정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance
번역: 이광춘, 한정수
(xwmooc)

Printing history:

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth. Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

The LaTeX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from `http://www.thinkpython.com`.

Chapter 1

데이터베이스와 SQL(Structured Query Language) 사용하기

1.1 데이터베이스가 뭔가요?

데이터베이스(database)는 데이터를 저장하기 위한 목적으로 조직된 파일이다. 대부분의 데이터베이스는 키(key)와 값(value)를 매핑한다는 의미에서 딕셔너리 처럼 조직되었다. 가장 큰 차이점은 데이터베이스는 디스크(혹은 다른 영구 저장소)에 위치하게 되어서, 프로그램 종료 후에도 정보가 지속적으로 저장된다. 데이터베이스가 영구 저장소에 저장되어서, 컴퓨터의 메모리 크기에 제한을 받는 딕셔너리보다 훨씬 더 많은 정보를 저장할 수 있다.

딕셔너리처럼, 데이터베이스 소프트웨어는 엄청난 양의 데이터 조차도 매우 빠르게 삽입하고 접근하도록 설계되었다. 컴퓨터가 특정 항목으로 빠르게 넘어갈 수 있도록 데이터베이스에 데이터를 추가하여 **인덱스(indexes)**를 구축하여 성능을 보장하는 것이 데이터베이스 소프트웨어다.

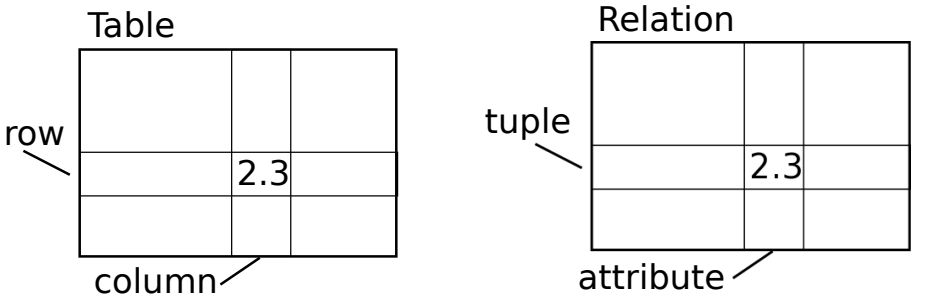
다양한 종류의 목적에 사용되는 서로 다른 많은 데이터베이스 시스템이 있다. Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite이 여기에 포함된다. SQLite를 집중해서 살펴볼 것이다. 왜냐하면 매우 일반적인 데이터베이스이고 파이썬에 이미 내장되어 있기 때문이다. SQLite는 응용프로그램 내부에서 데이터베이스 지원을 제공하도록 다른 응용프로그램에 *내장(embedded)*되도록 설계되었다. 예를 들어, 다른 많은 소프트웨어 제품이 그렇듯이, 파이어폭스 브라우져는 SQLite를 사용한다.

`http://sqlite.org/`

이번 장에서 기술하는 트위터 스파이더링 응용프로그램같은 인포매틱스(Informatics)에서 마주치는 몇몇 데이터 조작 문제에 SQLite가 적합하다.

1.2 데이터베이스 개념

처음 데이터베이스를 볼때 드는 생각은 마치 엑셀같은 다중 시트를 지닌 스프레드쉬트(spreadsheet)같다. 데이터베이스의 주요 데이터구조는 **테이블(tables)**, **행(rows)**, and **열(columns)**이다.



관계형 데이터베이스의 기술적은 설명으로 테이블, 행, 열의 개념은 **관계(relation)**, **튜플(tuple)**, and **속성(attribute)** 각각 형식적으로 참조된다. 이번장에서는 좀더 덜 형식 용어를 사용한다.

1.3 파이어폭스 애드온 SQLite 매니저

SQLite 데이터베이스 파일에 데이터를 다루기 위해서 이번장에서 파이썬의 사용에 집중을 하지만, 다음 웹사이트에서 무료로 이용가능한 **SQLite 데이터베이스 매니저(SQLite Database Manager)**로 불리는 파이어폭스 애드온을 사용해서 좀더 쉽게 많은 연산을 수행한다.

`https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/`

브라우져를 사용해서 쉽게 테이블을 생성하고, 데이터를 삽입, 편집하고 데이터베이스의 데이터에 간단한 SQL 질의를 실행할 수 있다.

