| <code>scalar()</code> | возвращает одно значение если результат запроса одна строка с одним столбцом |

```
cookies = session.query(Cookie).all()
print(cookies)
```

Если использовать через итератор, то без `all()`:

```
for cookie in session.query(Cookie):
    print(cookie)
```

Получение определенных столбцов:

```
cookies = session.query(Cookie.cookie_id).all()
```

Сортировка:

```
session.query(Cookie).order_by(Cookie.quantity).all()
.order_by(desc(Cookie.quantity))
```

Ограничения через срезы или `.limit(2)`

Встроенные функции:

```
from sqlalchemy import func
inv_count = session.query(func.sum(Cookie.quantity)).scalar()
print(inv_count)
rec_count = session.query(func.count(Cookie.cookie_name)).first()
```

Метки: можно добавить для дальнейшего обращения

```
rec_count = session.query(func.count(Cookie.cookie_name).label('inventory_count')).first()
print(rec_count.inventory_count)
```

## Фильтрация

```
record = session.query(Cookie).filter(cookie_name=='chocolate chip').first()
record = session.query(Cookie).filter_by(cookie_name='chocolate chip').first()
query = session.query(Cookie).filter(Cookie.cookie_name.like('%chocolate%'))
query = session.query(Cookie).filter(Cookie.quantity > 23, Cookie.unit_cost < 0.40)
```

## Изменение строк при выводе

```
results = session.query(Cookie.cookie_name, 'SKU-' + Cookie.cookie_sku).all()
query = session.query(Cookie.cookie_name,
    cast((Cookie.quantity * Cookie.unit_cost),
        Numeric(12,2)).label('inv_cost'))
```

## Join:

```
query = session.query(Order.order_id, User.username)
results = query.join(User).all()

query = session.query(User.username, func.count(Order.order_id))
query = query.outerjoin(Order).group_by(User.username)
```

## Group:

```
query = session.query(User.username, func.count(Order.order_id))
query = query.outerjoin(Order).group_by(User.username)
```

## Сырые запросы

```
from sqlalchemy import text
query = session.query(User).filter(text("username='cookiemon'"))
```

## Обновление данных

### Через объект

```
query = session.query(Cookie)
cc_cookie = query.filter(Cookie.cookie_name == "chocolate chip").first()
cc_cookie.quantity = cc_cookie.quantity + 120
```

```
session.commit()
```

Через метод update

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "chocolate chip")
query.update({Cookie.quantity: Cookie.quantity - 20})
```

### Удаление

```
session.delete(dcc_cookie)
session.commit()
```

### Исключения

```
from sqlalchemy.orm.exc import MultipleResultsFound
try:
    results = session.query(Cookie).one()
except MultipleResultsFound as error:
    print('We found too many cookies... is that even possible?')
```

### Транзакции

В ORM транзакция создается автоматически до очередного коммита.

```
session.add(order)
try:
    session.commit()
except IntegrityError as error:
    session.rollback()
```

### Пример структурирования

db.py

```
from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine)

class DataAccessLayer:
    connection = None
    engine = None
    conn_string = None
```



```

metadata = MetaData()
cookies = Table('cookies',
                metadata,
                Column('cookie_id', Integer(), primary_key=True),
                Column('cookie_name', String(50), index=True),
                Column('cookie_recipe_url', String(255)),
                Column('cookie_sku', String(55)),
                Column('quantity', Integer()),
                Column('unit_cost', Numeric(12, 2))
            )

```

□□

```

def db_init(self, conn_string):
    self.engine = create_engine(conn_string or self.conn_string)
    self.metadata.create_all(self.engine)
    self.connection = self.engine.connect()
dal = DataAccessLayer()

```

## app.py

```

from db import dal
from sqlalchemy.sql import select

def get_orders_by_customer(cust_name, shipped=None, details=False):
    columns = [dal.orders.c.order_id, dal.users.c.username, dal.users.c.phone]
    joins = dal.users.join(dal.orders)
    if details:
        columns.extend([dal.cookies.c.cookie_name,
                        dal.line_items.c.quantity,
                        dal.line_items.c.extended_cost])
        joins = joins.join(dal.line_items).join(dal.cookies)
    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins).where(
        dal.users.c.username == cust_name)
    if shipped is not None:
        cust_orders = cust_orders.where(dal.orders.c.shipped == shipped)
    return dal.connection.execute(cust_orders).fetchall()

```

## test.py

