15 Python y Bases de datos

Uso de bases de datos y SQL

¿Qué es una base de datos?

Una **base de datos** es un archivo que está organizado para almacenar datos. La mayoría de las bases de datos están organizadas como un diccionario en el sentido de que se asignan de claves a valores. La mayor diferencia es que la base de datos está en el disco (u otro almacenamiento permanente), por lo que persiste después de que finalice el programa. Debido a que una base de datos se almacena en un almacenamiento permanente, puede almacenar muchos más datos que un diccionario, que está limitado al tamaño de la memoria en el ordenador.

Al igual que un diccionario, el software de base de datos está diseñado para mantener la inserción y el acceso de datos muy rápidos, incluso para grandes cantidades de datos. El software de la base de datos mantiene su rendimiento mediante la creación de índices a medida que se agregan datos para permitir que el ordenador salte rápidamente a una entrada en particular.

Hay muchos sistemas de bases de datos diferentes que se utilizan para una amplia variedad de propósitos, incluyendo: Oracle, MySQL, Microsoft SQL Server, PostgreSQL y SQLite. Nos centramos en SQLite en este libro porque es una base de datos muy común y ya está integrada en Python. SQLite está diseñado para ser **integrada** en otras aplicaciones para proporcionar soporte de base de datos dentro de la aplicación. Por ejemplo, el navegador Firefox también usa la base de datos SQLite internamente al igual que muchos otros productos.

http://sqlite.org/

SQLite se adapta bien a algunos de los problemas de manipulación de datos que vemos en Informática, como la aplicación de rastreo de Twitter que describimos en este capítulo.

Conceptos de base de datos

Cuando miras por primera vez una base de datos, parece una hoja de cálculo con varias hojas. Las estructuras de datos principales en una base de datos son: **tablas**, **filas** y **columnas**.

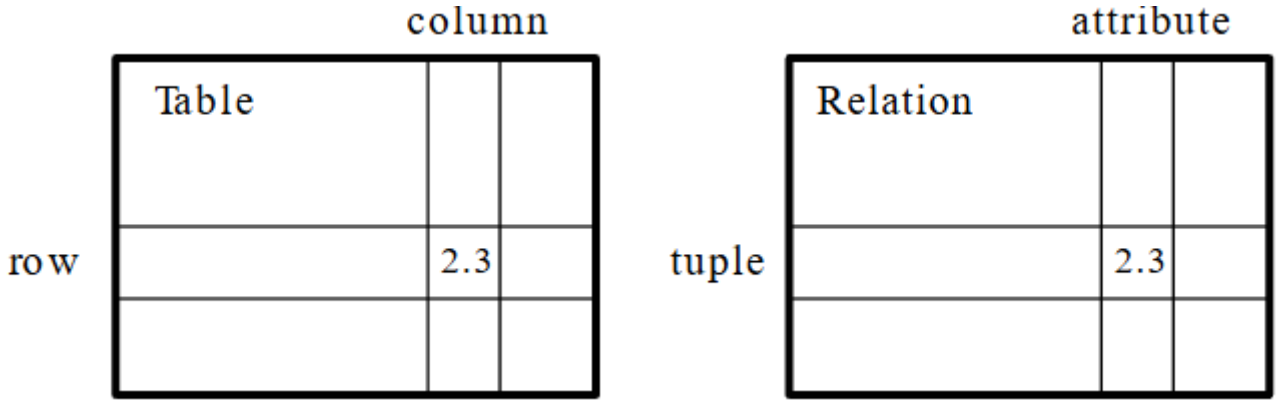

Imagen - Bases de datos relacionales

En las descripciones técnicas de las bases de datos relacionales, los conceptos de tabla, fila y columna se denominan más formalmente **relación**, **tupla** y **atributo**, respectivamente. Usaremos los términos menos formales en este capítulo.

Database Browser for SQLite

Si bien este capítulo se centrará en el uso de Python para trabajar con datos en archivos de base de datos SQLite, muchas operaciones se pueden realizar de manera más conveniente utilizando el software denominado **Database Browser for SQLite**, que está disponible gratuitamente en:

http://sqlitebrowser.org/

Con el navegador puedes crear fácilmente tablas, insertar datos, editar datos o ejecutar consultas SQL simples.

En cierto sentido, el navegador de la base de datos es similar a un editor de texto cuando se trabaja con archivos de texto. Cuando desees realizar una o muy pocas operaciones en un archivo de texto, puedes abrirlo en un editor de texto y realizar los cambios que desees. Cuando tienes muchos cambios que debes hacer en un archivo de texto, a menudo escribirás un programa Python simple. Encontrarás el mismo patrón cuando trabajes con bases de datos. Harás operaciones simples en el administrador de bases de datos y las operaciones más complejas se realizarán de manera más conveniente en Python.

Creando una tabla de base de datos

Las bases de datos requieren una estructura más definida que las listas o los diccionarios de Python [1].

Cuando creamos una tabla en la base de datos debemos decir a la base de datos por adelantado los nombres de cada una de las **columnas** en la tabla y el tipo de datos que planeamos almacenar en cada cada una de ellas. Cuando el software de la base de datos conoce el tipo de datos en cada columna, puedes elegir la forma más eficiente de almacenar y buscar los datos según el tipo de datos.

Puedes ver los diversos tipos de datos admitidos por SQLite en la siguiente url:

http://www.sqlite.org/datatypes.html

Definir la estructura de tus datos por adelantado puede parecer inconveniente al principio, pero la recompensa es un acceso rápido a tus datos, incluso cuando la base de datos contenga una gran cantidad de datos.

