Peewee

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Peewee is a Python ORM (Object-Relational Mapping) library which supports SQLite, MySQL, PostgreSQL and Cockroach databases. This tutorial will help you to understand how to insert a new record, delete a record, create an index, etc., with the help of Peewee. Moreover, you will gain knowledge about connection management, text and binary fields, subqueries, filters, etc., with regards to Peewee.

Audience

This tutorial is for all those programmers who wish to learn in detail about the basics and the functions of Peewee which is an expressive object relational mapper (ORM) implemented in joining the relational data with python objects.

Prerequisites

We assume that the readers of this tutorial have basic knowledge of Python, Object Relational Mapping (ORM) and databases like SQLite, MySQL, etc.

Copyright & Disclaimer

© Copyright 2020 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

tutorialspoint
SIMPLYEASYLEARNING

1. Peewee – Overview

Peewee is a Python Object Relational Mapping (ORM) library which was developed by a U.S. based software engineer **Charles Leifer** in October 2010. Its latest version is **3.13.3.** Peewee supports SQLite, MySQL, PostgreSQL and Cockroach databases.

Object Relational Mapping is a programming technique for converting data between incompatible type systems in object-oriented programming languages.

Class as defined in an Object Oriented (OO) programming language such as Python, is considered as non-scalar. It cannot be expressed as primitive types such as integers and strings.

On the other hand, databases like Oracle, MySQL, SQLite and others can only store and manipulate scalar values such as integers and strings organised within tables.

The programmer must either convert the object values into groups of scalar data types for storage in the database or convert them back upon retrieval, or only use simple scalar values within the program.

In an ORM system, each class maps to a table in the underlying database. Instead of writing tedious database interfacing code yourself, an **ORM** takes care of these issues, while you can focus on programming the logics of the system.

Environment setup

To install latest version of Peewee as hosted on PyPI (Python Package Index), use pip installer.

```
pip3 install peewee
```

There are no other dependencies for Peewee to work. It works with SQLite without installing any other package as sqlite3 module is bundled with standard library.

However, to work with MySQL and PostgreSQL, you may have to install DB-API compatible driver modules pymysql and pyscopg2 respectively. Cockroach database is handled through playhouse extension that is installed by default along with Peewee.

Peewee is an open source project hosted on https://github.com/coleifer/peewee repository. Hence, it can be installed from here by using git.

```
git clone https://github.com/coleifer/peewee.git

cd peewee

python setup.py install
```

2. Peewee — Database Class

An object of Database class from Peewee package represents connection to a database. Peewee provides out-of-box support for SQLite, PostgreSQL and MySQL databases through corresponding subclasses of Database class.

Database class instance has all the information required to open connection with database engine, and is used to execute queries, manage transactions and perform introspection of tables, columns, etc.

Database class has **SqliteDatabase**, **PostgresqlDatabase** and **MySQLDatabase** sub-classes. While DB-API driver for SQLite in the form of sqlite3 module is included in Python's standard library, **psycopg2** and **pymysql** modules will have to be installed first for using PostgreSql and MySQL databases with Peewee.

Using Sqlite Database

Python has built-in support for SQLite database in the form of sqlite3 module. Hence, it is very easy to connect. Object of SqliteDatabase class in Peewee represents connection object.

```
con=SqliteDatabase(name, pragmas, timeout)
```

Here, **pragma** is SQLite extension which is used to modify operation of SQLite library. This parameter is either a dictionary or a list of 2-tuples containing pragma key and value to set every time a connection is opened.

Timeout parameter is specified in seconds to set busy-timeout of SQLite driver. Both the parameters are optional.

Following statement creates a connection with a new SQLite database (if it doesn't exist already).

```
>>> db = peewee.SqliteDatabase('mydatabase.db')
```

Pragma parameters are generally given for a new database connection. Typical attributes mentioned in pragmase dictionary are **journal_mode**, **cache_size**, **locking_mode**, **foreign-keys,** etc.

```
>>> db = peewee.SqliteDatabase('test.db', pragmas={'journal_mode': 'wal',
'cache_size': 10000,'foreign_keys': 1})
```

Following pragma settings are ideal to be specified:

Pragma attribute	Recommended value	Meaning
journal_mode	wal	allow readers and writers to co-exist
cache_size	-1 * data_size_kb	set page-cache size in KiB
foreign_keys	1	enforce foreign-key constraints
ignore_check_constraints	0	enforce CHECK constraints
Synchronous	0	let OS handle fsync

Peewee also has Another Python SQLite Wrapper (apsw), an advanced sqlite driver. It provides advanced features such as virtual tables and file systems, and shared connections. APSW is faster than the standard library sqlite3 module.

3. Peewee — Model

An object of Model sub class in Peewee API corresponds to a table in the database with which connection has been established. It allows performing database table operations with the help of methods defined in the Model class.

A user defined Model has one or more class attributes, each of them is an object of Field class. Peewee has a number of subclasses for holding data of different types. Examples are TextField, DatetimeField, etc. They correspond to the fields or columns in the database table. Reference of associated database and table and model configuration is mentioned in Meta class. Following attributes are used to specify configuration:

Meta class Attributes

The meta class attributes are explained below:

Attribute	Description
Database	Database for model.
db_table	Name of the table to store data. By default, it is name of model class.
Indexes	A list of fields to index.
primary_key	A composite key instance.
Constraints	A list of table constraints.
Schema	The database schema for the model.
Temporary	Indicate temporary table.
depends_on	Indicate this table depends on another for creation.
without_rowid	Indicate that table should not have rowid (SQLite only).

Following code defines Model class for User table in mydatabase.db:

```
from peewee import *
db = SqliteDatabase('mydatabase.db')
class User (Model):
    name=TextField()
    age=IntegerField()
    class Meta:

        database=db
        db_table='User'
```

```
User.create_table()
```

The **create_table()** method is a classmethod of Model class that performs equivalent CREATE TABLE query. Another instance method **save()** adds a row corresponding to object.

```
from peewee import *
db = SqliteDatabase('mydatabase.db')
class User (Model):
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'

User.create_table()
rec1=User(name="Rajesh", age=21)
rec1.save()
```

Methods in Model class

Other methods in Model class are as follows:

Classmethod alias()	Create an alias to the model-class. It allows the same Model to any referred multiple times in a query.
Classmethod select()	Performs a SELECT query operation. If no fields are explicitly provided as argument, the query will by default SELECT * equivalent.
Classmethod update()	Performs an UPDATE query function.
classmethod insert()	Inserts a new row in the underlying table mapped to model.
classmethod delete()	Executes delete query and is usually associated with a filter by where clause.
classmethod get()	Retrieve a single row from mapped table matching the given filters.
get_id()	Instance method returns primary key of a row.
save()	Save the data of object as a new row. If primary-key value is already present, it will cause an UPDATE query to be executed.
classmethod bind()	Bind the model to the given database.

4. Peewee — Field Class

Model class contains one or more attributes that are objects of Field class in Peewee. Base Field class is not directly instantiated. Peewee defines different sub classes for equivalent SQL data types.

Constructor of Field class has following parameters:

column_name (str)	Specify column name for field.
primary_key (bool)	Field is the primary key.
constraints (list)	List of constraints to apply to column
choices (list)	An iterable of 2-tuples mapping column values to display labels.
null (bool)	Field allows NULLs.
index (bool)	Create an index on field.
unique (bool)	Create an unique index on field.
Default	Default value.
collation (str)	Collation name for field.
help_text (str)	Help-text for field, metadata purposes.
verbose_name (str)	Verbose name for field, metadata purposes.

Subclasses of Field class are mapped to corresponding data types in various databases, i.e. SQLite, PostgreSQL, MySQL, etc.

Numeric Field classes

The numeric field classes in Peewee are given below:

IntegerField	Field class for storing integers.
BigIntegerField	Field class for storing big integers (maps to integer, bigint, and bigint type in SQLite, PostegreSQL and MySQL respectively).
SmallIntegerField	Field class for storing small integers (if supported by database).
FloatField	Field class for storing floating-point numbers corresponds to real data types.
DoubleField	Field class for storing double-precision floating-point numbers maps to equivalent data types in corresponding SQL databases.
DecimalField	Field class for storing decimal numbers. The parameters are mentioned below: • max_digits (int) – Maximum digits to store. • decimal_places (int) – Maximum precision. • auto_round (bool) – Automatically round values.

Text fields

The text fields which are available in Peewee are as follows:

CharField	Field class for storing strings. Max 255 characters. Equivalent SQL data type is varchar.
FixedCharField	Field class for storing fixed-length strings.
TextField	Field class for storing text. Maps to TEXT data type in SQLite and PostgreSQL, and longtext in MySQL.

Binary fields

The binary fields in Peewee are explained below:

BlobField	Field class for storing binary data.
BitField	Field class for storing options in a 64-bit integer column.
BigBitField	Field class for storing arbitrarily-large bitmaps in a Binary Large OBject (BLOB). The field will grow the underlying buffer as necessary.
UUIDField	Field class for storing universally unique identifier (UUID) objects. Maps to UUID type in Postgres. SQLite and MySQL do not have a UUID type, it is stored as a VARCHAR.

Date and Time fields

The date and time fields in Peewee are as follows:

DateTimeField	Field class for storing datetime.datetime objects. Accepts a special parameter string formats, with which the datetime can be encoded.
DateField	Field class for storing datetime.date objects. Accepts a special parameter string formats to encode date.
TimeField	Field class for storing datetime.time objectsAccepts a special parameter formats to show encoded time.

Since SQLite doesn't have DateTime data types, this field is mapped as string.

ForeignKeyField

This class is used to establish foreign key relationship in two models and hence, the respective tables in database. This class in instantiated with following parameters:

model (Model)	Model to reference. If set to 'self', it is a self-referential foreign key.
field (Field)	Field to reference on model (default is primary key).
backref (str)	Accessor name for back-reference. "+" disables the back-reference accessor.

on_delete (str)	ON DELETE action.
on_update (str)	ON UPDATE action.
lazy_load (bool)	Fetch the related object, when the foreign-key field attribute is accessed. If FALSE, accessing the foreign-key field will return the value stored in the foreign-key column.

