ORM

Brandstätter, Forsthuber, Jernej

Inhalt

[Datenbank aufsetzen 3](#_Toc42338862)

[Python Klassen anlegen 3](#_Toc42338863)

[Datenbankverbindung 3](#_Toc42338864)

# Auswahl der Daten

Auswahl:

Link: <https://www.kaggle.com/kyanyoga/sample-sales-data#sales_data_sample.csv>

Es handelt sich beim Datenset um ein künstliches Verkaufsdatenset.

Das Datenset liegt als csv vor. Die Charakter-Enkodierung ist latin-1. Das Datenset enthält 25 Spalten

und 2823 Zeilen. Details zu den Variablen werden in der nächsten Tabelle dargestellt (nächste Seite).

Offenbar gab es bei der Daten-Erstellung einen Fehler, die Variable Priceeach hatte bei 100 eine

Obergrenze. Das wurde über die offenkundige Beziehung Sales = PRICEEACH \* QUANTITYORDERED

korrigiert. Zur Richtigstellung wurde über die Berechnung Sales / QUANTITYORDERED der korrekte

PRICEEACH eingefügt.

Zusätzlich wurde ein Primary Key erzeugt, durch die Zusammenfügung von ORDERNUMBER und

ORDERLINENUMBER.

Die Datei wurde in ein Stern-Schema umgewandelt. Dabei wurden die zentralen Variablen, die die

Bestellungen (Orders) betreffen, im Zentrum (Dimension center) belassen (siehe nächste Tabelle). Um die Daten aufzuteilen, wurde in Python eine Funktion definiert, die für Unique-Entries aufsteigende

Integer-Sequenzen erzeugt, die sich auch mit einem Text kombinieren lassen.

# Datenbank aufsetzen

Die Datenbank ist, wie in Abbildung 1 zu sehen ist, aus 4 Tabellen in einem Sternschema aufgebaut. Die Tabellen werden mit den folgenden SQL Befehlen erstellt.

CREATE TABLE public.centertable (

pk character varying(8),

on\_id character varying(5),

pr\_id character varying(5),

cu\_id character varying(5),

ordernumber numeric(50,0),

quantityordered numeric(50,0),

priceeach numeric(52,2),

orderlinenumber numeric(50,0),

sales numeric(52,2),

status character varying(50),

dealsize character varying(50)

);

CREATE TABLE public.customertable (

cu\_id character varying(5),

customername character varying(50),

phone character varying(50),

addressline1 character varying(50),

addressline2 character varying(50),

city character varying(50),

"STATE" character varying(50),

postalcode character varying(50),

country character varying(50),

territory character varying(50),

contactlastname character varying(50),

contactfirstname character varying(50)

);

CREATE TABLE public.ordertimes (

on\_id character varying(5),

orderdate timestamp without time zone,

qtr\_id numeric(50,0),

month\_id numeric(50,0),

year\_id numeric(50,0)

);

CREATE TABLE public.producttable (

pr\_id character varying(5),

msrp numeric(50,0),

productline character varying(50),

productcode character varying(50)

);

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 1:Klassendiagramm

# Python Klassen anlegen

Als Object Relational Mapper (ORM) wird SQLAlchemy verwendet. SQLAlchemy fungiert als Schnittstelle zwischen der PostgreSQL Datenbank und der Python Applikation. SQLAlchemy wird mit folgendem Konsolenbefehl installiert:

Pip install sqlalchemy

Klassen müssen entsprechend den Tabellen in der Datenbank angelegt werden und die Attribute müssen definiert werden. Die Attribute in der Klasse bekommen die entsprechende Spalte in der Tabelle zugewiesen und der entsprechende Datentyp wird angegeben. Die Klassen erben von einer Base-Klasse die mit der Factory declarative\_base() erzeugt wird und müssen alle von der selben Base-Klassen Instanz erben, damit die Beziehungen beim Instanziieren schon bekannt sind.