El código para crear un archivo de base de datos y una tabla llamada `Tracks` con dos columnas en la base de datos es la siguiente:

https://trinket.io/embed/python3/09d1152020

La operación `connect` hace una "conexión" a la base de datos almacenada en el archivo `music.sqlite3` en el directorio actual. Si el archivo no existe, será creado. La razón por la que esto se denomina "conexión" es que a veces la base de datos se almacena en un "servidor de base de datos" independiente del servidor en el que ejecutamos nuestra aplicación. En nuestros ejemplos simples, la base de datos solo será un archivo local en el mismo directorio que el código Python que estamos ejecutando.

Un **cursor** es como un identificador de archivo que podemos usar para realizar operaciones en los datos almacenados en la base de datos. Llamar a `cursor()` es muy similar conceptualmente a llamar a `open()` cuando se trata de archivos de texto.

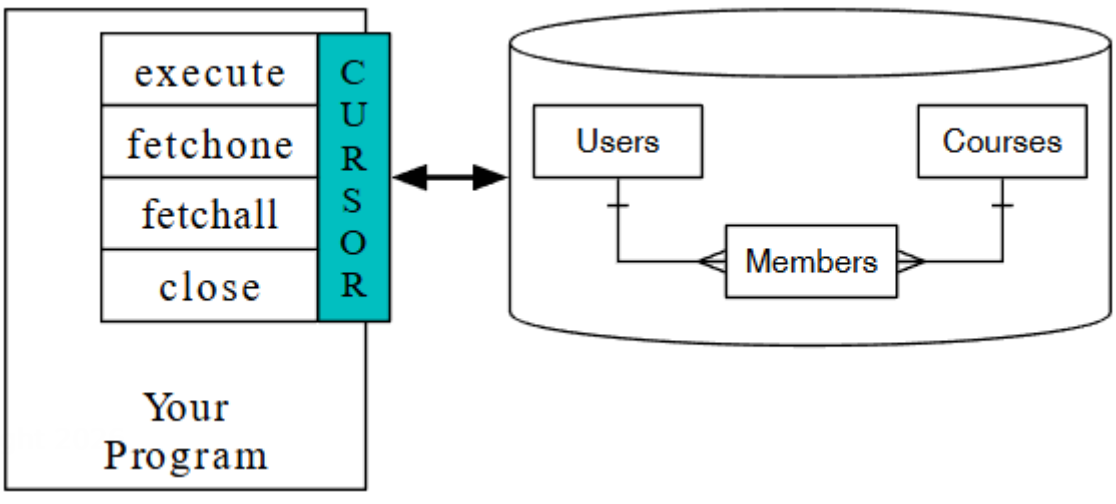

Imagen - A Database Cursor

Una vez que tengamos el cursor, podemos comenzar a ejecutar comandos en el contenido de la base de datos usando el método `execute()`.

Los comandos de la base de datos se expresan en un lenguaje especial que se ha estandarizado en muchos proveedores de bases de datos diferentes para permitirnos aprender un solo idioma de base de datos. El lenguaje de la base de datos se llama **Lenguaje de consulta estructurada** o **SQL** (Structured Query Language) para abreviar.

http://en.wikipedia.org/wiki/SQL

En nuestro ejemplo, estamos ejecutando dos comandos SQL en nuestra base de datos. Como convención, mostraremos las palabras clave de SQL en mayúsculas y las partes del comando que estamos agregando (como los nombres de tablas y columnas) se mostrarán en minúsculas.

El primer comando SQL elimina la tabla `Tracks` de la base de datos si existe. Este patrón es simplemente para permitirnos ejecutar el mismo programa para crear la tabla `Tracks` una y otra vez sin causar un error. Tenga en cuenta que el comando `DROP TABLE` elimina la tabla y todo su contenido de la base de datos (es decir, no hay "deshacer").

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

El segundo comando crea una tabla llamada `Tracks` con una columna de texto llamada `title` y una columna entera llamada `plays`.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

Ahora que hemos creado una tabla llamada `Tracks`, podemos poner algunos datos en esa tabla usando la operación SQL `INSERT`. Nuevamente, comenzamos haciendo una conexión a la base de datos y obteniendo el `cursor`. Luego podemos ejecutar comandos SQL usando el cursor.

El comando SQL `INSERT` indica qué tabla estamos usando y luego define una nueva fila al enumerar los campos que queremos incluir `(título, jugadas)` seguido de los `VALORES` que queremos colocar en la nueva fila. Especificamos los valores como signos de interrogación `(?,?)` Para indicar que los valores reales se pasan como una tupla `('My Way', 15)` como el segundo parámetro a la llamada `execute()`.

https://trinket.io/embed/python3/e665c26f67

Primero, insertamos dos filas en nuestra tabla y usamos `commit()` para forzar que los datos se escriban en el archivo de la base de datos.

Tracks

title	plays
Thunderstruck	20
My Way	15

Imagen - Filas en una tabla

Luego usamos el comando `SELECT` para recuperar las filas que acabamos de insertar de la tabla. En el comando `SELECT`, indicamos en qué columnas nos gustaría extraer, `(título, jugamos)`, e indicamos de qué tabla queremos recuperar los datos. Después de ejecutar la instrucción `SELECT`, el cursor es algo que podemos recorrer en una instrucción `for`. Por eficiencia, el cursor no lee todos los datos de la base de datos cuando ejecutamos la instrucción `SELECT`. En su lugar, los datos se leen a medida que recorramos las filas en la declaración `for`.

