Для возможности работы с БД в питоне надо импортировать, в нашем случае, модуль **psycopg2.**

Для того, чтобы сделать подключение, надо вызвать метод connect, этот метод возвращает результат, поэтому вызов надо присваивать переменной, указав название базы данных, имя пользователя и пароль. Имя пользователя часто или иногда бывает разным, а в нашем случае оно идет по умолчанию именно таким. Там есть еще хост и порт, но пока они не нужны нам.

**conn = psycopg2.connect(dbname='testdb', user='postgres', password='A8nDIVDh23@#')**

есть еще экземпляр класса Connection, он инкапсулирует сессию подключения к БД и создает новый объект – курсор. Курсор помогает исполнять команды и запросы на стороне БД. Для получения курсора нам надо вызвать метод курсор.

**cur = conn.cursor()** # возвращает инстанцию типа курсор

курсор помогает отправлять команды, используя методы execute и executemany и забирать данные из БД, используя такие методы как fetchone, fetchmany, fetchall.

Как мы создаем команды: **cur.execute(‘DROP TABLE IF EXISTS superheroes’) –** то есть мы просто берем наше соединение, вызываем на нем метод, а затем указываем просто строку, как мы ее указали бы в SQL. Однако эти команды не будут исполнены до тех пор, пока не будет вызван другой метод также на подключении: **conn.commit()**. Данный метод уже все запускает и исполняет на стороне БД.

cur.execute("DROP TABLE IF EXISTS superheroes")  
cur.execute('DROP TABLE IF EXISTS traffic\_light')  
  
  
cur.execute("CREATE TABLE superheroes (hero\_id SERIAL PRIMARY KEY, hero\_name varchar, strength int); ")  
  
cur.execute('insert into superheroes (hero\_name, strength) values (%s, %s)', ('superman', 100))  
cur.execute('insert into superheroes (hero\_name, strength) values (%s, %s)', ('batman', 90))  
  
conn.commit()

import psycopg2  
  
# ф-я коннект создает соединение с БД и возвращает результат  
conn = psycopg2.connect(dbname='testdb', user='postgres', password='A8nDIVDh23@#')  
  
cur = conn.cursor() # возвращает инстанцию типа курсор  
  
cur.execute("DROP TABLE IF EXISTS superheroes")  
cur.execute('DROP TABLE IF EXISTS traffic\_light')  
  
  
cur.execute("CREATE TABLE superheroes (hero\_id SERIAL PRIMARY KEY, hero\_name varchar, strength int); ")  
# плейсхолдеры надо ставить обязательно, если по одному мы вставляем, то просто %s, если больше, то иначе  
cur.execute('insert into superheroes (hero\_name, strength) values (%s, %s)', ('superman', 100))  
cur.execute('insert into superheroes (hero\_name, strength) values (%s, %s)', ('batman', 90))  
# выводятся они то же с плейсхолдерами, но немного иначе выглядят, и еще имена их мб отличными тех, что в 1 строке  
# вводимые данные должны быть в виде словаря  
cur.execute('''  
 insert into superheroes (hero\_name, strength)  
 values (%(name)s, %(strength)s);  
 ''', {'name': 'Riddler', 'strength': 80})  
  
conn.commit()  
# перенос строк удобно совершать при помощи тройных кавычек

ввод данных через input()

for x in range(3):  
 cur.execute('insert into superheroes (hero\_name, strength) values (%s, %s)', (input("hero's name: "), int(input("hero's strength: "))))

обновление данных

cur.execute('update superheroes set strength = 100 where hero\_id = 6')

**или так**

cur.execute('’’update superheroes

set strength = %s

where hero\_id = %s

'’’, (90, 6)

)

**как нельзя вставлять данные**

сur.execute(‘insert into table (smth) values (‘%s or %d’), (10)’), короче. Нельзя использовать херь для форматирования строк, если передаются инты, обязательно ставить запятую, где 10, если передаются данные в кортеже.