Example

Here is an example of ForeignKeyField.

```
from peewee import *


db = SqliteDatabase('mydatabase.db')

class Customer(Model):

    id=IntegerField(primary_key=True)

    name = TextField()

    address = TextField()

    phone = IntegerField()

    class Meta:

        database=db

        db_table='Customers'


class Invoice(Model):

    id=IntegerField(primary_key=True)

    invno=IntegerField()

    amount=IntegerField()

    custid=ForeignKeyField(Customer, backref='Invoices')

    class Meta:

        database=db

        db_table='Invoices'


db.create_tables([Customer, Invoice])
```

When above script is executed, following SQL queries are run:

```
CREATE TABLE Customers (

id INTEGER NOT NULL

PRIMARY KEY,
name TEXT NOT NULL,
address TEXT NOT NULL,
phone INTEGER NOT NULL
);

CREATE TABLE Invoices (
id INTEGER NOT NULL
PRIMARY KEY,
invno INTEGER NOT NULL,
amount INTEGER NOT NULL,
custid_id INTEGER NOT NULL,
FOREIGN KEY (

custid_id
)
REFERENCES Customers (id)
);
```

When verified in SQLiteStuidio GUI tool, the table structure appears as below:

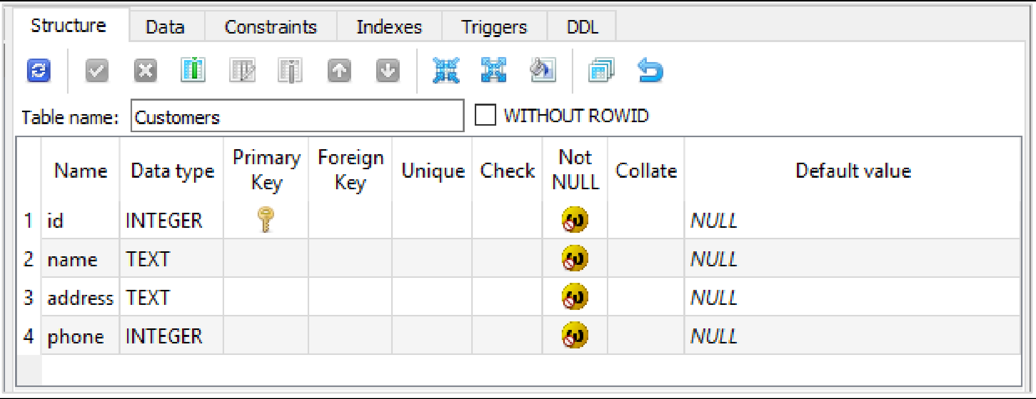


Table name: Customers ☐ WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	id	INTEGER	🔑				●		*NULL*
2	name	TEXT					●		*NULL*
3	address	TEXT					●		*NULL*
4	phone	INTEGER					●		*NULL*

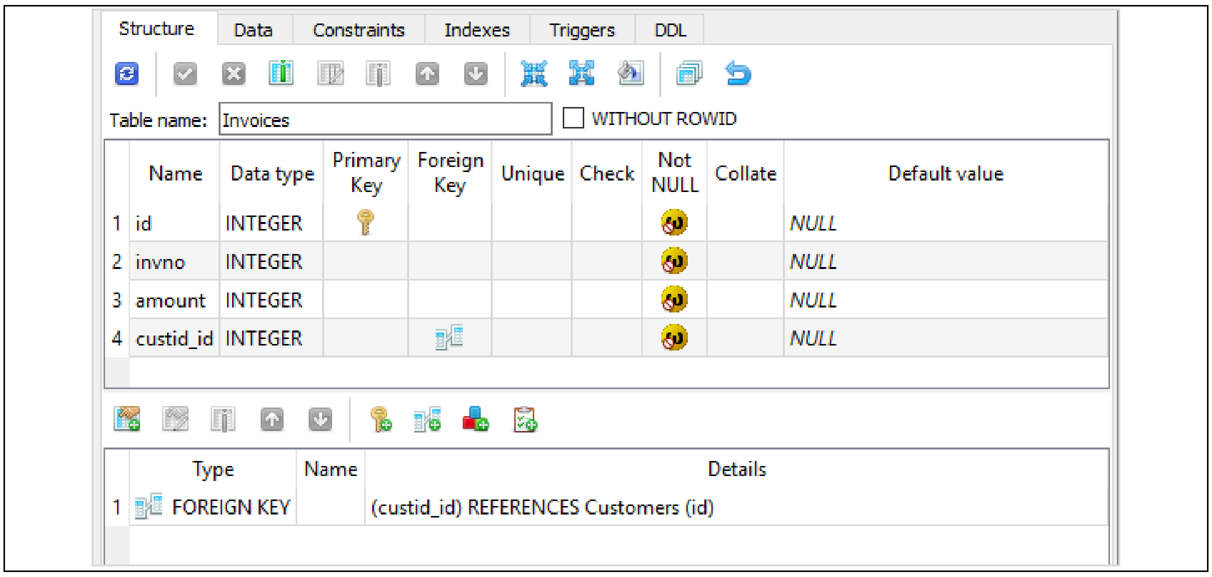


Table name: Invoices

WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	id	INTEGER	✓				✓		NULL
2	invno	INTEGER					✓		NULL
3	amount	INTEGER					✓		NULL
4	custid_id	INTEGER		✓			✓		NULL

	Type	Name	Details
1	FOREIGN KEY		(custid_id) REFERENCES Customers (id)

Other Field Types

The other field types in Peewee include:

IPField	Field class for storing IPv4 addresses efficiently (as integers).
BooleanField	Field class for storing boolean values.
AutoField	Field class for storing auto-incrementing primary keys.
IdentityField	Field class for storing auto-incrementing primary keys using the new Postgres 10 **IDENTITY** column type.

5. Peewee — Insert a New Record

In Peewee, there are more than one commands by which, it is possible to add a new record in the table. We have already used save() method of Model instance.

```
rec1=User(name="Rajesh", age=21)
rec1.save()
```

The Peewee.Model class also has a create() method that creates a new instance and add its data in the table.

```
User.create(name="Kiran", age=19)
```

In addition to this, Model also has **insert()** as class method that constructs SQL insert query object. The **execute()** method of Query object performs adding a row in underlying table.

```
q = User.insert(name='Lata', age=20)
q.execute()
```

The query object is an equivalent INSERT query.q.sql() returns the query string.

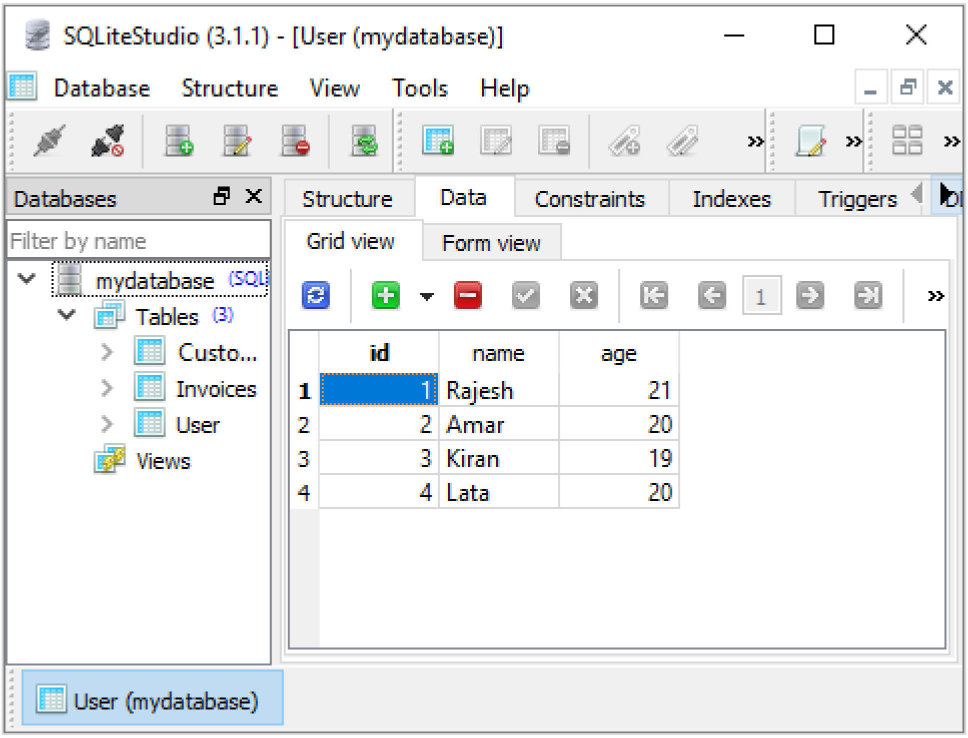
```
print (q.sql())
('INSERT INTO "User" ("name", "age") VALUES (?, ?)', ['Lata', 20])
```

Here is the complete code that demonstrates the use of above ways of inserting record.

```
from peewee import *
db = SqliteDatabase('mydatabase.db')
class User (Model):
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'

db.create_tables([User])
rec1=User(name="Rajesh", age=21)
rec1.save()
a=User(name="Amar", age=20)
```

```
a.save()
User.create(name="Kiran", age=19)
q = User.insert(name='Lata', age=20)
q.execute()
db.close()
```

We can verify the result in SQLiteStudio GUI.



	id	name	age
1	1	Rajesh	21
2	2	Amar	20
3	3	Kiran	19
4	4	Lata	20

Bulk Inserts

In order to use multiple rows at once in the table, Peewee provides two methods: **bulk_create** and **insert_many.**

insert_many()

The insert_many() method generates equivalent INSERT query, using list of dictionary objects, each having field value pairs of one object.

```
rows=[{"name":"Rajesh", "age":21}, {"name":"Amar", "age":20}]
q=User.insert_many(rows)
q.execute()
```

Here too, q.sql() returns the INSERT query string is obtained as below:

```
print (q.sql())
('INSERT INTO "User" ("name", "age") VALUES (?, ?), (?, ?)', ['Rajesh', 21,
'Amar', 20])
```

bulk_create()

This method takes a list argument that contains one or more unsaved instances of the model mapped to a table.

```
a=User(name="Kiran", age=19)
b=User(name='Lata', age=20)
User.bulk_create([a,b])
```

Following code uses both approaches to perform bulk insert operation.

```
from peewee import *
db = SqliteDatabase('mydatabase.db')
class User (Model):
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'

db.create_tables([User])
rows=[{"name":"Rajesh", "age":21}, {"name":"Amar", "age":20}]
q=User.insert_many(rows)
q.execute()
a=User(name="Kiran", age=19)
b=User(name='Lata', age=20)
User.bulk_create([a,b])
db.close()
```

6. Peewee — Select Records

Simplest and the most obvious way to retrieve data from tables is to call **select()** method of corresponding model. Inside select() method, we can specify one or more field attributes. However, if none is specified, all columns are selected.