Beziehungen müssen mit relationship() definiert werden. Wir haben, in Abbildung 1 zu sehen, ausgehend vom Centertable n:1 Beziehungen auf Ordertimes, Producttable und Customertable. Folgendermaßen werden die Klassen im Code definiert:

**from** sqlalchemy **import** Table, Column, String, Integer, ForeignKey, DateTime  
**from** sqlalchemy.ext.declarative **import** declarative\_base  
**from** sqlalchemy.orm **import** relationship  
  
Base = declarative\_base()  
  
**class** Centertable(Base):  
 \_\_tablename\_\_ = **'centertable'** pk = Column(String(8), primary\_key=**True**)  
 on\_id = Column(String(5), ForeignKey(**"ordertimes.on\_id"**))  
 pr\_id = Column(String(5), ForeignKey(**"producttable.pr\_id"**))  
 cu\_id = Column(String(5), ForeignKey(**"customertable.cu\_id"**))  
 ordernumber = Column(**'ordernumber'**, Integer)  
 quantityordered = Column(**'quantityordered'**, Integer)  
 priceeach = Column(**'priceeach'**, Integer)  
 orderlinenumber = Column(**'orderlinenumber'**, Integer)  
 sales = Column(**'sales'**, Integer)  
 status = Column(**'status'**, String(50))  
 dealsize = Column(**'dealsize'**, String(50))  
 ordertimes = relationship(**"Ordertimes"**, back\_populates=**'centertable'**)  
 producttable = relationship(**"Producttable"**, back\_populates=**'centertable'**)  
 customertable = relationship(**"Customertable"**, back\_populates=**'centertable'**)  
  
 **def** \_\_init\_\_(self, ordernumber, quantityorderd, priceeach, orderlinenumber, sales, status, dealsize):  
 self.ordernumber = ordernumber  
 self.quantityorderd = quantityorderd  
 self.priceeach = priceeach  
 self.orderlinenumber = orderlinenumber  
 self.sales = sales  
 self.status = status  
 self.dealsize = dealsize  
  
**class** Ordertimes(Base):  
 \_\_tablename\_\_ = **'ordertimes'** on\_id = Column(**'on\_id'**, String(5), primary\_key=**True**)  
 orderdate = Column(**'orderdate'**, DateTime())  
 qtr\_id = Column(**'qtr\_id'**, Integer)  
 month\_id = Column(**'month\_id'**, Integer)  
 year\_id = Column(**'year\_id'**, Integer)  
 centertable = relationship(**'Centertable'**, back\_populates=**'ordertimes'**)  
  
 **def** \_\_init\_\_(self, on\_id, orderdate, qtr\_id, month\_id, year\_id):  
 self.on\_id = on\_id  
 self.orderdate = orderdate  
 self.qtr\_id = qtr\_id  
 self.month\_id = month\_id  
 self.year\_id = year\_id  
  
**class** Producttable(Base):  
 \_\_tablename\_\_ = **'producttable'** pr\_id = Column(**'pr\_id'**, String(5), primary\_key=**True**)  
 msrp = Column(**'msrp'**, Integer)  
 productline = Column(**'productline'**, String(20))  
 productcode = Column(**'productcode'**, String(20))  
 centertable = relationship(**'Centertable'**, back\_populates=**'producttable'**)  
  
 **def** \_\_init\_\_(self,pr\_id, msrp, productline, productcode):  
 self.pr\_id = pr\_id  
 self.msrp = msrp  
 self.productline = productline  
 self.productcode = productcode  
  
**class** Customertable(Base):  
 \_\_tablename\_\_=**'customertable'** cu\_id = Column(**'cu\_id'**, String(5),primary\_key=**True**)  
 customername=Column(**'customername'**,String(50))  
 phone = Column(**'phone'**, String(20))  
 addressline1 = Column(**'addressline1'**, String(50))  
 addressline2 = Column(**'addressline2'**, String(50))  
 city = Column(**'city'**, String(50))  
 state = Column(**'STATE'**, String(20))  
 postalcode = Column(**'postalcode'**,String(20))  
 country = Column(**'country'**, String(20))  
 territory = Column(**'territory'**,String(20))  
 contactlastname = Column(**'contactlastname'**, String(20))  
 contactfirstname = Column(**'contactfirstname'**, String(20))  
 centertable = relationship(**"Centertable"**, back\_populates=**'customertable'**)  
  
 **def** \_\_init\_\_(self,customername,phone,addressline1,addressline2,city,state,postalcode,country,territory,contactlastname,contactfirstname):  
 self.customername = customername  
 self.phone = phone  
 self.addressline1 = addressline1  
 self.addressline2 = addressline2  
 self.city = city  
 self.state = state  
 self.postalcode = postalcode  
 self.country = country  
 self.territory = territory  
 self.contactlastname = contactlastname  
 self.contactfirstname = contactfirstname

