IIC2413 – Bases de Datos

Control Bonus
Programación

Entrega

El plazo para la entrega es Domingo 7 de Noviembre, hasta las 23:59. La entrega debe hacerse subiendo un .zip a canvas con todo lo necesario para ejecutar su control incluyendo un README.md donde se explique como hacerlo.

Introducción

En este control vamos a implementar una versión básica de un ORM usando `Python` y la librería `sqlite3`. En particular vamos a implementar las siguientes funcionalidades: poder soportar la creación de un esquema desde 0 a partir de una descripción de este escrita en `Python`, una clase padre `Model` que permita crear clases hijas representativas de las tablas del esquema, mediante las cuales se podrá hacer consultas simples a la tabla además de poder crear, actualizar y eliminar los datos. Finalmente queremos soportar que el usuario pueda juntar operaciones a la DB en una transacción.

1. Definición e instanciación del esquema

En primer lugar el esquema de la base de datos que esté debajo del ORM, será definido mediante un archivo llamado `schema.py` que contendrá un simple diccionario llamado `schema` con el siguiente formato:

```
schema = {
    "table_name": {
        "column_name": "column_type"
    },
}
```

Donde `"table_name"` es el nombre de la tabla en snake case singular, `"column_name"` es el nombre de una columna de la tabla, también en snake case y `"column_type"` es el tipo de dato de la columna: puede ser `"int"` para tipo `INT`, y `"string"` para un `VARCHAR(255)` (solo soportaremos esos dos tipos). Por ejemplo el siguiente esquema:

- `Convencional(id, nombre, id_distrito, afiliacion, edad )`
- `Distrito(id, nombre)`

Tendría el siguiente `schema.py`:

```
schema = {
    "convencional": {
        "nombre": "string",
        "id_distrito": "int",
        "afiliacion": "string",
        "edad": "int",
    },
```

```
    "distrito": {
        "nombre": "string",
    }
}
```

Notar que el archivo no describe el campo `"id"` ya que el ORM debe agregar por defecto un campo `id` de tipo `INT` que se genere de manera automática autoincremental a cada tabla y sea llave primaria. Como primera parte del control debes implementar un script `db_init.py` que a partir de un archivo `schema.py`, genere una base de datos sqlite con las tablas descritas en el archivo.

2. class Model

La componente principal del ORM es la clase `Model` que tu deberás implementar. A través de herencia (o algún otro método similar) `Model` debe hacer que sus clases hijas puedan interactuar con las tablas de la DB creada mediante la ejecución de `db_init.py`.

Por ejemplo, si consideramos el esquema del ejemplo de arriba, y agregamos el siguiente código en un archivo `distrito.py` (notar que ahora el nombre esta en PascalCase):

```
from models import Model

class Distrito(Model):

    def print_name(self):
        print(self.nombre)
```

entonces la clase debe exponer el siguiente comportamiento:

```
from distrito import Distrito

Distrito.create(nombre="Distrito 1")
distro = Distrito.first()
distro.print_name()
>> Distrito 1
```

Nota: Recuerda que tu labor es programar la clase `Model` y no casos particulares como distrito o convencional. `Distrito` es un ejemplo que muestra cómo se usaría y cómo se interactuaría con `Model` y un modelo que herede de esa clase, en un caso de uso real.

2.1. Métodos a implementar

En esta sección se describen los métodos de clase e instancia que debe exponer la clase `Model`. No es necesario que ninguno de los métodos maneje errores no mencionados en el enunciado. Todos los métodos que modifiquen la DB deben ejecutarse en la db inmmediatamente (es decir haciendo `conection.commit()`). Hacer `Connection.isolation_level = None` como se explica acá probablemente simplifique las cosas.

Hint: Todos los métodos son versiones más simples de los que expone el ORM ActiveRecord.

2.1.1. `Model.create(**kwargs)`

Este método permite crear una nueva fila en la tabla. Recibe como parámetros los kwargs con los valores de cada atributo a actualizar.

2.1.2. `Model.update(id: Int, **kwargs)`

Este método hace un update de la fila de id `id`. Recibe como parámetros los kwargs con los valores de cada atributo a actualizar.

2.1.3. `model.update(**kwargs)`

Lo mismo que arriba pero el id se obtiene de la instancia desde la cual se llama al método.

2.1.4. `Model.delete(id: Int, **kwargs)`

Este método hace un delete de la fila de id `id`.

2.1.5. `model.delete(**kwargs)`

Lo mismo que arriba pero el id se obtiene de la instancia desde la cual se llama al método.

2.1.6. `Model.all()`

Retorna una lista de instancias de la clase donde cada instancia corresponde con una fila de la tabla.

2.1.7. `Model.first()`

Retorna una instancia de la clase que corresponde a la primera fila de la tabla según `id`

2.1.8. `Model.find(id: Int)`

Retorna una instancia de la clase que corresponde a la fila de la tabla de id `id`. Si es que no encuentra nada debe retornar `None`

2.1.9. `Model.select(columns: List)`

Igual que `Model.all` pero solo extrae de la db las columnas especificadas en `columns`.

3. Transacciones

Queremos que los usuarios de nuestro ORM puedan generar transacciones de la base de datos en su código. Para eso tienes que implementar el método `.transaction` que inicia y cierra una transacción en la db sqlite. Así:

```
from models import Model
from distrito import Distrito

with Model.transaction():
    Distrito.create(nombre="Distrito 1")
    Distrito.create(nombre="Distrito 2")
    Distrito.delete(1)
```

Lo que haría que las 3 operaciones se hagan en una sola transacción.