이러한 점에서 데이터베이스 매니저는 텍스트 파일을 작업할 때 사용하는 텍스트 편집기와 유사하다. 텍스트 파일에 하나 혹은 몇개의 작업을 하고자 하면, 텍스트 편집기에 파일을 열어 원하는 수정을 하면 된다. 텍스트 파일에 작업할 사항이 많은 경우는 종종 간단한 파이썬 프로그램을 작성한다. 데이터베이트로 작업할 때 동일한 패턴을 찾을 수 있다. 간단한 작업은 데이터베이스 매니저를 통해서 수행하고, 좀더 복잡한 작업은 파이썬으로 수행하는 것이 가장 편리하다.

1.4 데이터베이스 테이블 생성하기

데이터베이스는 파이썬 리스트 혹은 딕셔너리보다 좀더 명확히 정의된 구조를 요구한다.[1].

[1]SQLite 실질적으로 열에 저장되는 데이터 형식에 좀더 많은 유연성을 부여하지만, 이번 장에서는 데이터 형식을 엄격하게 유지해서 MySQL 같은 다른 데이터베이스 시스템에도 동일하게 개념이 적용되게 한다.

데이터베이스 **테이블(table)**을 생성할 때, 데이터베이스에게 테이블의 각 **열(column)**의 명칭과 각 **열(column)**에 저장하려고 하는 데이터의 형식을 미리 알려줘야 한다. 데이터베이스 소프트웨어가 각 열의 데이터 형식을 인식하게 되면, 데이터 형식에 따라 가장 효율적으로 데이터를 저장하고 찾아오는 방법을 선택할 수 있다.

다음 url에서 SQLite에서 지원하는 다양한 데이터 형식을 볼 수 있다.

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

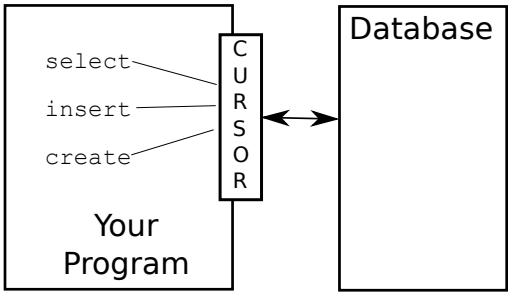
처음에는 사전에 데이터 구조를 정의하는 것이 불편하게 보이지만, 데이터베이스가 대량의 데이터를 포함하는 경우에도 데이터의 빠른 접근을 보장하는 잇점이 있다.

데이터베이스 파일과 데이터베이스에 두개의 열을 가진 `Tracks` 이름의 테이블을 생성하는 코드는 다음과 같다.

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks ')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()
```

`connect` 연산은 현재 디렉토리의 `music.sqlite3` 파일에 저장된 데이터베이스에 "연결(connection)"한다. 파일이 존재하지 않으면, 자동 생성이 된다. "연결(connection)"이라고 부르는 이유는 때때로 데이터베이스가 응용프로그램이 실행되는 서버로부터 분리된 "데이터베이스 서버(database server)"에 저장되기 때문이다. 여기 간단한 예제에서는 데이터베이스가 실행되는 파이썬 코드처럼 동일한 디렉토리에 로컬 파일이다.

커서(cursor)는 파일을 다루는 파일핸들러처럼 데이터베이스에 저장된 파일에 연산을 수행하기 위해서 사용한다. `cursor()`를 호출하는 것은 개념적으로 텍스트 파일을 다룰 때 `open()`을 호출하는 것과 개념적으로 매우 유사하다.



커서가 생성되면, `execute()` 메쏘드를 사용하여 데이터베이스 콘텐츠에 명령어 실행을 할 수 있다.

데이터베이스 명령어는 특별한 언어로 표현되어 단 하나의 데이터베이스 언어를 학습하도록 서로 다른 많은 데이터베이스 업체사이에서 표준화되었다. 데이터베이스 언어는 **SQL(Structured Query Language 구조적 질의 언어)**로 불린다.

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

상기 예제에서, 데이터베이스에 두개의 SQL 명령어를 실행했다. 관습적으로 데이터베이스 키워드는 대문자로 테이블이나 열의 명칭처럼 사용자가 추가한 명령어 부분은 소문자로 표기한다.

첫 SQL 명령어는 만약 존재한다면 데이터베이스에서 Tracks 테이블을 삭제한다. 이런 형태의 패턴은 단순하게 오류 없이 반복적으로 Tracks 테이블을 생성하도록 동일한 프로그램을 실행할 수 있게 한다. DROP TABLE 명령어는 데이터베이스로부터 테이블 및 테이블 콘텐츠 전부를 삭제함을 주목하세요. (즉, "실행취소(undo)"가 없다.)

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

두번째 명령어는 title 문자형 열과 plays 정수형 열을 가진 Tracks으로 명명된 테이블을 생성한다.

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

이제 Tracks로 명명된 테이블을 생성했으니, SQL INSERT 연산을 통해서 테이블에 데이터를 넣을 수 있다. 다시 한번, 데이터베이스에 연결하여 커서(cursor)를 얻어서 작업을 시작한다. 그리고 나서 커서를 사용하여 SQL 명령어를 수행한다.

SQL INSERT 명령어는 어느 테이블을 사용하는지, (title, plays) 을 포함하는 필드를 통해서 신규 행을 정의하고 테이블의 신규 행에 VALUES에 해당 데이터를 입력한다. 실제 값이 execute() 호출의 두번째 매개변수로 ('My Way', 15) 튜플로 넘겨는 것을 표기하기 위해서 값을 물음표 (?, ?)로 명기한다.

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'Thunderstruck', 20 ) )
cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'My Way', 15 ) )
conn.commit()

print 'Tracks:'
cur.execute('SELECT title, plays FROM Tracks')
for row in cur :
   print row

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()
```

```
cur.close()
```