# Datenbankverbindung

Um die Daten aus den Tabellen in die Objekte zu bekommen, muss eine Verbindung hergestellt werden. Die Verbindung zur Datenbank wird mit folgendem Befehl hergestellt und eine Session geöffnet. Über diese Session können Daten aus der Datenbank gelesen, geschrieben und aktualisiert werden.

engine = create\_engine(**'postgresql://postgres:Test@localhost:5432/postgres'**)  
  
*# create a configured "Session" class*Session = sessionmaker(engine)  
  
*# create a Session*session = Session()

Der String, der dem create\_engine() Befehl übergeben wird, hat folgenden Aufbau:

postgresql://<Benutzername>:<Passwort>@<Host>:<Port>/<DB-Name>

# UseCase

Als UseCase wurde ein simpler PoC implementiert. Dazu gibt es ein UI in der Konsole und eines mit einer Nutzeroberfläche, die mit Hilfe von Tkinter erstellt wurde.

In der Konsolen Applikation hat man als Benutzer die Möglichkeit Daten zu lesen, zu schreiben und zu bearbeiten. Für jede dieser Verarbeitungsmöglichkeiten wurde ein kurzes Beispiel erstellt, um zu zeigen, wie diese Möglichkeit implementiert wird.

## Lesen

Beim Lesen von Daten, wird ein Datensatz oder eine Zeile einer Tabelle in ein entsprechendes Objekt aus der Datenbank gelesen. Mit diesem Objekt können dann im Programm die Daten weiterverarbeitet werden.

Das lesen funktioniert über die Session, mit dem query() Befehl.

countries = []  
[countries.append(x[0]) **for** x **in** database.session.query(Customertable.country).distinct()]  
print(countries)

Mit diesem Befehl werden zum Beispiel alle Werte aus Spalte „country“ in der Tabelle „Customertable“ als Liste in das Objekt „countries“ geschrieben. Durch den Befehl distinct() werden nur verschiedene Werte ausgegeben und doppelte Länder nicht ausgelesen. Mit dem print() Befehl wird dann eine Liste aller Ländernamen, die in der Spalte „country“ vorkommen, ausgegeben.

## Schreiben

Beim Schreiben von Daten aus der Applikation in die Datenbank, wird ein neues Objekt im Programm angelegt und dann in die entsprechende Tabelle geschrieben. Hier würde es Sinn machen, einen Primarykey (PK) zu wählen, der vom Programm automatisch so ausgewählt wird, dass dieser auf jeden Fall eindeutig ist. Auch ForeignKeys sollten so gewählt werden, dass diese auf einen wirklich existierenden Eintrag in der entsprechenden Tabelle zeigen.

product = Producttable(pr\_id=id, msrp=msrp, productline=productline, productcode=productcode)  
database.create\_object(product)

Der Programmcode erstellt ein Objekt „Producttable“ und übergibt dazu die nötigen Parameter, die dann in dem Tabelleneintrag gespeichert werden sollen.

Mit create\_object() wird das erstellte Objekt dann in die Datenbank gespeichert. Von dort kann es mit folgendem Befehl wieder ausgelesen werden:

database.session.query(Producttable).filter(Producttable.pr\_id==id, Producttable.msrp==msrp, Producttable.productline==productline)

## Update

Beim Update wird ein Eintrag in der Datenbank in ein Objekt gespeichert. Dieses Objekt wird dann im Programm verändert und die Änderung in die Datenbank übernommen.

customer = database.session.query(Customertable).get(**'CU1'**)

newname = ui.getinput()  
customer.customername = newname  
database.session.commit()

Mit dem ersten Befehl wird der Eintrag mit dem PK „CU1“ aus der Tabelle „Customertable in das Objekt „customer“ gelesen. Das Attribut „customername“ wird mit einem, vom Benutzer eingegebenen String überschrieben. Mit dem Befehl commit() wird die Änderung in die Datenbank übernommen.

Update funktioniert nur, wenn PK eindeutig sind, weil SQLAlchemy sonst nicht weiß, welche Zeile upgedatet werden soll. Deshalb wäre es beim Schreiben von Objekten in die DB sinnvoll, den PK automatisch von Programm erstellen zu lassen, mit einer Methode, die garantiert, dass der PK eindeutig ist. Wenn man so wie wir, den Benutzer einen PK aussuchen lässt, kann man sich sicher sein, dass diese nicht lang eindeutig bleiben werden.