La salida del programa es la siguiente:

```
Tracks:
('Thunderstruck', 20)
('My Way', 15)
```

Nuestro bucle `for` encuentra dos filas, y cada fila es una tupla de Python con el primer valor como `title` y el segundo valor como el número de `plays`.

Nota: Puede ver cadenas que comienzan con `u'` en otros libros o en Internet. Esto era una indicación en Python 2 de que las cadenas son Unicode *cadenas que son capaces de almacenar conjuntos de caracteres no latinos. En Python 3, todas las cadenas son cadenas Unicode por defecto.*

Al final del programa, ejecutamos un comando SQL para 'BORRAR' las filas que acabamos de crear para que podamos ejecutar el programa una y otra vez. El comando `DELETE` muestra el uso de una cláusula `WHERE` que nos permite expresar un criterio de selección para que podamos pedir a la base de datos que aplique el comando solo a las filas que coincidan con el criterio. En este ejemplo, el criterio pasa a aplicarse a todas las filas, de modo que vaciamos la tabla para poder ejecutar el programa repetidamente. Después de que se realiza el `BORRAR`, también llamamos a `commit ()` para forzar que los datos se eliminen de la base de datos.

Resumen de lenguaje de consulta estructurado

Hasta ahora, hemos estado utilizando el lenguaje de consulta estructurado en nuestros ejemplos de Python y hemos cubierto muchos de los conceptos básicos de los comandos SQL. En esta sección, analizamos el lenguaje SQL en particular y ofrecemos una descripción general de la sintaxis de SQL.

Dado que hay tantos proveedores de bases de datos diferentes, el lenguaje de consulta estructurado (SQL) se estandarizó para que pudiéramos comunicarnos facilmente con los sistemas de bases de datos de varios proveedores.

Una base de datos relacional está formada por tablas, filas y columnas. Las columnas generalmente tienen un tipo como datos de texto, numéricos o de fecha. Cuando creamos una tabla, indicamos los nombres y tipos de las columnas:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

Para insertar una fila en una tabla, usamos el comando SQL `INSERT`:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

La declaración `INSERT` especifica el nombre de la tabla, luego una lista de los campos/columnas que te gustaría establecer en la nueva fila, y luego la palabra clave `VALUES` y una lista de valores correspondientes para cada uno de los campos.

El comando SQL `SELECT` se usa para recuperar filas y columnas de una base de datos. La declaración `SELECT` te permite especificar qué columnas deseas recuperar, así como una cláusula `WHERE` para seleccionar qué filas deseas ver. También permite una cláusula opcional `ORDER BY` para controlar la clasificación de las filas devueltas.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

Usar `*` indica que deseas que la base de datos devuelva todas las columnas para cada fila que coincida con la cláusula `WHERE`.

Ten en cuenta que, a diferencia de Python, en una cláusula SQL `WHERE` usamos un único signo igual para indicar una prueba de igualdad en lugar de un doble signo igual. Otras operaciones lógicas permitidas en una cláusula `WHERE` incluyen `<`, `>`, `<=`, `>=`, `!=`, así como `AND` y `OR` y paréntesis

para construir tus expresiones lógicas.

Puedes solicitar que las filas devueltas se ordenen por uno de los campos de la siguiente manera:

```
SELECT title,plays FROM Tracks ORDER BY title
```

Para eliminar una fila, necesitas una cláusula `WHERE` en una sentencia SQL `DELETE`. La cláusula `WHERE` determina qué filas se van a eliminar:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

Es posible `UPDATE` una columna o columnas dentro de una o más filas en una tabla usando la declaración SQL `UPDATE` de la siguiente manera:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

La declaración `UPDATE` especifica una tabla y luego una lista de campos y valores para cambiar después de la palabra clave `SET` y luego una cláusula opcional `WHERE` para seleccionar las filas que se actualizarán. Una sola instrucción `UPDATE` cambiará todas las filas que coincidan con la cláusula `WHERE`. Si no se especificas una cláusula `WHERE`, realiza el `UPDATE` en todas las filas de la tabla.

Estos cuatro comandos SQL básicos (INSERT, SELECT, UPDATE, y DELETE) permiten las cuatro operaciones básicas necesarias para crear y mantener datos.

Spidering Twitter usando una base de datos

En esta sección, crearemos un programa de spidering simple que pasará por las cuentas de Twitter y creará una base de datos de ellas. **Nota: Ten mucho cuidado al ejecutar este programa. No deseas extraer demasiados datos o ejecutar el programa durante demasiado tiempo y terminar con el cierre de tu acceso a Twitter.**

Uno de los problemas de cualquier tipo de programa spidering es que debe poder detenerse y reiniciarse muchas veces y no quieres perder los datos que has recuperado hasta ahora. No siempre deseas reiniciar tu recuperación de datos desde el principio, por lo que queremos almacenar datos a medida que los recuperamos para que nuestro programa pueda iniciarse nuevamente y continuar donde lo dejó.

Comenzaremos por recuperar los amigos de Twitter de una persona y sus estados, recorreremos la lista de amigos y agregaremos a cada uno de los amigos a una base de datos para recuperarlos en el futuro. Después de procesar los amigos de Twitter de una persona, revisamos nuestra base de datos y recuperamos a uno de los amigos del amigo. Hacemos esto una y otra vez, seleccionando a una persona "no visitada", recuperando su lista de amigos y agregando amigos que no hemos visto en nuestra lista para una futura visita.

También hacemos un seguimiento de cuántas veces hemos visto a un amigo en particular en la base de datos para tener una idea de su "popularidad".