```
import unittest

class TestApp(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        dal.db_init('sqlite:///memory:')

    def test_one(self):
        res = get_orders_by_customer("", False)
        self.assertEqual(res, [])
```

# Core режим

Сначала необходимо определить, как данные хранятся в таблице. Варианты определения:

- Объект Table
- Декларативный класс
- Получение структуры из базы данных

## Сопоставление типов

| SQLAlchemy  | Python             | SQL                         |
|-------------|--------------------|-----------------------------|
| BigInteger  | int                | BIGINT                      |
| Boolean     | bool               | BOOLEAN or SMALLINT         |
| Date        | datetime.date      | DATE (SQLite: STRING)       |
| DateTime    | datetime.datetime  | DATETIME (SQLite: STRING)   |
| Time        | datetime.time      | DATETIME                    |
| Enum        | str                | ENUM or VARCHAR             |
| Float       | float or Decimal   | FLOAT or REAL               |
| Integer     | int                | INTEGER                     |
| Interval    | datetime.timedelta | INTERVAL or DATE from epoch |
| LargeBinary | byte               | BLOB or BYTEA               |
| Numeric     | decimal.Decimal    | NUMERIC or DECIMAL          |
| Unicode     | unicode            | UNICODE or VARCHAR          |
| Text        | str                | CLOB or TEXT                |

## Metadata

Каталог объектов Table с опциональной информацией о engine и соединении.

```
from sqlalchemy import MetaData
metadata = MetaData()
```

## Создание таблицы

```
metadata.create_all(engine)
```

Метод ...create\_all не пересоздает таблицы.

Объект таблицы состоит из названия, переменной метаданных и столбцов.

```
from sqlalchemy import Table, Column, Integer, Numeric, String, ForeignKey
from datetime import datetime
from sqlalchemy import DateTime

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2))
)

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.now)
)
```

## Класс Column

- название столбца
  - тип данных
    - в String обязательно указывать длину
    - Numeric(11,2) означает 11
  - доп. параметры

```
primary_key=True
index=True
nullable=False
unique=True
default=datetime.now
onupdate=datetime.now
```

## Ключи, ограничения и индексы

Могут быть определены в конструкторе столбца (`primary_key=True`) или позже в конструкторе таблицы.

```
from sqlalchemy import PrimaryKeyConstraint, UniqueConstraint, CheckConstraint

users = Table(...
    PrimaryKeyConstraint('user_id', name='user_pk'),
    UniqueConstraint('username', name='uix_username'),
    CheckConstraint('unit_cost >= 0.00', name='unit_cost_positive'),
    ...)
```

Множественные ключи перечисляются через запятую.

```
from sqlalchemy import Index

Index('ix_cookies_cookie_name', 'cookie_name')
Index('ix_test', mytable.c.cookie_sku, mytable.c.cookie_name)
```

## Внешние связи

```
from sqlalchemy import ForeignKey

orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
```

```
)
```

Связь для поля order\_id:

```
Column('user_id', ForeignKey('users.user_id'))  
#При желании - ограничение  
ForeignKeyConstraint(['order_id'], ['orders.order_id'])
```

## Добавление данных

```
from sqlalchemy import insert  
перем = таблица.insert().values()
```

Лучше (?) вариант

```
from sqlalchemy import insert  
перем = insert(таблица).values()
```

Строковое представление запроса

```
str(перем)
```

Компиляция запроса

```
перем.compile()
```

```
перем.compile().params
```

## Примеры

```
with engine.connect() as connection:  
    metadata = ...  
    cookies = Table...  
    ins = insert(cookies).values(...)  
    res = connection.execute(ins)  
    #res.inserted_primary_key - какой в будущем будет ключ (сейчас фактически в БД нет данных)  
    connection.commit()
```