Model.select() returns a list of model instances corresponding to rows. This is similar to the result set returned by SELECT query, which can be traversed by a for loop.

```
from peewee import *
db = SqliteDatabase('mydatabase.db')
class User (Model):
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'
rows=User.select()
print (rows.sql())
for row in rows:
    print ("name: {} age: {}".format(row.name, row.age))
db.close()
```

The above script displays the following output:

```
('SELECT "t1"."id", "t1"."name", "t1"."age" FROM "User" AS "t1"', [])
name: Rajesh age: 21
name: Amar age: 20
name: Kiran age: 19
name: Lata age: 20
```

7. Peewee — Filters

It is possible to retrieve data from SQLite table by using where clause. Peewee supports following list of logical operators.

==	x equals y
<	x is less than y
<=	x is less than or equal to y
>	x is greater than y
>=	x is greater than or equal to y
!=	x is not equal to y
<<	x IN y, where y is a list or query
>>	x IS y, where y is None/NULL
%	x LIKE y where y may contain wildcards
**	x ILIKE y where y may contain wildcards
^	x XOR y
~	Unary negation (e.g., NOT x)

Following code displays name with **age>=20:**

```
rows=User.select().where (User.age>=20)
for row in rows:
    print ("name: {} age: {}".format(row.name, row.age))
```

Following code displays only those name present in the names list.

```
names=['Anil', 'Amar', 'Kiran', 'Bala']
rows=User.select().where (User.name << names)
for row in rows:
    print ("name: {} age: {}".format(row.name, row.age))
```

The SELECT query thus generated by Peewee will be:

```
('SELECT "t1"."id", "t1"."name", "t1"."age" FROM "User" AS "t1" WHERE
("t1"."name" IN (?, ?, ?, ?))', ['Anil', 'Amar', 'Kiran', 'Bala'])
```

Resultant output will be as follows:

```
name: Amar age: 20
name: Kiran age: 19
```

Filtering Methods

In addition to the above logical operators as defined in core Python, Peewee provides following methods for filtering:

.in_(value)	IN lookup (identical to <<).
.not_in(value)	NOT IN lookup.
.is_null(is_null)	IS NULL or IS NOT NULL. Accepts boolean param.
.contains(substr)	Wild-card search for substring.
.startswith(prefix)	Search for values beginning with prefix.
.endswith(suffix)	Search for values ending with suffix.
.between(low, high)	Search for values between low and high.
.regexp(exp)	Regular expression match (case-sensitive).
.iregexp(exp)	Regular expression match (case-insensitive).
.bin_and(value)	Binary AND.
.bin_or(value)	Binary OR.
.concat(other)	Concatenate two strings or objects using ||.
.distinct()	Mark column for DISTINCT selection.
.collate(collation)	Specify column with the given collation.
.cast(type)	Cast the value of the column to the given type.

As an example of above methods, look at the following code. It retrieves names starting with 'R' or ending with 'r'.

```
rows=User.select().where (User.name.startswith('R') | User.name.endswith('r'))
```

Equivalent SQL SELECT query is:

```
('SELECT "t1"."id", "t1"."name", "t1"."age" FROM "User" AS "t1" WHERE
(("t1"."name" LIKE ?) OR ("t1"."name" LIKE ?))', ['R%', '%r'])
```

Alternatives

Python's built-in operators in, not in, and, or etc. will not work. Instead, use Peewee alternatives.

You can use:

- .in_() and .not_in() methods instead of in and not in operators.
- & instead of and.

- | instead of or.
- ~ instead of not.
- .is_null() instead of is.
- None or == None.

8. Peewee — Primary and Composite Keys

It is recommended that the table in a relational database, should have one of the columns applied with primary key constraint. Accordingly, Peewee Model class can also specify field attribute with primary-key argument set to True. However, if model class doesn't have any primary key, Peewee automatically creates one with the name "id". Note that the User model defined above doesn't have any field explicitly defined as primary key. Hence, the mapped User table in our database has an id field.

To define an auto-incrementing integer primary key, use **AutoField** object as one attribute in the model.

```
class User (Model):
    user_id=AutoField()
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'
```

This will translate into following CREATE TABLE query:

```
CREATE TABLE User (
user_id INTEGER NOT NULL
PRIMARY KEY,
name TEXT NOT NULL,
age INTEGER NOT NULL
);
```

You can also assign any non-integer field as a primary key by setting primary_key parameter to True. Let us say we want to store certain alphanumeric value as user_id.

```
class User (Model):
    user_id=TextField(primary_key=True)
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'
```

However, when model contains non-integer field as primary key, the **save()** method of model instance doesn't cause database driver to generate new ID automatically, hence we need to pass **force_insert=True** parameter. However, note that the **create()** method implicitly specifies force_insert parameter.

```
User.create(user_id='A001',name="Rajesh", age=21)
b=User(user_id='A002',name="Amar", age=20)
b.save(force_insert=True)
```

The save() method also updates an existing row in the table, at which time, force_insert primary is not necessary, as ID with unique primary key is already existing.

Peewee allows feature of defining composite primary key. Object of **CompositeKey** class is defined as primary key in Meta class. In following example, a composite key consisting of name and city fields of User model has been assigned as composite key.

```
class User (Model):
    name=TextField()
    city=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'
        primary_key=CompositeKey('name', 'city')
```

This model translates in the following CREATE TABLE query.

```
CREATE TABLE User (
name TEXT NOT NULL,
city TEXT NOT NULL,
age INTEGER NOT NULL,
PRIMARY KEY (
name,
city
)
);
```

If you wish, the table should not have a primary key, then specify primary_key=False in model's Meta class.

9. Peewee — Update Existing Records

Existing data can be modified by calling **save()** method on model instance as well as with **update()** class method.

Following example fetches a row from User table with the help of **get()** method and updates it by changing the value of age field.

```
row=User.get(User.name=="Amar")
print ("name: {} age: {}".format(row.name, row.age))
row.age=25
row.save()
```

The **update()** method of Method class generates UPDATE query. The query object's execute() method is then invoked.

Following example uses update() method to change the age column of rows in which it is >20.

```
qry=User.update({User.age:25}).where(User.age>20)
print (qry.sql())
qry.execute()
```

The SQL query rendered by update() method is as follows:

```
('UPDATE "User" SET "age" = ? WHERE ("User"."age" > ?)', [25, 20])
```

Peewee also has a **bulk_update()** method to help update multiple model instance in a single query operation. The method requires model objects to be updated and list of fields to be updated.

Following example updates the age field of specified rows by new value.

```
rows=User.select()
rows[0].age=25
rows[2].age=23
User.bulk_update([rows[0], rows[2]], fields=[User.age])
```

10. Peewee — Delete Records

Running **delete_instance()** method on a model instance delete corresponding row from the mapped table.

```
obj=User.get(User.name=="Amar")
obj.delete_instance()
```

On the other hand, delete() is a class method defined in model class, which generates DELETE query. Executing it effectively deletes rows from the table.

```
db.create_tables([User])
qry=User.delete().where (User.age==25)
qry.execute()
```

Concerned table in database shows effect of DELETE query as follows:

```
('DELETE FROM "User" WHERE ("User"."age" = ?)', [25])
```

11. Peewee — Create Index

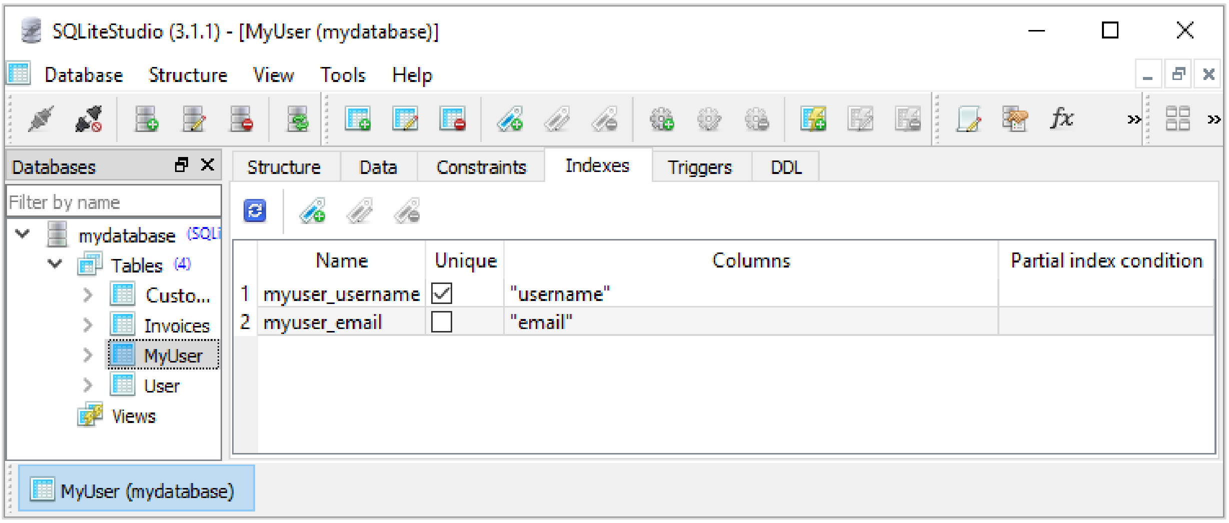
By using Peewee ORM, it is possible to define a model which will create a table with index on single column as well as multiple columns.

As per the Field attribute definition, setting unique constraint to True will create an index on the mapped field. Similarly, passing index=True parameter to field constructor also create index on the specified field.