Светофор:

cur.execute('CREATE TABLE traffic\_light (light\_id serial PRIMARY KEY, light text);')  
cur.execute('insert into traffic\_light(light) values(%s)', ("red",))  
cur.execute('insert into traffic\_light(light) values(%s)', ("yellow",))  
cur.execute('insert into traffic\_light(light) values(%s)', ('green', ))  
conn.commit()

**чтение данных**

для чтения данных используем fetchone, fetchmany, fetchall, если мы сначала взяли первую строку через fetchone, а потом решили взять оставшиеся скопом через fetchall, то вывод начнется уже со второго элемента, потому что курсор однонаправлен, у него сохраняется инфа, какие поля уже были взяты ранее. Но индексацию можно скинуть. Затем, если мы берем все данные, то посмотреть их можно через цикл. Взятие данных идет через создание переменной.

cur.execute(‘select \* from superheroes’)

one\_line = cur.fetchone()

print(one\_line)

full\_table = cur.fetchall()

for x in full\_table:

print(x)

кроме того можно выводить данные по индексу

full\_table[0][0]

**закрытие соединения**

cur.close()

conn.close()

после того, как мы поработали с БД, лучше бы закрыть соединение, чтобы оно не тратило лишних ресурсов. Кроме того, всю эпопею лучше выполнять маленькими кусочками, коммитить вовремя. Всегда может произойти сбой по какой бы то ни было причине, и если коммит не был вызван, будет неявно вызван метод роллбэк, который удалит все изменения, и данные не сохранятся. Также есть такое понятие как транзакция, она открывается при первой команде вроде как манипуляции с БД, на одном коннекте может быть не один курсор, и все они будут работать в рамках одной транзакции.

Модуль псайкопэгэ поддерживает работу с менеджером контекстов with.

with psycopg2.connect(dbname='testdb', user='postgres', password='A8nDIVDh23') as conn:  
 with conn.cursor(cursor\_factory=RealDictCursor) as curs:  
 execute\_values(curs, 'insert into traffic\_light (light) values %s', [("blue",), ("orange",)])  
  
 curs.execute("SELECT \* FROM traffic\_light")  
 records = curs.fetchall()  
 print(records)  
 print(records[0]['light'])

[RealDictRow([('light\_id', 1), ('light', 'red')]), RealDictRow([('light\_id', 2), ('light', 'yellow')]), RealDictRow([('light\_id', 3), ('light', 'green')]), RealDictRow([('light\_id', 4), ('light', 'blue')]), RealDictRow([('light\_id', 5), ('light', 'orange')])]

Red

что тут происходит – подключение к БД через МК, присваиваем подключению имя, затем таким же макаром создаем курсор, но в методе курсор мы указали cursor\_factory=RealDictCursor, при выводе всех строк из таблицы этот cursor\_factory выведет список экземпляров класса RealDictRow, ну это как видно из вывода. А другой вывод print(records[0]['light']) – тут уже обращение к конкретному столбику ЛАЙТ по индексу, тут обратная упорядоченность – сначала номер строки, потом название столбика.

По завершении блока WITH транзакция неявно и автоматический закрывается, то есть коммитится при условии, что не было никаких ошибок и прочей фигни. Курсор при этом будет закрыт, занятая память будет освобождена. Если же были ошибки, то неявно уже бужет вызван метод роллбэк, откат. При этом при покидании блока WITH само соединение не закрывается, можно создать еще один курсор и уже заниматься с ним, а потом еще и еще. Поэтому по завершении работы нужно явно вызвать conn.close(). Поэтому основную работу можно выполнять в блоке TRY, а conn.close() делать в блоке FINALLY

**Этот код позволяет достать данные из файла и записать их в БД**

with open('C:\\test\\test.txt', mode='r') as file:  
 text = file.read()  
  
 print(text.split(' '))  
  
import psycopg2  
conn = psycopg2.connect(dbname='testdb', user='postgres', password='A8nDIVDh23')  
cur = conn.cursor()  
  
cur.execute('insert into person (first\_name, surname, gender, birth\_date) values (%s, %s, %s, %s)', text.split(' '))  
conn.commit()