```
ins = cookies.insert()  
inventory_list = [  
    {
```

```

        ['cookie_name': 'peanut butter',
         'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html',
        ],
    )
    ['cookie_name': 'oatmeal raisin',
     'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html',
    ]
]

result = connection.execute(ins, inventory_list)

```

```

from sqlalchemy import create_engine
from sqlalchemy import MetaData
from sqlalchemy import Table, Column, Integer, Numeric, String, ForeignKey

metadata = MetaData()
cookies = Table('cookies', metadata,
                [Column('cookie_id', Integer(), primary_key=True),
                 Column('cookie_name', String(50), index=True),
                 Column('cookie_recipe_url', String(255))
                ])
engine = create_engine('sqlite:///memory:')
connection = engine.connect()
metadata.create_all(engine)

from sqlalchemy import insert

ins = cookies.insert().values(
    [cookie_name="chocolate chip",
     cookie_recipe_url="http://some.aweso.me/cookie/recipe.html"
    ])
print(str(ins))

```

## Получение данных

```

from sqlalchemy.sql import select
s = select(cookies)
rp = connection.execute(s)

```

Список столбцов

```
rp.keys()
```

## Получение результата

```
results = rp.fetchall()
```

|                         |  |
|-------------------------|--|
| <code>fetchall()</code> | Все записи   |
| <code>first()</code>    | Возвращает одну запись если она единственная и закрывает соединение          |
| <code>fetchone()</code> | Возвращает одну запись и оставляет соединения. Аккуратно!                    |
| <code>scalar()</code>   | Возвращает одно значение если результат запроса одна строка с одним столбцом |

## Доступ возможен по:

|   |                                |
|---|--------------------------------|
| <code>first_row = results[0]</code>           | по индексу результата          |
| <code>first_row[1]</code>                     | по номеру столбца в результате |
| <code>first_row.cookie_name</code>            | по имени столбца               |
| <code>first_row[cookies.c.cookie_name]</code> | через объект таблицы           |

## Сортировка

```
s = select(cookies.c.cookie_name, cookies.c.quantity)
s = s.order_by(cookies.c.quantity)
s = s.order_by(desc(cookies.c.quantity))
```

## Ограничения количества

```
s = s.limit(2)
```

## Встроенные функции

```
from sqlalchemy.sql import func
s = select([func.sum(cookies.c.quantity)])
rp = connection.execute(s)
print(rp.scalar())
```

## Фильтрация

```
s = select([cookies]).where(cookies.c.cookie_name == 'chocolate chip')
```



| Оператор               | Описание                                   |
|------------------------|--|
| ==                     | Точное равенство                           |
| like('%chocolate%')    | Вхождение элемента (регистрозависимый)     |
| ilike(string)          | Вхождение элемента                         |
| between(cleft, cright) | Элемент между значениями                   |
| concat(column_two)     | Объединение столбцов                       |
| distinct()             | Только уникальные значения столбца         |
| in_(list)              | Значения столбца в списке                  |
| is_(None)              | Значение в столбце None                    |
| contains(string)       | Содержит в себе строку (регистрозависимый) |
| endswith(string)       | Заканчивается строкой (регистрозависимый)  |
| startswith(string)     | Начинается строкой (регистрозависимый)     |
| notin_()               | Отрицание                                  |
| isnot()                | Исключение                                 |

Внутри where можно использовать and\_, or\_, not\_

```
from sqlalchemy import and_, or_, not_
s = select([cookies]).where(
    and_(
        cookies.c.quantity > 23,
        cookies.c.unit_cost < 0.40
    )
)
```

Join

```
columns = [orders.c.order_id, users.c.username, users.c.phone,
            cookies.c.cookie_name, line_items.c.quantity, line_items.c.extended_cost]
cookiemon_orders = select(*columns)
cookiemon_orders =
cookiemon_orders.select_from(orders.join(users).join(line_items).join(cookies)).where(users.c.username ==
```

```
'cookiemon')
result = connection.execute(cookiemon_orders).fetchall()
for row in result:
    print(row)
```

Для outerjoin: join -> outerjoin

Алиасы

```
manager = employee_table.alias('mgr')
```

Grouping:

```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username)
```

Обновление данных:

```
from sqlalchemy import update
u = update(cookies).where(cookies.c.cookie_name == "chocolate chip")
u = u.values(quantity=(cookies.c.quantity + 120))
result = connection.execute(u)
```

Удаление данных:

```
from sqlalchemy import delete
u = delete(cookies).where(cookies.c.cookie_name == "dark chocolate chip")
result = connection.execute(u)
```

Сырые запросы (raw)

```
result = connection.execute("select * from orders").fetchall()
```

## Обработка исключений

AttributeError - ошибка набора данных

IntegrityError - ошибка ограничений

Стандартная обработка подходит если выполняется один независимый запрос.