In following example, we have two fields in MyUser model, with username field having unique parameter set to True and email field has **index=True.**

```
class MyUser(Model):
    username = CharField(unique=True)
    email = CharField(index=True)
    class Meta:
        database=db
        db_table='MyUser'
```

As a result, SQLiteStudio graphical user interface (GUI) shows indexes created as follows:


SQLiteStudio (3.1.1) - [MyUser (mydatabase)]
Database Structure View Tools Help
Databases
Filter by name
mydatabase
Tables (4)
Custo...
Invoices
MyUser
User
Views
Structure Data Constraints Indexes Triggers DDL
MyUser (mydatabase)

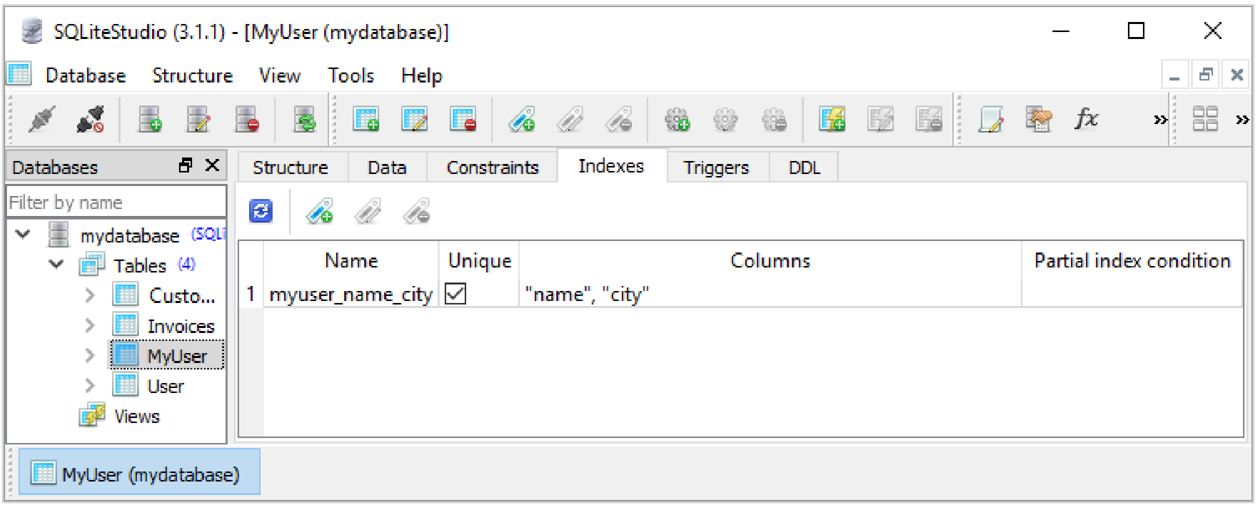
	Name	Unique	Columns	Partial index condition
1	myuser_username	✓	"username"	
2	myuser_email	☐	"email"	

In order to define a multi-column index, we need to add indexes attribute in Meta class inside definition of our model class. It is a tuple of 2-item tuples, one tuple for one index definition. Inside each 2-element tuple, the first part of which is a tuple of the names of the fields, the second part is set to True to make it unique, and otherwise is False.

We define MyUser model with a two-column unique index as follows:

```
class MyUser (Model):


    name=TextField()
    city=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='MyUser'
        indexes=(
            (('name', 'city'), True),
            )
```

Accordingly, SQLiteStudio shows index definition as in the following figure:


SQLiteStudio (3.1.1) - [MyUser (mydatabase)]
Database Structure View Tools Help
Databases
Filter by name
mydatabase (SQLi
Tables (4)
Custo...
Invoices
MyUser
User
Views
Structure Data Constraints Indexes Triggers DDL
MyUser (mydatabase)

	Name	Unique	Columns	Partial index condition
1	myuser_name_city	✓	"name", "city"	

Index can be built outside model definition as well.

You can also create index by manually providing SQL helper statement as parameter to **add_index()** method.

```
MyUser.add_index(SQL('CREATE INDEX idx on MyUser(name);'))
```

Above method is particularly required when using SQLite. For MySQL and PostgreSQL, we can obtain Index object and use it with **add_index()** method.

```
ind=MyUser.index(MyUser.name)
MyUser.add_index(ind)
```

12. Peewee — Constraints

Constraints are restrictions imposed on the possible values that can be put in a field. One such constraint is primary key. When **primary_key=True** is specified in Field definition, each row can only store unique value – same value of the field cannot be repeated in another row.

If a field is not a primary key, still it can be constrained to store **unique** values in table. Field constructor also has constraints parameter.

Following example applies **CHECK** constraint on age field.

```
class MyUser (Model):
    name=TextField()
    city=TextField()
    age=IntegerField(constraints=[Check('name<10')])
    class Meta:
        database=db
        db_table='MyUser'
```

This will generate following Data Definition Language (DDL) expression:

```
CREATE TABLE MyUser (
id INTEGER NOT NULL
PRIMARY KEY,
name TEXT NOT NULL,
city TEXT NOT NULL,
age INTEGER NOT NULL
CHECK (name < 10)
);
```

As a result, if a new row with age<10 will result in error.

```
MyUser.create(name="Rajesh", city="Mumbai",age=9)
peewee.IntegrityError: CHECK constraint failed: MyUser
```

In the field definition, we can also use **DEFAULT** constraint as in following definition of city field.

```
city=TextField(constraints=[SQL("DEFAULT 'Mumbai'")])
```

So, the model object can be constructed with or without explicit value of city. If not used, city field will be filled by default value – Mumbai.

13. Peewee — Using MySQL

As mentioned earlier, Peewee supports MySQL database through **MySQLDatabase** class. However, unlike SQLite database, Peewee can't create a **MySql** database. You need to create it manually or using functionality of DB-API compliant module such as **pymysql**.

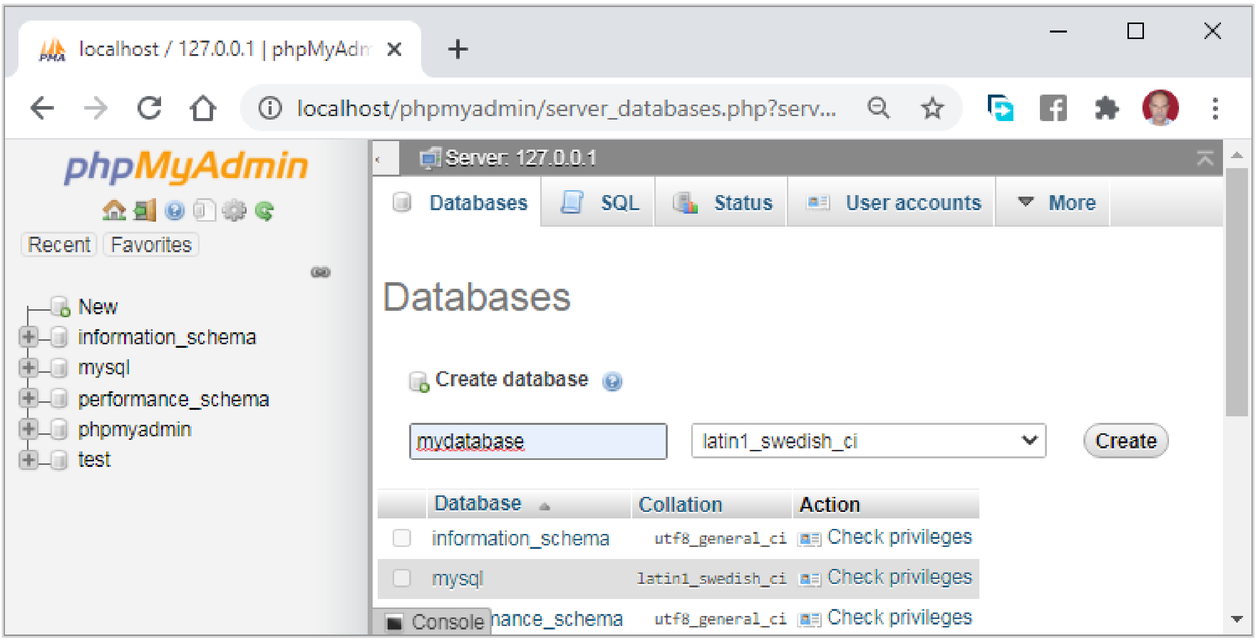
First, you should have MySQL server installed in your machine. It can be a standalone MySQL server installed from https://dev.mysql.com/downloads/installer/.

You can also work on Apache bundled with MySQL (such as XAMPP downloaded and installed from https://www.apachefriends.org/download.html).

Next, we install pymysql module, DB-API compatible Python driver.

```
pip install pymysql
```

The create a new database named mydatabase. We shall use **phpmyadmin** interface available in XAMPP.



Databases

Create database

mydatabase | latin1_swedish_ci | Create

Database	Collation	Action
information_schema	utf8_general_ci	Check privileges
mysql	latin1_swedish_ci	Check privileges
nance_schema	utf8_general_ci	Check privileges

If you choose to create database programmatically, use following Python script:

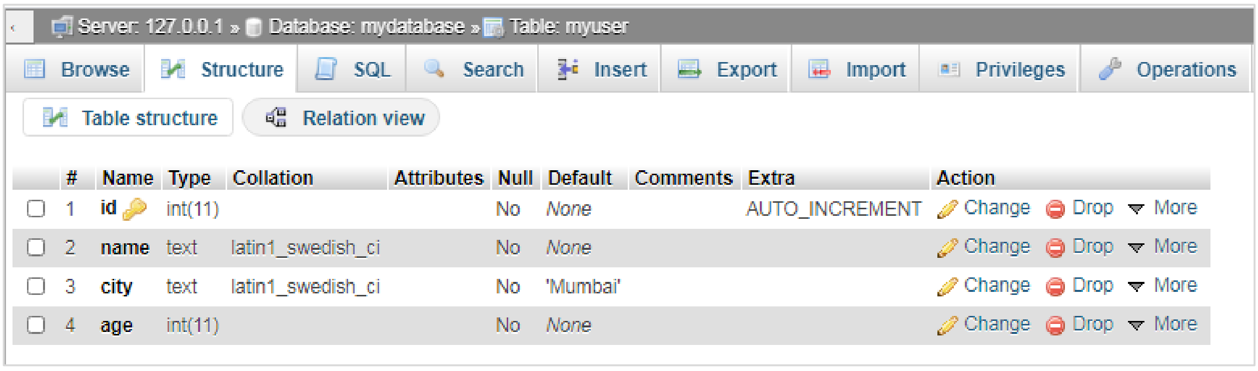
```
import pymysql


conn = pymysql.connect(host='localhost', user='root', password='')

conn.cursor().execute('CREATE DATABASE mydatabase')

conn.close()
```

Once a database is created on the server, we can now declare a model and thereby, create a mapped table in it.