Al almacenar nuestra lista de cuentas conocidas y si hemos recuperado la cuenta o no, y cuán popular es la cuenta en una base de datos en el disco de el ordenador, podemos detener y reiniciar nuestro programa tantas veces como lo deseemos.

Este programa es un poco complejo. Se basa en el código del ejercicio anterior en el libro que utiliza la API de Twitter.

Aquí está el código fuente de nuestra aplicación de spidering de Twitter:

https://trinket.io/embed/python3/f9ee36fa1f

Nuestra base de datos se almacena en el archivo `spider.sqlite3` y tiene una tabla llamada `Twitter`. Cada fila en la tabla `Twitter` tiene una columna para el nombre de la cuenta, si hemos recuperado los amigos de esta cuenta y cuántas veces esta cuenta ha sido "seguida".

En el ciclo principal del programa, le pedimos al usuario un nombre de cuenta de Twitter o "salir" para salir del programa. Si el usuario ingresa una cuenta de Twitter, recuperamos la lista de amigos y estados de ese usuario y agregamos a cada amigo a la base de datos si aún no está en la base de datos. Si el amigo ya está en la lista, agregamos 1 al campo `friends` en la fila de la base de datos.

Si el usuario presiona Intro, buscamos en la base de datos la próxima cuenta de Twitter que aún no hemos recuperado, recuperamos los amigos y los estados de esa cuenta, los agregamos a la base de datos o los actualizamos, y aumentamos el recuento de sus "amigos".

Una vez que recuperamos la lista de amigos y estados, recorremos todos los elementos `user` en el JSON devuelto y recuperamos el `screen_name` para cada usuario. Luego usamos la declaración `SELECT` para ver si ya hemos almacenado este `screen_name` en particular en la base de datos y recuperar el recuento de amigos (`friends`) si el registro existe.

```
countnew = 0
countold = 0
for u in js['users'] :
    friend = u['screen_name']
    print friend
    cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
        (friend, ) )
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
            (count+1, friend) )
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
            VALUES ( ?, 0, 1 )''', ( friend, ) )
        countnew = countnew + 1
print 'New accounts=',countnew,' revisited=',countold
conn.commit()
```

Una vez que el cursor ejecuta la instrucción `SELECT`, debemos recuperar las filas. Podríamos hacer esto con una declaración `for`, pero como solo estamos recuperando una fila (`LIMIT 1`), podemos usar el método `fetchone()` para recuperar la primera (y única) fila que es el resultado de la operación `SELECT`. Como `fetchone()` devuelve la fila como **tupla** (aunque solo haya un campo), tomamos el primer valor de la tupla para obtener el recuento actual de amigos y guardarlo en la variable `count`.

Si esta recuperación es exitosa, usamos la declaración SQL `UPDATE` con una cláusula `WHERE` para agregar 1 a la columna `friends` para la fila que coincide con la cuenta del amigo. Observa que hay dos marcadores de posición (es decir, signos de interrogación) en el SQL, y el segundo parámetro de `execute()` es una tupla de dos elementos que contiene los valores que se sustituirán en el SQL en lugar de los signos de interrogación.

Si el código en el bloque `try` falla, es probable que no haya ningún registro que coincida con la cláusula WHERE name =? en la instrucción SELECT. Entonces, en el bloque `except`, usamos la declaración SQL `INSERT` para agregar el `screen_name` del amigo a la tabla con una indicación de que aún no hemos recuperado el `screen_name` y establecer el recuento de amigos en cero.

Entonces, la primera vez que se ejecuta el programa y entramos en una cuenta de Twitter, el programa se ejecuta de la siguiente manera:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

Como esta es la primera vez que ejecutamos el programa, la base de datos está vacía y creamos la base de datos en el archivo `spider.sqlite3` y agregamos una tabla llamada `Twitter` a la base de datos. Luego recuperamos algunos amigos y los agregamos a la base de datos, que está vacía.

En este punto, podríamos querer escribir un dumper de base de datos simple para echar un vistazo a lo que está en nuestro archivo `spider.sqlite3`:

https://trinket.io/embed/python3/e986702fd6

Este programa simplemente abre la base de datos y selecciona todas las columnas de todas las filas en la tabla `Twitter`, luego recorre las filas e imprime cada fila.

Si ejecutamos este programa después de la primera ejecución de nuestra araña de Twitter anterior, su salida será la siguiente:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

Vemos una fila para cada `screen_name`, que no hemos recuperado los datos para ese `screen_name`, y todos en la base de datos tienen un amigo.

Ahora nuestra base de datos refleja la recuperación de los amigos de nuestra primera cuenta de Twitter (**drchuck**). Podemos ejecutar el programa nuevamente y decirle que recupere a los amigos de la próxima cuenta "sin procesar" simplemente presionando Intro en lugar de una cuenta de Twitter de la siguiente manera:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Desde que presionamos enter (es decir, no especificamos una cuenta de Twitter), se ejecuta el siguiente código:

```
if ( len(acct) < 1 ) :
    cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print 'No unretrieved twitter accounts found'
        continue
```

Usamos la instrucción SQL `SELECT` para recuperar el nombre del primer usuario (`LIMIT 1`) que todavía tiene su valor establecido en cero. También utilizamos el patrón `fetchone() [0]` dentro de un bloque try / except para extraer un `screen_name` de los datos recuperados o para mostrar un mensaje de error y realizar un bucle de copia de seguridad.