а вот этот код позволяет делать то же самое, но уже из большего количества строк

file = open('C:\\test\\test2.txt', mode='r')  
  
import psycopg2  
conn = psycopg2.connect(dbname='testdb', user='postgres', password='A8nDIVDh23')  
cur = conn.cursor()  
  
for x in range(2):  
 text = file.readline()  
 cur.execute('insert into person (first\_name, surname, gender, birth\_date) values (%s, %s, %s, %s)', text.split())  
 conn.commit()

тут я проверяю два варианта прохода по файлу с данными, чтобы их загрузить. Первый будет таким, что я открываю файл, считаю количество строчек, потом возвращаюсь на начало файла и вношу их уже в таблицу

from src.deco import measure\_time  
  
file = open('C:\\test\\words.txt', mode='r')  
  
import psycopg2  
conn = psycopg2.connect(dbname='testdb', user='postgres', password='A8nDIVDh23')  
cur = conn.cursor()  
  
@measure\_time  
def testing(file):  
 for i in range(len(file.readlines())):  
 file.seek(0)  
 text = file.readline()  
 cur.execute('insert into words (word) values (%s)', text.split())  
 conn.commit()  
 if not text:  
 print('data input is done')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 testing(file)

а второй вариант не получился, потому что выходит ошибка

**SQLALCHEMY**

Второй вариант работы с БД доступен через СКЛАлкеми. Для того, чтобы создать соединение, предварительно надо создать движок.

import sqlalchemy  
  
engine = sqlalchemy.create\_engine('postgresql+psycopg2://postgres:A8nDIVDh23@localhost:5432/testdb')  
  
connection = engine.connect()  
  
#connection.execute('insert into person (first\_name, surname, gender, birth\_date) values (%s, %s, %s, %s) ')  
  
result = connection.execute("select \* from person")  
print(result)  
for el in result:  
 print('Person', el['first\_name'], el['birth\_date'])

result.close()

Непосредственно создание движка, создание соединения, вывод данных из таблицы, причем без ограничения по количеству строк.

Транзакция – это вроде как раз и есть тема для ввода данных.

trans = connection.begin()  
  
try:  
 connection.execute("insert into person (first\_name, surname, gender, birth\_date) values ('Alena', 'Kirillova', 'female', '1994-05-19')")  
 trans.commit()  
finally:  
 trans.rollback()

Здесь мы создали транзакцию через бегин, вводим данные через коннекшн, а закрываем или откатываем транзакцию и изменения опять через транз. Но лучше управлять через менеджер контекстов, вроде как.

with connection.begin() as trans:  
 connection.execute(  
 "insert into person (first\_name, surname, gender, birth\_date) values ('Marusya', 'Kirillova', 'female', '2017-08-08')")  
  
result = connection.execute("select \* from person")  
print(result)  
for el in result:  
 print('Person', el['first\_name'], el['birth\_date'])

Тут мы просто создали эту транзакцию, и работаем с коннектами. А надо будет закрыть. Обратимся к транз напрямую.

Одна из основных важных тем Алхимии – это использование ее как орм, задачи ОРМ - Необходимо обеспечить работу с данными в терминах классов, а не таблиц данных, и, напротив, преобразовать термины и данные классов в данные, пригодные для хранения в СУБД. Необходимо также обеспечить интерфейс для CRUD-операций над данными. В общем, необходимо избавиться от необходимости писать SQL-код для взаимодействия в СУБД.

Для того, чтобы использовать алхимию как ОРМ, нам необходимо описать таблицы, с которыми мы будем иметь дело, в виде классов и декорировать эти классы с помощью специальных типов из Алхимии. И классы будут мапиться на таблички, которые находятся в БД. – мапиться – это походу типа создание классов на основе базового класса.