The MySQLDatabase object requires server credentials such as host, port, user name and password.

```
from peewee import *
db = MySQLDatabase('mydatabase', host='localhost', port=3306, user='root',
password='')
class MyUser (Model):
    name=TextField()
    city=TextField(constraints=[SQL("DEFAULT 'Mumbai'")])
    age=IntegerField()
    class Meta:
        database=db
        db_table='MyUser'

db.connect()
db.create_tables([MyUser])
```

The Phpmyadmin web interface now shows myuser table created.



	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
☐	2	name	text	latin1_swedish_ci		No	None			Change Drop More
☐	3	city	text	latin1_swedish_ci		No	'Mumbai'			Change Drop More
☐	4	age	int(11)			No	None			Change Drop More

14. Peewee — Using PostgreSQL

Peewee supports PostgreSQL database as well. It has **PostgresqlDatabase** class for that purpose. In this chapter, we shall see how we can connect to Postgres database and create a table in it, with the help of Peewee model.

As in case of MySQL, it is not possible to create database on Postgres server with Peewee's functionality. The database has to be created manually using Postgres shell or **PgAdmin** tool.

First, we need to install Postgres server. For windows OS, we can download https://get.enterprisedb.com/postgresql/postgresql-13.1-1-windows-x64.exe and install.

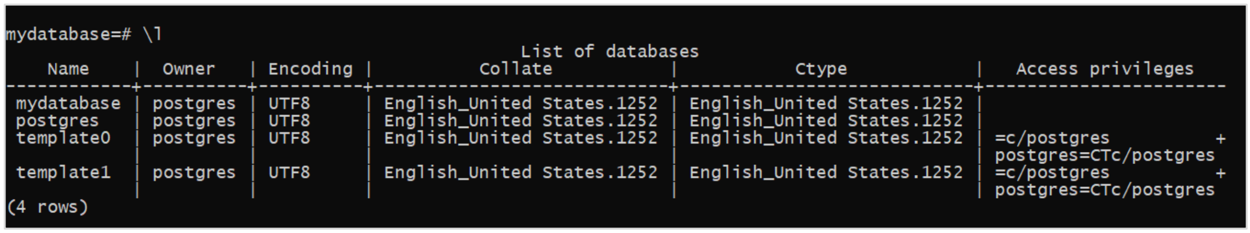
Next, install Python driver for Postgres – **Psycopg2** package using pip installer.

```
pip install psycopg2
```

Then start the server, either from PgAdmin tool or psql shell. We are now in a position to create a database. Run following Python script to create mydatabase on Postgres server.

```
import psycopg2


conn = psycopg2.connect(host='localhost', user='postgres', password='postgres')
conn.cursor().execute('CREATE DATABASE mydatabase')
conn.close()
```

Check that the database is created. In **psql** shell, it can be verified with \l command:

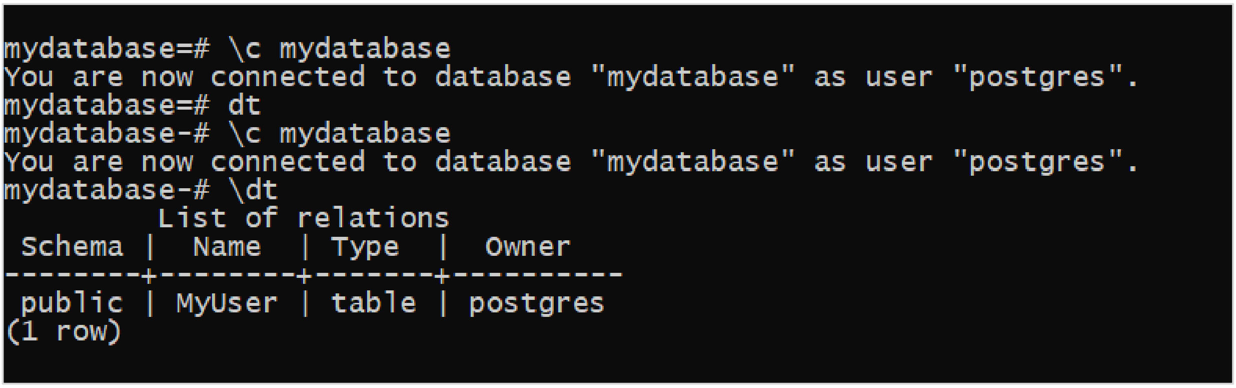


```
mydatabase=# \l
                                                    List of databases
    Name    |  Owner   | Encoding |          Collate           |           Ctype            |   Access privileges
------------+----------+----------+----------------------------+----------------------------+-----------------------
 mydatabase | postgres | UTF8     | English_United States.1252 | English_United States.1252 |
 postgres   | postgres | UTF8     | English_United States.1252 | English_United States.1252 |
 template0  | postgres | UTF8     | English_United States.1252 | English_United States.1252 | =c/postgres          +
            |          |          |                            |                            | postgres=CTc/postgres
 template1  | postgres | UTF8     | English_United States.1252 | English_United States.1252 | =c/postgres          +
            |          |          |                            |                            | postgres=CTc/postgres
(4 rows)
```

To declare MyUser model and create a table of same name in above database, run following Python code:

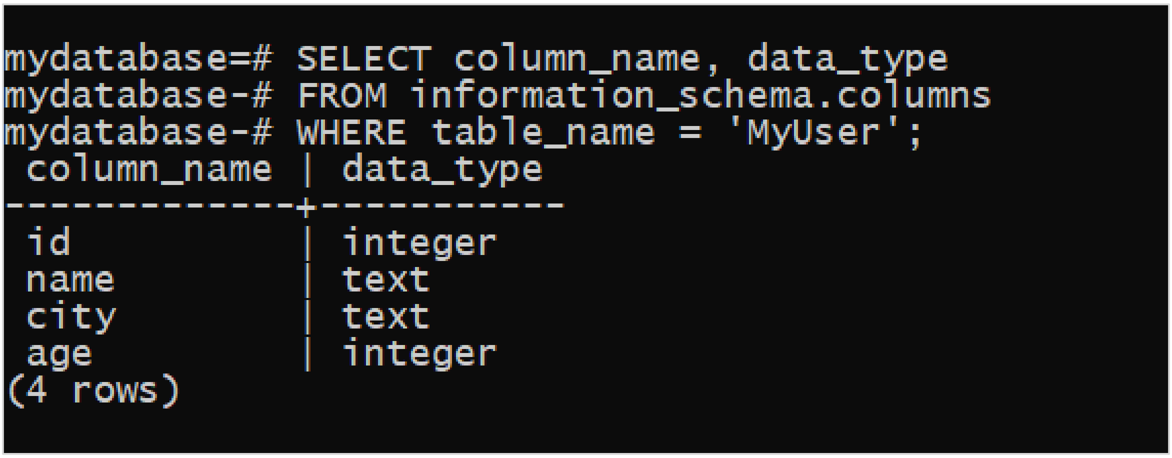
```
from peewee import *


db = PostgresqlDatabase('mydatabase', host='localhost', port=5432,
user='postgres', password='postgres')
class MyUser (Model):
    name=TextField()
    city=TextField(constraints=[SQL("DEFAULT 'Mumbai'")])
    age=IntegerField()
    class Meta:
        database=db
        db_table='MyUser'

db.connect()
db.create_tables([MyUser])
```

We can verify that table is created. Inside the shell, connect to mydatabase and get list of tables in it.



```
mydatabase=# \c mydatabase
You are now connected to database "mydatabase" as user "postgres".
mydatabase=# dt
mydatabase-# \c mydatabase
You are now connected to database "mydatabase" as user "postgres".
mydatabase-# \dt
         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | MyUser | table | postgres
(1 row)
```

To check structure of newly created MyUser database, run following query in the shell.



```
mydatabase=# SELECT column_name, data_type
mydatabase-# FROM information_schema.columns
mydatabase-# WHERE table_name = 'MyUser';
 column_name | data_type
-------------+-----------
 id          | integer
 name        | text
 city        | text
 age         | integer
(4 rows)
```

15. Peewee — Defining Database Dynamically

If your database is scheduled to vary at run-time, use **DatabaseProxy** helper to have better control over how you initialise it. The DatabaseProxy object is a placeholder with the help of which database can be selected in run-time.

In the following example, an appropriate database is selected depending on the application's configuration setting.

```
from peewee import *
db_proxy = DatabaseProxy()  # Create a proxy for our db.

class MyUser (Model):
    name=TextField()
    city=TextField(constraints=[SQL("DEFAULT 'Mumbai'")])
    age=IntegerField()
    class Meta:
        database=db_proxy
        db_table='MyUser'

# Based on configuration, use a different database.
if app.config['TESTING']:
    db = SqliteDatabase(':memory:')
elif app.config['DEBUG']:
    db = SqliteDatabase('mydatabase.db')
else:
    db = PostgresqlDatabase('mydatabase', host='localhost', port=5432,
user='postgres', password='postgres')

# Configure our proxy to use the db we specified in config.
db_proxy.initialize(db)

db.connect()
db.create_tables([MyUser])
```

You can also associate models to any database object during run-time using **bind()** method declared in both database class and model class.