Si recuperamos con éxito un `screen_name` sin procesar, recuperamos sus datos de la siguiente manera:

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'})
print('Retrieving')
url connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)
cur.execute('UPDATE Twitter SET retrieved = 1 WHERE nombre =?', (acct,))
```

Una vez que recuperamos los datos con éxito, usamos la instrucción `UPDATE` para establecer la columna `retrieved` en 1 para indicar que hemos completado la recuperación de los amigos de esta cuenta. Esto nos impide recuperar los mismos datos una y otra vez y nos mantiene progresando a través de la red de amigos de Twitter.

Si ejecutamos el programa de amigos y presionamos Intro dos veces para recuperar a los amigos de los siguientes amigos no visitados, luego ejecutamos el programa de descarga, nos dará el siguiente resultado:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

Podemos ver que hemos registrado correctamente que hemos visitado `lhawthorn` y `opencontent`. También las cuentas `cnxorg` y `kthanos` ya tienen dos seguidores. Como ahora hemos recuperado los amigos de tres personas (`drchuck`, `opencontent` y `lhawthorn`), nuestra tabla tiene 55 filas de amigos para recuperar.

Cada vez que ejecutamos el programa y presionamos enter, elegimos la siguiente cuenta no visitada (por ejemplo, la siguiente cuenta será `steve_coppin`), recuperará a sus amigos, los marcará como recuperados y, para cada uno de los amigos de `steve_coppin`, agregará hasta el final de la base de datos o actualizará el recuento de sus amigos si ya están en la base de datos.

Dado que todos los datos del programa se almacenan en el disco en una base de datos, la actividad de rastreo se puede suspender y reanudar tantas veces como desee sin perder datos.

Modelado básico de datos

El poder real de una base de datos relacional es cuando creamos varias tablas y hacemos enlaces entre esas tablas. El acto de decidir cómo dividir los datos de su aplicación en varias tablas y establecer las relaciones entre las tablas se denomina **modelado de datos**. El documento de diseño que muestra las tablas y sus relaciones se denomina **modelo de datos**.

El modelado de datos es una habilidad relativamente sofisticada y solo presentaremos los conceptos más básicos del modelado de datos relacionales en esta sección. Para más detalles sobre el modelado de datos, puedes comenzar con:

http://en.wikipedia.org/wiki/Relational_model

Digamos que para nuestra aplicación de araña de Twitter, en lugar de simplemente contar los amigos de una persona, queríamos mantener una lista de todas las relaciones entrantes para poder encontrar una lista de todas las personas que siguen una cuenta en particular.

Dado que potencialmente todos tendrán muchas cuentas que los siguen, no podemos simplemente agregar una sola columna a nuestra tabla `Twitter`. Así que creamos una nueva tabla que hace un seguimiento de parejas de amigos. La siguiente es una forma simple de hacer una tabla de este tipo:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Cada vez que nos encontramos con una persona que está siguiendo `drchuck`, insertamos una fila del formulario:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

Como estamos procesando a los 20 amigos del feed de Twitter `drchuck`, insertaremos 20 registros con "drchuck" como primer parámetro, por lo que terminaremos duplicando la cadena muchas veces en la base de datos.

Esta duplicación de datos de cadena viola una de las mejores prácticas para **la normalización de la base de datos** que básicamente dice que nunca debemos colocar los mismos datos de cadena en la base de datos más de una vez. Si necesitamos los datos más de una vez, creamos una clave numérica para los datos y hacemos referencia a los datos reales utilizando esta clave.

En términos prácticos, una cadena ocupa mucho más espacio que un entero en el disco y en la memoria de nuestra ordenador, y toma más tiempo de procesador para comparar y ordenar. Si solo tenemos unos pocos cientos de entradas, el tiempo de almacenamiento y procesador apenas importa. Pero si tenemos un millón de personas en nuestra base de datos y una posibilidad de 100 millones de enlaces de amigos, es importante poder escanear los datos lo más rápido posible.

Almacenaremos nuestras cuentas de Twitter en una tabla llamada `People` en lugar de la tabla `Twitter` utilizada en el ejemplo anterior. La tabla `People` tiene una columna adicional para almacenar la clave numérica asociada a la fila para este usuario de Twitter. SQLite tiene una función que agrega automáticamente el valor clave para cualquier fila que insertamos en una tabla

usando un tipo especial de columna de datos (`INTEGER PRIMARY KEY`).

Podemos crear la tabla `People` con esta columna adicional `id` de la siguiente manera:

```
CREATE TABLE People (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

Ten en cuenta que ya no estamos manteniendo un recuento de amigos en cada fila de la tabla "People". Cuando seleccionamos `INTEGER PRIMARY KEY` como el tipo de nuestra columna `id`, estamos indicando que nos gustaría que SQLite administre esta columna y asigne una clave numérica única a cada fila que insertamos automáticamente. También agregamos la palabra clave `UNIQUE` para indicar que no permitiremos que SQLite inserte dos filas con el mismo valor para `name` .

Ahora, en lugar de crear la tabla `Pals` anterior, creamos una tabla llamada `Follows` con dos columnas enteras `from_id` y `to_id` y una restricción en la tabla que la combinación de `from_id` y `to_id` debe ser única en esta tabla (es decir, no podemos insertar filas duplicadas) en nuestra base de datos.

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

Cuando agregamos cláusulas `UNIQUE` a nuestras tablas, estamos comunicando un conjunto de reglas que le estamos pidiendo a la base de datos que aplique cuando intentemos insertar registros. Las reglas nos impiden cometer errores y simplifican la escritura de algunos de nuestros códigos.

En esencia, al crear esta tabla "Follows", estamos modelando una "relación" donde una persona "sigue" a otra persona y la representamos con un par de números que indican que (a) las personas están conectadas y (b) la dirección de la relación.