```
try:
    result = connection.execute(ins)
except IntegrityError as error:
    print(error.orig.message, error.params)
```

В случае нескольких взаимозависимых запросов необходимо использовать транзакции.

```
transaction = connection.begin()
try:
    ...
    transaction.commit()
except IntegrityError as error:
    transaction.rollback()
```

Пример:

```
transaction = connection.begin()
cookies_to_ship = connection.execute(s).fetchall()
try:
    for cookie in cookies_to_ship:
        u = update(cookies).where(cookies.c.cookie_id == cookie.cookie_id)
        u = u.values(quantity = cookies.c.quantity-cookie.quantity)
        result = connection.execute(u)
        u = update(orders).where(orders.c.order_id == order_id)
        u = u.values(shipped=True)
        result = connection.execute(u)
        print("Shipped order ID: {}".format(order_id))
    transaction.commit()
except IntegrityError as error:
    transaction.rollback()
```

# Пример проекта

## Структура проекта

| Директория / файл                 | Описание  |
|-----------------------------------|---|
| alembic/                          | Настройки alembic   |
| conf/                             | Настройки окружений.  |
| conf/settings                     | Файлы основных настроек.  |
| db/                               | Описание структуры базы данных.<br>initializer.py - Инициализация базы данных, метаданных |
| db/tablesdefinition               | Файлы описания структур таблиц и методов взаимодействия с данными.                        |
| docker/                           | Настройки контейнера  |
| docker/data                       | Данные БД   |
| docker/docker-entrypoint-initdb.d | Скрипты инициализации БД<br>main.sql - Файл скрипта инициализации                         |
| docker/docker-compose.yml         | Compose файл  |
| src/                              | Дополнительные модули   |
| main.py                           | Точка входа   |
| error.log                         | Файл лога.  |

## Предварительная настройка

Для работы примера необходимо установить docker.

Клонировать проекта с репозитория

```
git clone https://gitverse.ru/bobrobot/alembictemplate.git
```

Перейти в директорию проекта, создать виртуальное окружение и активировать

```
cd alembictemplate
python3 -m venv env
source env/bin/activate
```

Установить дополнительные модули

```
pip install -r requirements.txt
```

Перейти в директорию docker и в файле docker-compose.yml настроить пути, имя БД, логин и пароль к новой базе данных.

```
services:
  postgres:
    image: postgres:latest
    environment:
      POSTGRES_DB: "learnsqlalchemy"
      POSTGRES_USER: "learner"
      POSTGRES_PASSWORD: "StrongPassword123"
      PGDATA: "/home/sergey/projects/alembictemplate/docker/data/pgdata"
    volumes:
      - ../docker-entrypoint-initdb.d
      - mydata:/home/sergey/projects/alembictemplate/docker/data
    ports:
      - "5430:5432"
  volumes:
    mydata:
```

В файле docker-entrypoint-initdb.d/main.sql изменить имя БД, логин и пароль.

```
CREATE DATABASE learnsqlalchemy;
CREATE USER learner WITH PASSWORD 'StrongPassword123';
ALTER ROLE learner WITH PASSWORD 'StrongPassword123';
GRANT ALL PRIVILEGES ON DATABASE learnsqlalchemy to learner;
```

В директории docker запустить контейнер БД в фоновом режиме.

```
docker compose up -d
```

Для остановки контейнера:

```
docker compose stop
```

Сейчас, запустив контейнер, при помощи консольного клиента psql можно проверить соединение с базой данных для пользователя.