(Тоже самое. От слова map. Смысл — отображение, соответствие. Как правило отображение какого то ключа на какое то значение. В случае с файлами это будет отображение хендла файла на область памяти со значением. Можно делать соответствие между идентификатором и какой то сущностью. Можно обеспечивать соответствие между русским и английским словом, между неопределенной формой и набором падежей, и т.д. Можно мапить сущности на реляционные таблицы — это тоже маппинг.) <https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D0%B0_%D0%B2_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C>

Для этого в алхимии имеется подсистема, которая называется declarative, с ее помощью мы декларируем классы, которые будут мапиться на таблицы. В этой подсистеме есть базовый класс, который наследуют все остальные классы, которые мапятся на таблицы. Этот базовый класс содержит карту классов и взаимоотношений между ними, он называется declarative base class. Создается как правило один экземпляр.

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()  
  
class Person(Base): # это название класса, а не таблицы  
 \_\_tablename\_\_ = 'person' # то, как он прописан именно в БД  
  
 # здесь мы фактически описываем таблицу в терминах Алхимии  
 person\_id = Column(Integer, primary\_key=True)  
 first\_name = Column(Text)  
 surname = Column(Text)  
 gender = Column(Text)  
 birth\_date = Column(Date)

Вот так выглядит таблица, которая у меня в есть в простгре, мы сами явно описываем наименования колонок, их тип данных и всего остального. Другие ОРМ-ки по различным соглашениям и т.д. позволяют не вводить такие данные, мы указываем минимум, типа названия или еще чего, хз, а ОРМ сама догадывается, где что надо поставить и все дела. Здесь пропагандируется подход, когда мы видим, как все это там работает и тд. Это и плюс, и минус. С одной стороны мы все это видим и контролируем, с другой стороны мы могли бы это и не писать, сэкономили бы время. Маппинг производился бы автоматически.

Кстати, данные из БД поступают в виде кортежей.

Прежде чем мы начнем давать команды, нам необходимо вызвать метод create all на метаданных. Есть такая коллекция как метадата, метаданные. Это часть подсистемы declarative, это такой реестр, который помогает подключаться к табличкам и взаимодействовать с ними.

Base.metadate.create\_all(engine) # движок мы создавали ранее, и теперь мы готовы работать с БД.

Для того, чтобы взаимодействовать с БД, нам необходимо порождать сессии, а сессии мы порождаем с помощью специальной фабрики. Для того, чтобы создать фабрику, мы вызываем метод sessionmaker(bind=engine) – его надо импортировать.

Session = sessionmaker(bind=engine)

Потом мы инстанциируем сессию

session = Session()

Получилась сессия, которая ассоциирована с движком, которые мы создали заранее и ранее, но на данный момент никаких соединений еще не установлено. Когда мы начинаем использовать сессию (?), коннекшн берется из пула коннекшнов, и псайкопг и алхимия с ним работают. Сам пул нужен для того, чтобы эффективно работать с нагрузкой для подключения к БД. Сами коннекшны всегда висят открытыми, они не закрываются. Если я закончил с ним работу, он возвращается в пул, а оттуда его другой может забрать.

А вот таким способом уже происходит добавление данных в БД.

person1 = Person(person\_id=16, first\_name='Kesha', surname='Ololoev', gender='male', birth\_date='2020-09-14')  
session.add(person1)

Мы просто создаем экземпляр класса, и передаем туда данные, указывая, к чему они относятся. Все вместе выглядит вот так:

Base = declarative\_base()  
  
class Person(Base): # это название класса, а не таблицы  
 \_\_tablename\_\_ = 'person' # то, как он прописан именно в БД  
  
 # здесь мы фактически описываем таблицу в терминах Алхимии  
 person\_id = Column(Integer, primary\_key=True)  
 first\_name = Column(Text)  
 surname = Column(Text)  
 gender = Column(Text)  
 birth\_date = Column(Date)  
  
  
Base.metadate.create\_all(engine) # движок мы создавали ранее, и теперь мы готовы работать с БД.  
  
Session = sessionmaker(bind=engine)  
session = Session()  
  
person1 = Person(person\_id=16, first\_name='Kesha', surname='Ololoev', gender='male', birth\_date='2020-09-14')  
session.add(person1)

session.commit()

Нужно помнить, что объект сессии session – это продолжающаяся транзакция, которая накапливает в себе изменения. Эти изменения не происходят до тех пор, пока не будет применен метод commit(). Есть такая тема, как flush(), он передает в БД инфу о том, что там что-то планирует измениться, но изменения не происходят. Когда мы вызываем коммит, флаш вызывается неявно – имплицитно.