Relaciones entre tablas
Imagen - Relaciones entre tablas

Programación con varias tablas

Ahora reharemos el programa de arañas de Twitter utilizando dos tablas, las claves principales y las referencias clave, como se describe anteriormente. Aquí está el código para la nueva versión del programa:

https://trinket.io/embed/python3/88efa27a18

Este programa está empezando a complicarse un poco, pero ilustra los patrones que debemos usar cuando estamos usando claves de enteros para vincular tablas. Los patrones básicos son:

1. Crear tablas con claves primarias y restricciones.
2. Cuando tenemos una clave lógica para una persona (es decir, el nombre de la cuenta) y necesitamos el valor `id` para la persona, dependiendo de si la persona ya está en la tabla `People` o bien debemos: (1) buscar la persona en la tabla `People` y recuperar el valor `id` para la persona o (2) agrear la persona a la tabla `People` y obtener el valor `id` para la fila recién agregada.
3. Inserta la fila que captura la relación "follows".

Desarrollaremos brevemente estos puntos.

Restricciones en tablas de bases de datos

A medida que diseñamos nuestras estructuras de tablas, podemos decirle al sistema de base de datos que nos gustaría aplicar algunas reglas. Estas reglas nos ayudan a cometer errores e introducir datos incorrectos en las tablas. Cuando creamos nuestras tablas:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

Indicamos que la columna `name` en la tabla `People` debe ser `UNIQUE`. También indicamos que la combinación de los dos números en cada fila de la tabla `Follows` debe ser única. Estas restricciones nos impiden cometer errores, como agregar la misma relación más de una vez.

Podemos aprovechar estas restricciones en el siguiente código:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

Añadimos la cláusula `OR IGNORE` a nuestra declaración `INSERT` para indicar que si este `INSERT` en particular causaría una violación de la regla `name` *debe ser único*, el sistema de la base de datos puede ignorar el `INSERT` . Estamos utilizando la restricción de la base de datos como una red de seguridad para asegurarnos de no hacer algo incorrecto sin querer.

De manera similar, el siguiente código asegura que no agregamos la misma relación exacta de `Follows` dos veces.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

Una vez más, simplemente le pedimos a la base de datos que ignore nuestro intento de `INSERT` si viola la restricción de unicidad que especificamos para las filas de `Follows`.

Recuperar y/o insertar un registro

Cuando solicitamos al usuario una cuenta de Twitter, si la cuenta existe, debemos buscar su valor 'id'. Si la cuenta aún no existe en la tabla `People`, debemos insertar el registro y obtener el valor `id` de la fila insertada.

Este es un patrón muy común y se realiza dos veces en el programa anterior. Este código muestra cómo buscamos el `id` para la cuenta de un amigo cuando hemos extraído un `screen_name` de un nodo `usuario` en el JSON de Twitter recuperado.

Dado que con el tiempo será cada vez más probable que la cuenta ya esté en la base de datos, primero verificamos si existe el registro 'Personas' usando una declaración `SELECT`.

Si todo va bien [2] dentro de la sección `try`, recuperamos el registro usando `fetchone()` y luego recuperamos el primer (y único) elemento de la tupla devuelta y lo guardamos en `friend_id`.

Si el `SELECT` falla, el código `fetchone()[0]` fallará y el control se transferirá a la sección `except`.

```
    friend = u['screen_name']
    cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
        (friend, ) )
    try:
        friend_id = cur.fetchone()[0]
        countold = countold + 1
    except:
        cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
            VALUES ( ?, 0)''', ( friend, ) )
        conn.commit()
        if cur.rowcount != 1 :
            print 'Error inserting account:',friend
```

```
            continue
        friend_id = cur.lastrowid
        countnew = countnew + 1
```

Si terminamos en el código `except`, simplemente significa que no se encontró la fila, por lo que debemos insertar la fila. Usamos `INSERT OR IGNORE` solo para evitar errores y luego llamamos a `commit ()` para forzar que la base de datos se actualice realmente. Una vez que se realiza la escritura, podemos verificar el `cur.rowcount` para ver cuántas filas se vieron afectadas. Ya que estamos intentando insertar una sola fila, si el número de filas afectadas es diferente a 1, es un error.

Si el `INSERT` tiene éxito, podemos ver `cur.lastrowid` para averiguar qué valor asignó la base de datos a la columna `id` en nuestra fila recién creada.