Following example uses bind() method in database class.

```
from peewee import *


class MyUser (Model):
    name=TextField()
    city=TextField(constraints=[SQL("DEFAULT 'Mumbai'")])
    age=IntegerField()


db = MySQLDatabase('mydatabase', host='localhost', port=3306, user='root',
password='')
db.connect()
db.bind([MyUser])
db.create_tables([MyUser])
```

The same bind() method is also defined in Model class.

```
from peewee import *


class MyUser (Model):
    name=TextField()
    city=TextField(constraints=[SQL("DEFAULT 'Mumbai'")])
    age=IntegerField()


db = MySQLDatabase('mydatabase', host='localhost', port=3306, user='root',
password='')
db.connect()
MyUser.bind(db)
db.create_tables([MyUser])
```

16. Peewee — Connection Management

Database object is created with **autoconnect** parameter set as True by default. Instead, to manage database connection programmatically, it is initially set to False.

```
db=SqliteDatabase("mydatabase", autoconnect=False)
```

The database class has **connect()** method that establishes connection with the database present on the server.

```
db.connect()
```

It is always recommended to close the connection at the end of operations performed.

```
db.close()
```

If you try to open an already open connection, Peewee raises **OperationError**.

```
>>> db.connect()
True
>>> db.connect()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c:\peewee\lib\site-packages\peewee.py", line 3031, in connect
    raise OperationalError('Connection already opened.')
peewee.OperationalError: Connection already opened.
```

To avoid this error, use **reuse_if_open=True** as argument to connect() method.

```
>>> db.connect(reuse_if_open=True)
False
```

Calling **close()** on already closed connection won't result error. You can however, check if the connection is already closed with **is_closed()** method.

```
>>> if db.is_closed()==True:
        db.connect()


True
>>>
```

Instead of explicitly calling db.close() in the end, it is also possible to use database object as **context_manager**.

```
from peewee import *



db = SqliteDatabase('mydatabase.db', autoconnect=False)


class User (Model):
    user_id=TextField(primary_key=True)
    name=TextField()
    age=IntegerField()
    class Meta:
        database=db
        db_table='User'
with db:
    db.connect()
    db.create_tables([User])
```

17. Peewee — Relationships and Joins

Peewee supports implementing different type of SQL JOIN queries. Its Model class has a **join()** method that returns a Join instance.

```
M1.joint(m2, join_type, on)
```

The joins table mapped with M1 model to that of m2 model and returns Join class instance. The on parameter is None by default and is expression to use as join predicate.

Join Types

Peewee supports following Join types (Default is INNER).

- JOIN.INNER
- JOIN.LEFT_OUTER
- JOIN.RIGHT_OUTER
- JOIN.FULL
- JOIN.FULL_OUTER
- JOIN.CROSS

To show use of join() method, we first declare following models:

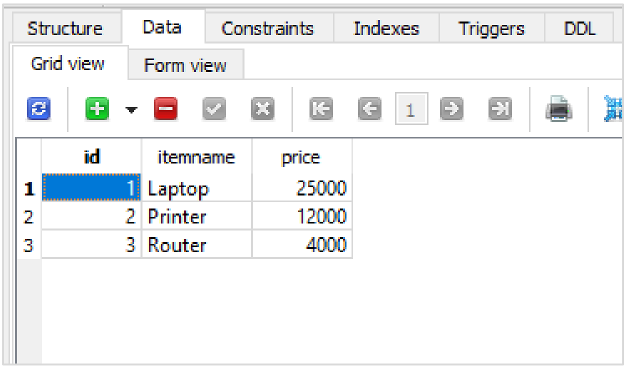
```
db = SqliteDatabase('mydatabase.db')

class BaseModel(Model):
    class Meta:
        database = db

class Item(BaseModel):
    itemname = TextField()
    price = IntegerField()

class Brand(BaseModel):
    brandname = TextField()
    item = ForeignKeyField(Item, backref='brands')

class Bill(BaseModel):
    item = ForeignKeyField(Item, backref='bills')
    brand = ForeignKeyField(Brand, backref='bills')
    qty = DecimalField()

db.create_tables([Item, Brand, Bill])
```

Tables

Next, we populate these tables with following test data:

Item Table

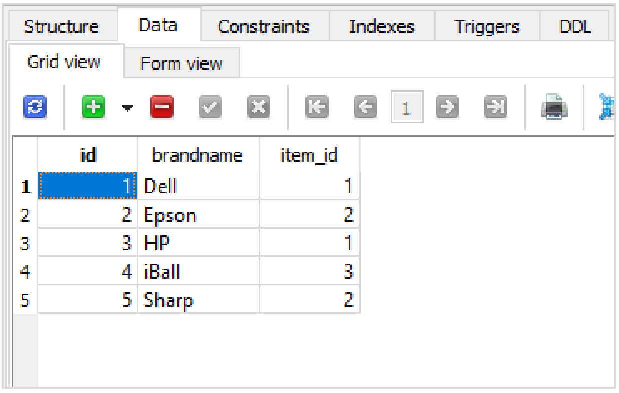
The item table is given below:



	id	itemname	price
1	1	Laptop	25000
2	2	Printer	12000
3	3	Router	4000

Brand Table

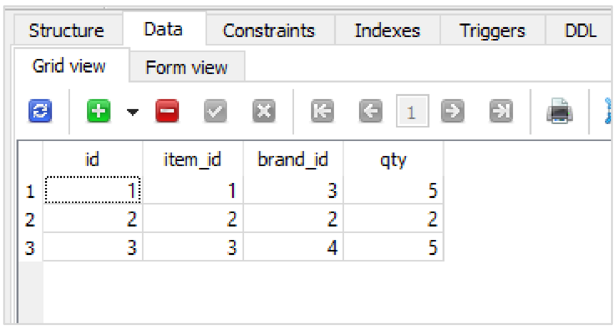
Given below is the brand table:



	id	brandname	item_id
1	1	Dell	1
2	2	Epson	2
3	3	HP	1
4	4	iBall	3
5	5	Sharp	2

Bill Table

The bill table is as follows:



	id	item_id	brand_id	qty
1	1	1	3	5
2	2	2	2	2
3	3	3	4	5

To perform a simple join operation between Brand and Item tables, execute the following code:

```
qs=Brand.select().join(Item)
for q in qs:
    print ("Brand ID:{} Item Name: {} Price: {}".format(q.id, q.brandname,
q.item.price))
```

The resultant output will be as follows:

```
Brand ID:1 Item Name: Dell Price: 25000
Brand ID:2 Item Name: Epson Price: 12000
Brand ID:3 Item Name: HP Price: 25000
Brand ID:4 Item Name: iBall Price: 4000
Brand ID:5 Item Name: Sharp Price: 12000
```

Joining Multiple Tables

We have a Bill model having two foreign key relationships with item and brand models. To fetch data from all three tables, use following code:

```
qs=Bill.select().join(Brand).join(Item)
for q in qs:
    print ("BillNo:{} Brand:{} Item:{} price:{} Quantity:{}".format(q.id, \
           q.brand.brandname, q.item.itemname, q.item.price, q.qty))
```

Following output will be displayed, based on our test data:

```
BillNo:1 Brand:HP Item:Laptop price:25000 Quantity:5
BillNo:2 Brand:Epson Item:Printer price:12000 Quantity:2
BillNo:3 Brand:iBall Item:Router price:4000 Quantity:5
```

18. Peewee — Subqueries

In SQL, a subquery is an embedded query in WHERE clause of another query. We can implement subquery as a **model.select()** as a parameter inside where attribute of outer model.select() statement.

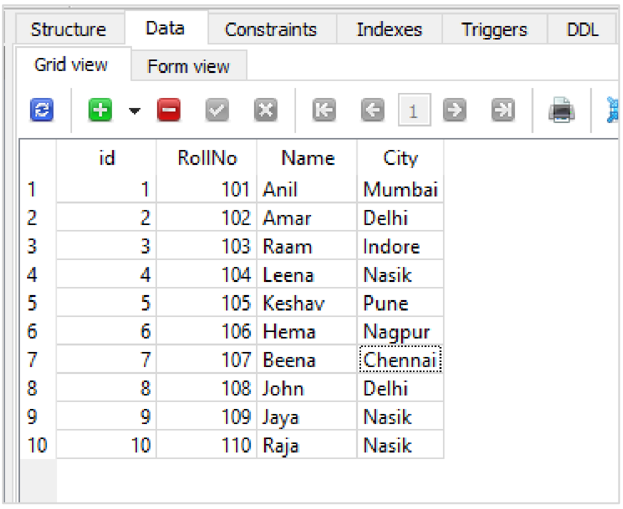
To demonstrate use of subquery in Peewee, let us use defined following models:

```
from peewee import *

db = SqliteDatabase('mydatabase.db')

class BaseModel(Model):
    class Meta:
        database = db

class Contacts(BaseModel):
    RollNo = IntegerField()
    Name = TextField()
    City = TextField()

class Branches(BaseModel):
    RollNo = IntegerField()
    Faculty = TextField()

db.create_tables([Contacts, Branches])
```

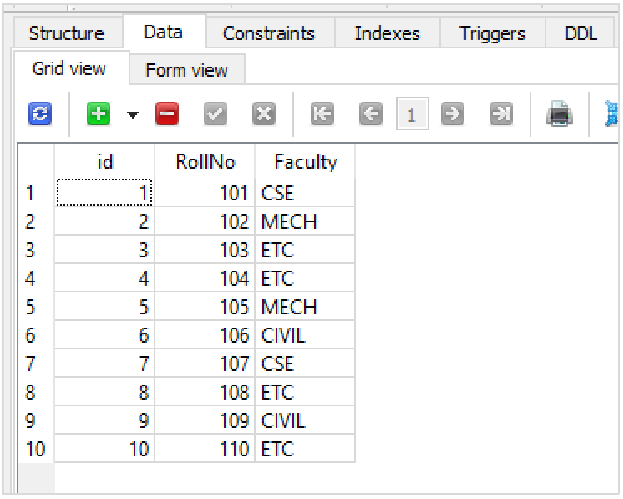
After tables are created, they are populated with following sample data:

Contacts table

The contacts table is given below:



	id	RollNo	Name	City
1	1	101	Anil	Mumbai
2	2	102	Amar	Delhi
3	3	103	Raam	Indore
4	4	104	Leena	Nasik
5	5	105	Keshav	Pune
6	6	106	Hema	Nagpur
7	7	107	Beena	Chennai
8	8	108	John	Delhi
9	9	109	Jaya	Nasik
10	10	110	Raja	Nasik



	id	RollNo	Faculty
1	1	101	CSE
2	2	102	MECH
3	3	103	ETC
4	4	104	ETC
5	5	105	MECH
6	6	106	CIVIL
7	7	107	CSE
8	8	108	ETC
9	9	109	CIVIL
10	10	110	ETC

In order to display name and city from contact table only for RollNo registered for ETC faculty, following code generates a SELECT query with another SELECT query in its WHERE clause.

```
#this query is used as subquery
faculty=Branches.select(Branches.RollNo).where(Branches.Faculty=="ETC")
names=Contacts.select().where (Contacts.RollNo .in_(faculty))


print ("RollNo and City for Faculty='ETC'")
for name in names:
    print ("RollNo:{} City:{}".format(name.RollNo, name.City))


db.close()
```

Above code will display the following result:

```
RollNo and City for Faculty='ETC'
RollNo:103 City:Indore
```

```
RollNo:104 City:Nasik
RollNo:108 City:Delhi
RollNo:110 City:Nasik
```

19. Peewee — Sorting

It is possible to select records from a table using **order_by** clause along with model's select() method. Additionally, by attaching **desc()** to the field attribute on which sorting is to be performed, records will be collected in descending order.