```
psql -d learnsqlalchemy -U learner -W -h 127.0.0.1 -p 5430
```

Для работы с настройками в формате json используется библиотека src/libsettings.py

[Описание библиотеки](#) Создать папку src, скопировать из проекта библиотеку libsettings.py

## Настройки системы

Файлы основных настроек расположены в conf/settings/ Файл base.py

```
'''Loading settings to project'''
import os
from src.libsettings import Jsettings

settingspath = os.path.join('conf', 'settings', 'settings.json')
schemapath = os.path.join('conf', 'settings', 'schema.json')
# dev settings
mysettings = Jsettings(settingsfname= settingspath,
                        schemafname=schemapath)
mysettings.load_settings()
```

Файл schema.json

```
{
  "type": "object",
  "properties": {
    "db_username": {"type": "string"},
    "db_password": {"type": "string"},
    "db_host": {"type": "string"},
    "db_port": {"type": "string"},
    "db_name": {"type": "string"}
  },
  "required": ["db_username", "db_password", "db_host",
               "db_port", "db_name"]
}
```

Файл settings.json

```
{
  "db_username": "learner",
  "db_password": "StrongPassword123",
  "db_host": "127.0.0.1",
  "db_port": "5430",
  "db_name": "learnsqllalchemy"
```

```
}
```

Файл `base.py` проверяет схему и создает объект настроек `mysettings` из файла `settings.json`. Для получения объекта настроек нужно импортировать объект `mysettings` в нужном модуле. В данный момент присутствуют настройки базы данных с префиксом `db_*`

Если такая усложненная система управления настройками покажется излишней, возможно использовать экспорт настроек напрямую из файла.

### Инициализация базы данных

Создаем папку `db`, в ней создаем файл `initializer.py`

```
"""Db classes and initialization"""
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base
from conf.settings.base import mysettings

#load engine settings
engine = create_engine(
    "postgresql+psycopg2://{db_username}:{db_password}@{db_host}:{db_port}/{db_name}".format(
        db_username=mysettings.db_username,
        db_password=mysettings.db_password,
        db_host=mysettings.db_host,
        db_port=mysettings.db_port,
        db_name=mysettings.db_name
    ),
    echo=True)
Base = declarative_base()
```

Здесь только создается `engine` для подключения к БД и класс `Base`. При настройке структуры таблиц данный файл будет обновлен, сейчас нужен только класс `Base`.

### Настройка системы версионирования базы данных alembic

Инициализируем `alembic` в корне проекта.

```
alembic init alembic
```

В корне проекта будет создан файл `alembic.ini`, будет создана папка `alembic` с файлами инициализации. В большинстве инструкций параметры подключения задаются в файле `alembic.ini` однако, для доступа к настройкам из единой точки будет использоваться способ установки параметров в файле `env.py`. Поэтому в файле `alembic.ini` переменная `sqlalchemy.url`

должна быть закомментирована. Часть файла alembic.ini

```
#sqlalchemy.url = driver://user:pass@localhost/dbname
```

В файле env.py

- импортируем путь

```
import os
import sys

sys.path.append(os.getcwd())
```

Импортируем настройки, создаем строку соединения и создаем закомментированный ранее в файле alembic.ini параметр sqlalchemy.url

```
from conf.settings.base import mysettings

connstring =
"postgresql+psycopg2://{db_username}:{db_password}@{db_host}:{db_port}/{db_name}".format(
    db_username=mysettings.db_username,
    db_password=mysettings.db_password,
    db_host=mysettings.db_host,
    db_port=mysettings.db_port,
    db_name=mysettings.db_name
)
config.set_main_option(name="sqlalchemy.url", value=connstring)
```

Импортируем db.initializer и создаем метаданные

```
import db.initializer

target_metadata = db.initializer.Base.metadata
```

Остальные параметры оставляем неизменными. Результирующий файл настроек окружения alembic env.py:

```
from logging.config import fileConfig
import os
import sys

from sqlalchemy import engine_from_config
```



```

from sqlalchemy import pool

from alembic import context

from conf.settings.base import mysettings

sys.path.append(os.getcwd())