Guardando la relación de amistad

Una vez que sepamos el valor clave tanto para el usuario de Twitter como para el amigo en el JSON, es muy sencillo insertar los dos números en la tabla `Siguiendo 'con el siguiente código:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id) )
```

Ten en cuenta que dejamos que la base de datos se encargue de evitar que "insertemos dos veces" una relación creando la tabla con una restricción de unicidad y luego agregando `OR IGNORE` a nuestra declaración `INSERT`.

Aquí hay una ejecución de muestra de este programa:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
```

```
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Comenzamos con la cuenta `drchuck` y luego dejamos que el programa seleccione automáticamente las siguientes dos cuentas para recuperar y agregar a nuestra base de datos.

A continuación se muestran las primeras filas de las tablas `People` y `Follows` después de completar esta ejecución:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

Puedes ver los campos `id`, `name` y `visited` en la tabla `People` y puedes ver los números de ambos extremos de la relación en la tabla `Follows`. En la tabla `People`, podemos ver que las tres primeras personas han sido visitadas y sus datos han sido recuperados. Los datos en la tabla `Follows` indican que `drchuck` (usuario 1) es un amigo para todas las personas que aparecen en las primeras cinco filas. Esto tiene sentido porque los primeros datos que recuperamos y almacenamos fueron los amigos de Twitter de `drchuck`. Si tuvieras que imprimir más filas de la tabla `Follows`, también verías a los amigos de los usuarios 2 y 3.

Tres tipos de teclas

TODO sigue por aquí Ahora que hemos empezado a construir un modelo que coloca nuestros datos en varias tablas vinculadas y vincula las filas en esas tablas usando **claves**, debemos analizar la terminología relacionada con las claves. En general, hay tres tipos de claves utilizadas en un modelo de base de datos.

- Una **clave lógica** es una clave que el "mundo real" puede usar para buscar una fila. En nuestro modelo de datos de ejemplo, el campo `nombre` es una clave lógica. Es el nombre de pantalla para el usuario y, de hecho, buscamos la fila de un usuario varias veces en el programa usando el campo `nombre`. A menudo verás que tiene sentido agregar una restricción `UNIQUE` a una clave lógica. Dado que la clave lógica es la forma en que buscamos una fila del mundo exterior, no tiene mucho sentido permitir varias filas con el mismo valor en la tabla.
- Una **clave principal** suele ser un número que la base de datos asigna automáticamente. Por lo general, no tiene ningún significado fuera del programa y solo se utiliza para vincular filas de diferentes tablas. Cuando queremos buscar una fila en una tabla, generalmente buscar la fila con la clave principal es la forma más rápida de encontrar la fila. Dado que las claves primarias son números enteros, ocupan muy poco espacio de almacenamiento y se pueden comparar o clasificar muy rápidamente. En nuestro modelo de datos, el campo `id` es un ejemplo de una clave primaria.
- Una **clave externa** es generalmente un número que apunta a la clave principal de una fila asociada en una tabla diferente. Un ejemplo de una clave externa en nuestro modelo de datos es el `from_id`.

Estamos utilizando una convención de nomenclatura para llamar siempre el nombre de campo de la clave principal `id` y añadir el sufijo `_id` a cualquier nombre de campo que sea una clave externa.

Uso de JOIN para recuperar datos

Ahora que hemos seguido las reglas de normalización de la base de datos y hemos separado los datos en dos tablas, vinculadas entre sí mediante claves primarias y externas, debemos poder construir un `SELECT` que vuelva a ensamblar los datos en todas las tablas.

SQL usa la cláusula `JOIN` para volver a conectar estas tablas. En la cláusula `JOIN` usted especifica los campos que se utilizan para volver a conectar las filas entre las tablas.

El siguiente es un ejemplo de un `SELECT` con una cláusula `JOIN`:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

La cláusula `JOIN` indica que los campos que estamos seleccionando cruzan las tablas `Follows` y `People`. La cláusula `ON` indica cómo se deben unir las dos tablas: toma las filas de `Follows` y agrega la fila de `People` donde el campo `from_id` en `Follows` es el mismo que el valor de `id` en el Tabla de personas.

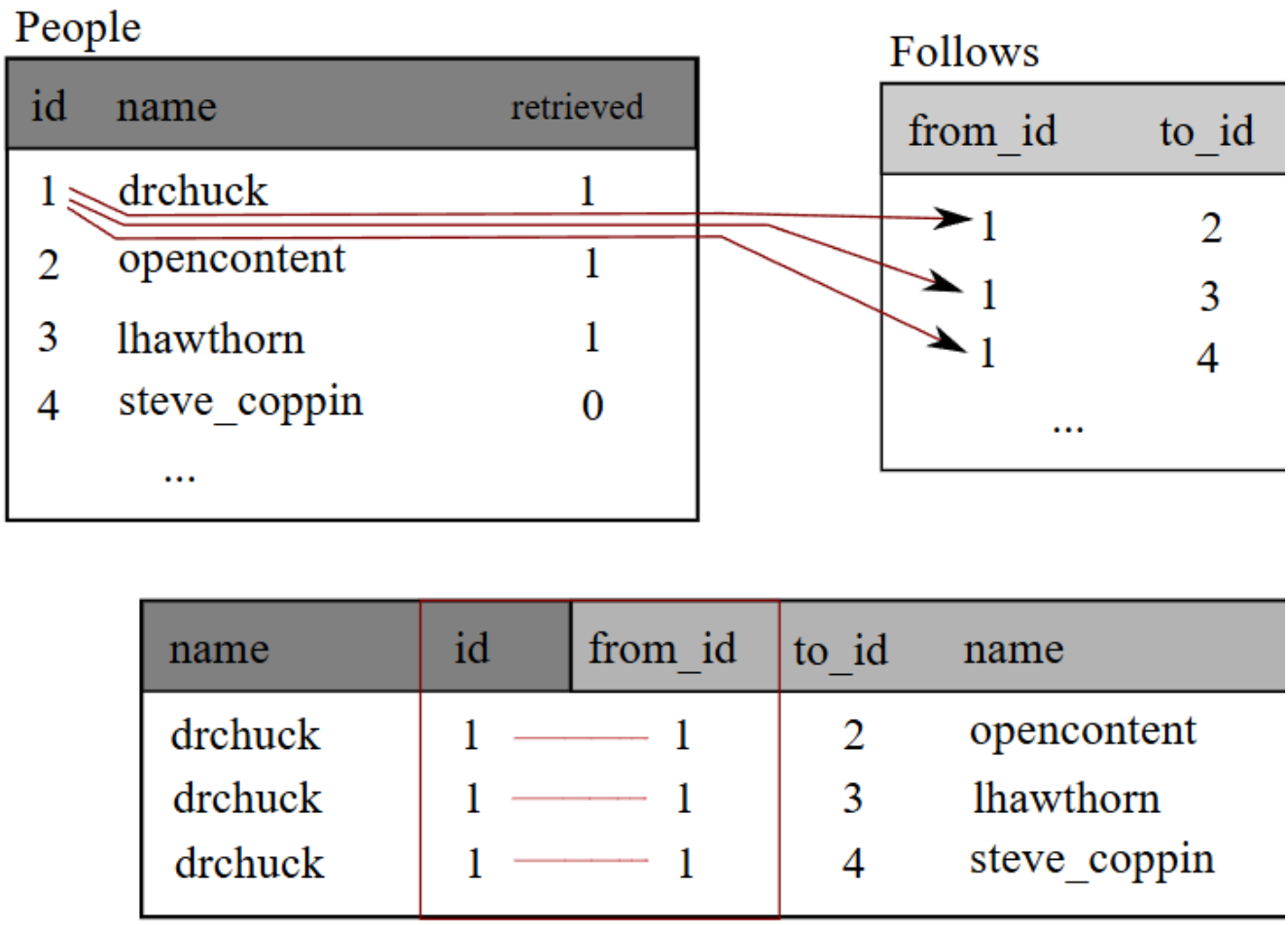

Imagen - Conectando tablas utilizando JOIN

El resultado de JOIN es crear "metarows" extra largos que tienen los campos de "Personas" y los campos correspondientes de los "Follows". Donde hay más de una coincidencia entre el campo `id` de `People` y el `from_id` de `People`, entonces JOIN crea un metarow para cada uno de los pares de filas correspondientes, duplicando los datos según sea necesario.

El siguiente código muestra los datos que tendremos en la base de datos después de que se haya ejecutado varias veces el programa de arañas de Twitter de múltiples tablas (arriba).

https://trinket.io/embed/python3/8f92ad783f

En este programa, primero desechamos `People` y `Follows` y luego volcamos un subconjunto de los datos en las tablas unidas.

Aquí está la salida del programa:

```
python twjoin.py
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```

Verás las columnas de las tablas `People` y `Follows` y el último conjunto de filas es el resultado de `SELECT` con la cláusula `JOIN`.

En la última selección, estamos buscando cuentas que sean amigos de "opencontent" (es decir, `People.id = 2`).

En cada uno de los "metarows" en la última selección, las dos primeras columnas son de la tabla "Follows" seguidas por las columnas tres a cinco de la tabla "People". También puedes ver que la segunda columna (`Follows.to_id`) coincide con la tercera columna (`People.id`) en cada uno de los "metarows" unidos.

Resumen

Este capítulo ha cubierto mucho terreno para ofrecerte una descripción general de los conceptos básicos del uso de una base de datos en Python. Es más complicado escribir el código para usar una base de datos para almacenar datos que los diccionarios de Python o los archivos planos, por lo que hay pocas razones para usar una base de datos a menos que su aplicación realmente necesite las capacidades de una base de datos. Las situaciones en las que una base de datos puede ser bastante útil son: (1) cuando tu aplicación necesita realizar muchas actualizaciones aleatorias dentro de un conjunto de datos grande, (2) cuando sus datos son tan grandes que no caben en un diccionario y necesitas buscar y actualizar la información repetidamente, o (3) cuando tengas un proceso de larga ejecución que desees poder detener, reiniciar y conservar los datos de una ejecución a la siguiente.

Puedes crear una base de datos simple con una sola tabla para satisfacer las necesidades de muchas aplicaciones, pero la mayoría de los problemas requerirán varias tablas y vínculos/relaciones entre filas en diferentes tablas. Cuando comienzas a crear enlaces entre tablas, es importante hacer un diseño cuidadoso y seguir las reglas de normalización de la base de datos para aprovechar al máximo sus capacidades. Dado que la motivación principal para usar una base de datos es que tienes una gran cantidad de datos con los que tratar, es importante modelar tus datos de manera eficiente para que tus programas se ejecuten lo más rápido posible.

depuración

Un patrón común cuando desarrolles un programa en Python para conectarte a una base de datos SQLite será ejecutar un programa y verificar los resultados utilizando el Explorador de bases de datos para SQLite. El navegador te permite verificar rápidamente si tu programa está funcionando correctamente.

Debes tener cuidado porque SQLite se encarga de evitar que dos programas cambien los mismos datos al mismo tiempo. Por ejemplo, si abres una base de datos en el navegador y realizas un cambio en la base de datos y aún no has presionado el botón "guardar" en el navegador, el navegador "bloquea" el archivo de la base de datos y evita que cualquier otro programa acceda al mismo. En particular, tu programa Python no podrá acceder al archivo si está bloqueado.

Por lo tanto, una solución es asegurarte de cerrar el navegador de la base de datos o usar el menú **Archivo** para cerrar la base de datos en el navegador antes de intentar acceder a la base de datos desde Python para evitar el problema.

> “ [1]. SQLite realmente permite cierta flexibilidad en el tipo de datos almacenados en una columna, pero mantendremos nuestros tipos de datos estrictos en este

> capítulo para que los conceptos se apliquen por igual a otros sistemas de bases de datos como MySQL. ↩

> “ [1]. En general, cuando una oración comienza con "si todo va bien", encontrarás que el código debe usar try / except. ↩

Revision #1
Created 2025-04-05 12:11:46 CEST by Javier Quintana
Updated 2025-04-05 12:14:28 CEST by Javier Quintana