Following code display records from contact table in ascending order of City names.

```
rows=Contacts.select().order_by(Contacts.City)
print ("Contact list in order of city")
for row in rows:
    print ("RollNo:{} Name: {} City:{}".format(row.RollNo,row.Name, row.City))
```

Here is the sorted list which is arranged according to ascending order of city name.

```
Contact list in order of city
RollNo:107 Name: Beena City:Chennai
RollNo:102 Name: Amar City:Delhi
RollNo:108 Name: John City:Delhi
RollNo:103 Name: Raam City:Indore
RollNo:101 Name: Anil City:Mumbai
RollNo:106 Name: Hema City:Nagpur
RollNo:104 Name: Leena City:Nasik
RollNo:109 Name: Jaya City:Nasik
RollNo:110 Name: Raja City:Nasik
RollNo:105 Name: Keshav City:Pune
```

Following code displays list in descending order of Name field.

```
rows=Contacts.select().order_by(Contacts.Name.desc())
print ("Contact list in descending order of Name")
for row in rows:
    print ("RollNo:{} Name: {} City:{}".format(row.RollNo,row.Name, row.City))
```

The output is as follows:

```
Contact list in descending order of Name
RollNo:110 Name: Raja City:Nasik
RollNo:103 Name: Raam City:Indore
```

```
RollNo:104 Name: Leena City:Nasik
RollNo:105 Name: Keshav City:Pune
RollNo:108 Name: John City:Delhi
RollNo:109 Name: Jaya City:Nasik
RollNo:106 Name: Hema City:Nagpur
RollNo:107 Name: Beena City:Chennai
RollNo:101 Name: Anil City:Mumbai
RollNo:102 Name: Amar City:Delhi
```

20. Peewee — Counting and Aggregation

We can find number of records reported in any SELECT query by attaching **count()** method. For example, following statement returns number of rows in Contacts table with City='Nasik'.

```
qry=Contacts.select().where (Contacts.City=='Nasik').count()
print (qry)
```

SQL has **GROUP BY** clause in SELECT query. Peewee supports it in the form of **group_by()** method. Following code returns city wise count of names in Contacts table.

```
from peewee import *

db = SqliteDatabase('mydatabase.db')
class Contacts(BaseModel):
    RollNo = IntegerField()
    Name = TextField()
    City = TextField()
    class Meta:
        database = db

db.create_tables([Contacts])

qry=Contacts.select(Contacts.City,
fn.Count(Contacts.City).alias('count')).group_by(Contacts.City)
print (qry.sql())
for q in qry:
    print (q.City, q.count)
```

The SELECT query emitted by Peewee will be as follows:

```
('SELECT "t1"."City", Count("t1"."City") AS "count" FROM "contacts" AS "t1"
GROUP BY "t1"."City"', [])
```

As per sample data in Contacts table, following output is displayed:

```
Chennai 1
Delhi 2
Indore 1
```

```
Mumbai 1

Nagpur 1

Nasik 3

Pune 1
```

21. Peewee — SQL Functions

American National Standards Institute (ANSI) Structured Query Language (SQL) standard defines many SQL functions.

Aggregate functions like the following are useful in Peewee.

- AVG() - Returns the average value.
- COUNT() - Returns the number of rows.
- FIRST() - Returns the first value.
- LAST() - Returns the last value.
- MAX() - Returns the largest value.
- MIN() - Returns the smallest value.
- SUM() - Returns the sum.

In order to implement these SQL functions, Peewee has a SQL helper function **fn().** In above example, we used it to find count of records for each city.

Following example builds a SELECT query that employs SUM() function.

Using Bill and Item tables from models defined earlier, we shall display sum of quantity of each item as entered in Bill table.

Item table

The item table with the data is given below:

Id	Item Name	Price
1	Laptop	25000
2	Printer	12000
3	Router	4000

Bill table

The bill table is as follows:

Id	Item_id	Brand_id	Quantity
1	1	3	5
2	2	2	2
3	3	4	5
4	2	2	6
5	3	4	3
6	1	3	1

We create a join between Bill and Item table, select item name from Item table and sum of quantity from Bill table.

```
from peewee import *


db = SqliteDatabase('mydatabase.db')




class BaseModel(Model):
    class Meta:
        database = db


class Item(BaseModel):
    itemname = TextField()
    price = IntegerField()


class Brand(BaseModel):
    brandname = TextField()
    item = ForeignKeyField(Item, backref='brands')


class Bill(BaseModel):
    item = ForeignKeyField(Item, backref='bills')
    brand = ForeignKeyField(Brand, backref='bills')
    qty = DecimalField()


db.create_tables([Item, Brand, Bill])


qs=Bill.select(Item.itemname,
fn.SUM(Bill.qty).alias('Sum')).join(Item).group_by(Item.itemname)
print (qs)
for q in qs:
    print ("Item: {} sum: {}".format(q.item.itemname, q.Sum))


db.close()
```

Above script executes the following SELECT query:

```
SELECT "t1"."itemname", SUM("t2"."qty") AS "Sum" FROM "bill" AS "t2" INNER JOIN
"item" AS "t1" ON ("t2"."item_id" = "t1"."id") GROUP BY "t1"."itemname"
```

Accordingly, the output is as follows:

```
Item: Laptop sum: 6
```

```
Item: Printer sum: 8
Item: Router sum: 8
```

22. Peewee — Retrieving Row Tuples/Dictionaries

It is possible to iterate over the resultset without creating model instances. This may be achieved by using the following:

- tuples() method.
- dicts() method.

To return data of fields in SELECT query as collection of tuples, use **tuples()** method.

```
qry=Contacts.select(Contacts.City,
fn.Count(Contacts.City).alias('count')).group_by(Contacts.City).tuples()
lst=[]
for q in qry:
    lst.append(q)
print (lst)
```

The output is given below:

```
[('Chennai', 1), ('Delhi', 2), ('Indore', 1), ('Mumbai', 1), ('Nagpur', 1),
('Nasik', 3), ('Pune', 1)]
```

To obtain collection of dictionary objects:

```
qs=Brand.select().join(Item).dicts()
lst=[]
for q in qs:
    lst.append(q)
print (lst)
```

The output is stated below:

```
[{'id': 1, 'brandname': 'Dell', 'item': 1}, {'id': 2, 'brandname': 'Epson',
'item': 2}, {'id': 3, 'brandname': 'HP', 'item': 1}, {'id': 4, 'brandname':
'iBall', 'item': 3}, {'id': 5, 'brandname': 'Sharp', 'item': 2}]
```

23. Peewee — User defined Operators

Peewee has **Expression** class with the help of which we can add any customized operator in Peewee's list of operators. Constructor for Expression requires three arguments, left operand, operator and right operand.

```
op=Expression(left, operator, right)
```

Using Expression class, we define a **mod()** function that accepts arguments for left and right and '%' as operator.

```
from peewee import Expression # the building block for expressions

def mod(lhs, rhs):
    return Expression(lhs, '%', rhs)
```

We can use it in a SELECT query to obtain list of records in Contacts table with even id.

```
from peewee import *
db = SqliteDatabase('mydatabase.db')

class BaseModel(Model):
    class Meta:
        database = db

class Contacts(BaseModel):
    RollNo = IntegerField()
    Name = TextField()
    City = TextField()

db.create_tables([Contacts])

from peewee import Expression # the building block for expressions

def mod(lhs, rhs):
    return Expression(lhs,'%', rhs)
qry=Contacts.select().where (mod(Contacts.id,2)==0)
print (qry.sql())
```

```
for q in qry:
    print (q.id, q.Name, q.City)
```

This code will emit following SQL query represented by the string:

```
('SELECT "t1"."id", "t1"."RollNo", "t1"."Name", "t1"."City" FROM "contacts" AS
"t1" WHERE (("t1"."id" % ?) = ?)', [2, 0])
```

Therefore, the output is as follows:

```
2 Amar Delhi
4 Leena Nasik
6 Hema Nagpur
8 John Delhi
10 Raja Nasik
```

24. Peewee — Atomic Transactions

Peewee's database class has **atomic()** method that creates a context manager. It starts a new transaction. Inside the context block, it is possible to commit or rollback the transaction depending upon whether it has been successfully done or it encountered exception.

```
with db.atomic() as transaction:
    try:
        User.create(name='Amar', age=20)
        transaction.commit()
    except DatabaseError:
        transaction.rollback()
```

The atomic() can also be used as decorator.

```
@db.atomic()
def create_user(nm,n):
    return User.create(name=nm, age=n)

create_user('Amar', 20)
```

More than one atomic transaction blocks can also be nested.

```
with db.atomic() as txn1:
    User.create('name'='Amar', age=20)

    with db.atomic() as txn2:
        User.get(name='Amar')
```

25. Peewee — Database Errors

Python’s DB-API standard (recommended by PEP 249) specifies the types of Exception classes to be defined by any DB-API compliant module (such as pymysql, pyscopg2, etc.).

Peewee API provides easy-to-use wrappers for these exceptions. **PeeweeException** is the base classes from which following Exception classes has been defined in Peewee API:

- DatabaseError
- DataError
- IntegrityError
- InterfaceError
- InternalError
- NotSupportedError
- OperationalError
- ProgrammingError

Instead of DB-API specific exceptions to be tried, we can implement above ones from Peewee.

26. Peewee — Query Builder

Peewee also provides a non-ORM API to access the databases. Instead of defining models and fields, we can bind the database tables and columns to **Table** and **Column** objects defined in Peewee and execute queries with their help.