Вывод данных выглядит достаточно декларативно, понятно.

for item in session.query(Person).order\_by(Person.birth\_date):  
 print(item.first\_name, ' ', item.birth\_date)

или фильтр

for item in session.query(Person).filter(Person.birth\_date > '2000-01-01'):  
 print(item.first\_name, ' ', item.birth\_date)

при этом тут не «старше» типа возраста, а более ранняя дата наоборот.

Алхимия – это крупная история часов на 10 видео, а у нас короткий ролик. Для разбора надо или пройти курс по алхимии, или что.

И для обоих пакетов актуально изучать из документацию.

**Еще вариант работы с БД**

Ранее написанные примеры работы с БД не позволяли работать с БД как с объектом класса питон. Нижеследующий пример работы это продемонстрирует. Этот модуль называется PeeWee, но есть еще асинхронный пиви. Короче, надо изучить этот, а потом перейти к асинхронному парсингу.

import csv  
from peewee import \*  
  
# устанавливает связь м/у ОРМ (пиви) м БД  
# класс в переменной db = он лежит в пиви  
db = PostgresqlDatabase(database='testdb', user='postgres', password='A8nDIVDh23', host='localhost')  
  
  
# сейчас создадим класс, который будет представлять собой таблицу из БД с данными  
# Model also was imported from PeeWee  
class Coin(Model):  
 coin\_name = CharField()  
 coin\_symbol = CharField()  
 coin\_link = TextField()  
 coin\_price = CharField()  
  
 # связываем класс с БД  
 class Meta():  
 database = db  
  
  
def main():  
  
 db.connect()  
 # принимает список таблиц, которые надо создать в БД, у нас только одна таблица. Ее поля описаны в классе выше  
 db.create\_tables([Coin])  
  
 with open('cmc4.csv', 'r', encoding='utf-8') as file:  
 order = ['coin\_name', 'coin\_symbol', 'coin\_link', 'coin\_price']  
 reader = csv.DictReader(file, fieldnames=order)  
 coins = list(reader)  
  
 # есть несколько вариантов записи данных,самый плохой показывать не буду, начну с более менее норм  
 # (\*\*row) подсвечен, но все сработало, данные записались  
 with db.atomic():  
 #for row in coins:  
 # Coin.create(\*\*row)  
  
 # другой вариант еще более быстрый, там также будет через with db.atomic():  
 # тут данные подаются по кускам, что позволяет сократить количество циклов  
 for index in range(0, len(coins), 10):  
 Coin.insert\_many(coins[index:index+10]).execute()  
  
 # надо еще изучить,как делать дамп БД из иде,  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

порядок действий:

1. Импортируем необходимые модули
2. Создаем подключение к БД, именуем его.
3. Создаем классы, наследующиеся от Model, которые будут представлять собой таблицы. Сколько таблиц, столько и классов. Потом мы будем передавать их списком.
4. В создаваемых классах нужно указать поля будущей таблицы и тип данных, первичный ключ вроде как сам собой создается. В каждом классе создается свой класс Meta, где прописывается database = db, где db – это экземпляр из 2 пункта.
5. Создается ф-я, которая будет исполнять код.
6. В этой ф-и мы уже подключаемся к созданному подключению из 2 пункта. Прописываем создание таблиц, передавая в него список классов.
7. После мы при помощи менеджера контекстов открываем нужный файл на чтение, создаем переменную, куда в виде списка строк передаются имена столбцов таблицы. Создаем читателя, который принимает в себя файл из М/К и порядок. Полученный результат переводим в список.
8. После создаем непосредственно запись в БД.
9. Также М\К будет, мы берем наше подключение и при помощи метода atomic() проходимся в цикле. Он выше указан. Мы там будем нарезать список элементов на куски. Для этого мы для каждого элемента из всего списка с определенным шагом запускаем ф-ю insert\_many, которая берет этот определенный шаг, этот нарезанный кусок, и исполняет его