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
if config.config_file_name is not None:
    fileConfig(config.config_file_name)

connstring =
"postgresql+psycopg2://{db_username}:{db_password}@{db_host}:{db_port}/{db_name}".format(
    db_username=mysettings.db_username,
    db_password=mysettings.db_password,
    db_host=mysettings.db_host,
    db_port=mysettings.db_port,
    db_name=mysettings.db_name
)
config.set_main_option(name="sqlalchemy.url", value=connstring)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
import db.initializer

target_metadata = db.initializer.Base.metadata

# other values from the config, defined by the needs of env.py,
# can be acquired:
# my_important_option = config.get_main_option("my_important_option")
# ... etc.

```

```
def run_migrations_offline() -> None:
```

```
    """Run migrations in 'offline' mode.
```

This configures the context with just a URL  
and not an Engine, though an Engine is acceptable  
here as well. By skipping the Engine creation  
we don't even need a DBAPI to be available.

Calls to context.execute() here emit the given string to the  
script output.

```
    """
```

```
    url = config.get_main_option("sqlalchemy.url")
    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )
```

```
    with context.begin_transaction():
        context.run_migrations()
```

```
def run_migrations_online() -> None:
```

```
    """Run migrations in 'online' mode.
```

In this scenario we need to create an Engine  
and associate a connection with the context.

```
    """
```

```
    connectable = engine_from_config(
        config.get_section(config.config_ini_section, {}),
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )
```

```
    with connectable.connect() as connection:
        context.configure(
```

```

        connection=connection, target_metadata=target_metadata
    )

    with context.begin_transaction():
        context.run_migrations()

if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()

```

## Обновление конфигурации таблиц

Для проверки создаем первую пустую миграцию. После ее выполнения создастся таблица `alembic_version`

```
alembic revision -m "Empty Init"
```

В данный момент фактического соединения с БД не было. В папке `versions` сформируется файл вида `<id>_empty_init.py`

После выполнения команды

```
alembic upgrade head
```

в таблице `alembic_version` появится одна запись - идентификатор текущей версии базы данных.

Сейчас в папке `db` создаем папку `tablesdefinition`. В ней будем хранить файлы описаний таблиц и методы для работы с таблицами. Создадим файл `userprofile.py`

```

"""Definition tables of userprofile"""
import logging
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import sessionmaker
import db.initializer
from db.initializer import engine

def create_userprofile_class(Curbase):
    class Userprofile(Curbase):
        """Class Userprofile definition"""

```

```

__tablename__ = 'userprofile'

user_id = Column(Integer(), primary_key=True)
username = Column(String(15), nullable=False, unique=True)
password = Column(String(255), nullable=False)
email = Column(String(255))
name = Column(String(100))
second_name = Column(String(100))
photo = Column(String(255))
balance = Column(Integer())

return Userprofile

def create_one_userprofile(username, password, email="", name="",
                           second_name="", photo="", balance=0):
    """ Create one userprofile """
    try:
        with engine.connect():
            Session = sessionmaker(bind=engine)
            with Session() as sess:
                upelem = db.initializer.userprofile(username=username, password=password,
                                                    email=email, name=name, second_name=second_name,
                                                    photo=photo, balance=balance)

                sess.add(upelem)
                sess.commit()
    except Exception as e:
        logging.error(e)

```

И в файле `initializer.py` после инициализации переменной `Base` добавим раздел инициализации описания таблицы

```

#===== Creation classes definitions
=====

# === Import Userprofile class ===
from db.tablesdefinition.userprofile import create_userprofile_class
userprofile = create_userprofile_class(Base)

#
=====
=====

```

Теперь после выполнения команды

```
alembic revision --autogenerate -m "Added userprofile model"
```

будет автоматически сгенерирован файл миграции, и после

```
alembic upgrade head
```

создастся таблица userprofile.

P.s. В точке входа необходимо полностью импортировать initializer иначе будет ошибка, пример:

```
'''Main learning module'''
import logging

from db import initializer
from db.tablesdefinition.userprofile import create_one_userprofile

logging.basicConfig(level=logging.INFO,
                    filename='error.log',
                    format="%(levelname)s %(message)s")

if __name__ == '__main__':
    create_one_userprofile(username = 'first6',
                          password = 'first',
                          balance = 1)
```