To begin with, declare a Table object corresponding to the one in our database. You have to specify table name and list of columns. Optionally, a primary key can also be provided.

```
Contacts=Table('Contacts', ('id', 'RollNo', 'Name', 'City'))
```

This table object is bound with the database with **bind()** method.

```
Contacts=Contacts.bind(db)
```

Now, we can set up a SELECT query on this table object with select() method and iterate over the resultset as follows:

```
names=Contacts.select()
for name in names:
    print (name)
```

The rows are by default returned as dictionaries.

```
{'id': 1, 'RollNo': 101, 'Name': 'Anil', 'City': 'Mumbai'}
{'id': 2, 'RollNo': 102, 'Name': 'Amar', 'City': 'Delhi'}
{'id': 3, 'RollNo': 103, 'Name': 'Raam', 'City': 'Indore'}
{'id': 4, 'RollNo': 104, 'Name': 'Leena', 'City': 'Nasik'}
{'id': 5, 'RollNo': 105, 'Name': 'Keshav', 'City': 'Pune'}
{'id': 6, 'RollNo': 106, 'Name': 'Hema', 'City': 'Nagpur'}
{'id': 7, 'RollNo': 107, 'Name': 'Beena', 'City': 'Chennai'}
{'id': 8, 'RollNo': 108, 'Name': 'John', 'City': 'Delhi'}
{'id': 9, 'RollNo': 109, 'Name': 'Jaya', 'City': 'Nasik'}
{'id': 10, 'RollNo': 110, 'Name': 'Raja', 'City': 'Nasik'}
```

If needed, they can be obtained as tuples, namedtuples or objects.

Tuples

The program is as follows:

```
names=Contacts.select().tuples()
for name in names:
    print (name)
```

The output is given below:

```
(1, 101, 'Anil', 'Mumbai')
(2, 102, 'Amar', 'Delhi')
(3, 103, 'Raam', 'Indore')
(4, 104, 'Leena', 'Nasik')
(5, 105, 'Keshav', 'Pune')
(6, 106, 'Hema', 'Nagpur')
(7, 107, 'Beena', 'Chennai')
(8, 108, 'John', 'Delhi')
(9, 109, 'Jaya', 'Nasik')
(10, 110, 'Raja', 'Nasik')
```

Namedtuples

The program is stated below:

```
names=Contacts.select().namedtuples()
for name in names:
    print (name)
```

The output is given below:

```
Row(id=1, RollNo=101, Name='Anil', City='Mumbai')
Row(id=2, RollNo=102, Name='Amar', City='Delhi')
Row(id=3, RollNo=103, Name='Raam', City='Indore')
Row(id=4, RollNo=104, Name='Leena', City='Nasik')
Row(id=5, RollNo=105, Name='Keshav', City='Pune')
Row(id=6, RollNo=106, Name='Hema', City='Nagpur')
Row(id=7, RollNo=107, Name='Beena', City='Chennai')
Row(id=8, RollNo=108, Name='John', City='Delhi')
Row(id=9, RollNo=109, Name='Jaya', City='Nasik')
Row(id=10, RollNo=110, Name='Raja', City='Nasik')
```

To insert a new record, INSERT query is constructed as follows:

```
id = Contacts.insert(RollNo=111, Name='Abdul', City='Surat').execute()
```

If a list of records to be added is stored either as a list of dictionaries or as list of tuples, they can be added in bulk.

```
Records=[{'RollNo':112, 'Name':'Ajay', 'City':'Mysore'}, {'RollNo':113,
'Name':'Majid','City':'Delhi'}}


Or


Records=[(112, 'Ajay','Mysore'), (113, 'Majid', 'Delhi')}
```

The INSERT query is written as follows:

```
Contacts.insert(Records).execute()
```

The Peewee Table object has **update()** method to implement SQL UPDATE query. To change City for all records from Nasik to Nagar, we use following query.

```
Contacts.update(City='Nagar').where((Contacts.City=='Nasik')).execute()
```

Finally, Table class in Peewee also has **delete()** method to implement DELETE query in SQL.

```
Contacts.delete().where(Contacts.Name=='Abdul').execute()
```

27. Peewee — Integration with Web Frameworks

Peewee can work seamlessly with most of the Python web framework APIs. Whenever the Web Server Gateway Interface (WSGI) server receives a connection request from client, the connection with database is established, and then the connection is closed upon delivering a response.

While using in a **Flask** based web application, connection has an effect on **@app.before_request** decorator and is disconnected on **@app.teardown_request**.

```
from flask import Flask
from peewee import *

db = SqliteDatabase('mydatabase.db')
app = Flask(__name__)

@app.before_request
def _db_connect():
    db.connect()

@app.teardown_request
def _db_close(exc):
    if not db.is_closed():
        db.close()
```

Peewee API can also be used in **Django**. To do so, add a middleware in Django app.

```
def PeeweeConnectionMiddleware(get_response):
    def middleware(request):
        db.connect()
        try:
            response = get_response(request)
        finally:
            if not db.is_closed():
                db.close()
        return response
    return middleware
```

Middleware is added in Django's settings module.

```
# settings.py
MIDDLEWARE_CLASSES = (
    # Our custom middleware appears first in the list.
    'my_blog.middleware.PeeweeConnectionMiddleware',
     #followed by default middleware list.
     ..
     )
```

Peewee can be comfortably used with other frameworks such as Bottle, Pyramid and Tornado, etc.

28. Peewee — SQLite Extensions

Peewee comes with a Playhouse namespace. It is a collection of various extension modules. One of them is a **playhouse.sqlite_ext** module. It mainly defines **SqliteExtDatabase** class which inherits SqliteDatabase class, supports following additional features:

Features of SQLite Extensions

The features of SQLite Extensions which are supported by Peewee are as follows:

- Full-text search.
- JavaScript Object Notation (JSON) extension integration.
- Closure table extension support.
- LSM1 extension support.
- User-defined table functions.
- Support for online backups using backup API: backup_to_file().
- BLOB API support, for efficient binary data storage.

JSON data can be stored, if a special **JSONField** is declared as one of the field attributes.

```
class MyModel(Model):
    json_data = JSONField(json_dumps=my_json_dumps)
```

To activate full-text search, the model can have **DocIdField** to define primary key.

```
class NoteIndex(FTSModel):
    docid = DocIDField()
    content = SearchField()

    class Meta:
        database = db
```

FTSModel is a Subclass of **VirtualModel** which is available at http://docs.peewee-orm.com/en/latest/peewee/sqlite_ext.html#VirtualModel to be used with the FTS3 and FTS4 full-text search extensions. Sqlite will treat all column types as TEXT (although, you can store other data types, Sqlite will treat them as text).

SearchFіeld is a Field-class to be used for columns on models representing full-text search virtual tables.

SqliteDatabase supports AutoField for increasing primary key. However, SqliteExtDatabase supports AutoIncrementField to ensure that primary always increases monotonically, irrespective of row deletions.

SqliteQ module in playhouse namespace (playhouse.sqliteq) defines subclass of SqliteExeDatabase to handle serialised concurrent writes to a SQlite database.

On the other hand, playhouse.apsw module carries support for apsw sqlite driver. Another Python SQLite Wrapper (APSW) is fast and can handle nested transactions, that are managed explicitly by you code.

```
from apsw_ext import *


db = APSWDatabase('testdb')


class BaseModel(Model):
    class Meta:
        database = db


class MyModel(BaseModel):
    field1 = CharField()
    field2 = DateTimeField()
```

29. Peewee — PostgreSQL and MySQL Extensions

Additional PostgreSQL functionality is enabled by helpers which are defined in **playhouse.postgres_ext** module. This module defines **PostgresqlExtDatabase** class and provides the following additional field types to be exclusively used for declaration of model to be mapped against PostgreSQL database table.

Features of PostgreSQL Extensions

The features of PostgreSQL Extensions which are supported by Peewee are as follows:

- ArrayField field type, for storing arrays.
- HStoreField field type, for storing key/value pairs.
- IntervalField field type, for storing timedelta objects.
- JSONField field type, for storing JSON data.
- BinaryJSONField field type for the jsonb JSON data type.
- TSVectorField field type, for storing full-text search data.
- DateTimeTZField field type, a timezone-aware datetime field.

Additional Postgres-specific features in this module are meant to provide

- hstore support.
- server-side cursors.
- full-text search.

Postgres **hstore** is a key:value store that can be embedded in a table as one of the fields of type **HStoreField**. To enable hstore support, create database instance with **register_hstore=True** parameter.

```
db = PostgresqlExtDatabase('mydatabase', register_hstore=True)
```

Define a model with one **HStoreField**.

```
class Vehicles(BaseExtModel):

    type = CharField()

    features = HStoreField()
```

Create a model instance as follows:

```
v=Vechicle.create(type='Car', specs:{'mfg':'Maruti', 'Fuel':'Petrol',
'model':'Alto'})
```

To access hstore values:

```
obj=Vehicle.get(Vehicle.id=v.id)
```

```
print (obj.features)
```

MySQL Extensions

Alternate implementation of MysqlDatabase class is provided by **MySQLConnectorDatabase** defined in **playhouse.mysql_ext** module. It uses Python's DB-API compatible official **mysql/python connector**.

```
from playhouse.mysql_ext import MySQLConnectorDatabase


db = MySQLConnectorDatabase('mydatabase', host='localhost', user='root',
password='')
```

30. Peewee — Using CockroachDB

CockroachDB or Cockroach Database (CRDB) is developed by computer software company **Cockroach Labs**. It is a scalable, consistently-replicated, transactional datastore which is designed to store copies of data in multiple locations in order to deliver speedy access.

Peewee provides support to this database by way of **CockroachDatabase** class defined in **playhouse.cockroachdb** extension module. The module contains definition of CockroachDatabase as subclass of PostgresqlDatabase class from the core module.

Moreover, there is **run_transaction()** method which runs a function inside a transaction and provides automatic client-side retry logic.

Field Classes

The extension also has certain special field classes that are used as attribute in CRDB compatible model.

- UUIDKeyField - A primary-key field that uses CRDB's UUID type with a default randomly-generated UUID.
- RowIDField - A primary-key field that uses CRDB's INT type with a default unique_rowid().
- JSONField - Same as the Postgres BinaryJSONField.
- ArrayField - Same as the Postgres extension, but does not support multi-dimensional arrays.