먼저 테이블에 두개의 열을 삽입(INSERT)하고 commit()을 사용하여 데이터가 데이터베이스에 써지도록 했다.

Tracks

title	plays
Thunderstruck	20
My Way	15

그리고 나서, SELECT 명령어를 사용하여 테이블에 방금전에 삽입된 행을 불러왔다. SELECT 명령어에 어느 열(title, plays)을 가져오는지와 어느 테이블Tracks에서 데이터를 가져올지를 나타낸다. SELECT 명령문을 수행한 후에, 커서는 for문의 반복을 수행하는 것과 같다. 효율성을 위해서, 커서는 SELECT 명령문을 수행할 때 데이터베이스에서 모든 데이터를 읽지 않는다. 대신에 데이터는 for문의 행을 반복하듯이 요청시에만 읽어온다.

프로그램 실행결과는 다음과 같다.

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

for 루프는 두개의 행을 읽어왔다. 각각의 행은 title로 첫번째 값을, plays로 두번째 값을 가진 파이썬 튜플이다. title 문자열이 'u'로 시작한다고 걱정하지 마라. 해당 문자열은 라틴 문자가 아닌 다국어를 저장할 수 있는 **유니코드 (Unicode)** 문자열을 나타내는 것이다.

프로그램 마지막에 SQL 명령어를 실행서 방금전에 생성한 행을 모두 삭제(DELETE)했서 프로그램을 다시금 실행할 수 있다. 삭제(DELETE) 명령어는 WHERE 문을 사용하여 선택 조건을 표기할 수 있다. 따라서 명령문이 조건을 충족하는 행에만 데이터베이스에 적용된다. 이번 예제에서 기준이 모든 행에 적용되어서 테이블에 아무 것도 없게 되어서 프로그램을 반복적으로 실행할 수 있다. 삭제(DELETE)를 실행한 후에 commit()을 호출하여 데이터가 데이터베이스에서 와전히 제거되게 했다.

1.5 SQL(Structured Query Language) 요약

So far, we have been using the Structured Query Language in our Python examples and have covered many of the basics of the SQL commands. In this section, we look at the SQL language in particular and give an overview of SQL syntax.

Since there are so many different database vendors, the Structured Query Language (SQL) was standardized so we could communicate in a portable manner to database systems from multiple vendors.

A relational database is made up of tables, rows, and columns. The columns generally have a type such as text, numeric, or date data. When we create a table, we indicate the names and types of the columns:

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

To insert a row into a table, we use the SQL `INSERT` command:

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

The `INSERT` statement specifies the table name, and then a list of the fields/columns that you would like to set in the new row, and then the keyword `VALUES` and then a list of corresponding values for each of the fields.

The SQL `SELECT` command is used to retrieve rows and columns from a database. The `SELECT` statement lets you specify which columns you would like to retrieve as well as a `WHERE` clause to select which rows you would like to see. It also allows an optional `ORDER BY` clause to control the sorting of the returned rows.

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

Using `*` indicates that you want the database to return all of the columns for each row that matches the `WHERE` clause.

Note, unlike in Python, in a SQL `WHERE` clause we use a single equal sign to indicate a test for equality rather than a double equal sign. Other logical operations allowed in a `WHERE` clause include `<`, `>`, `<=`, `>=`, `!=`, as well as `AND` and `OR` and parentheses to build your logical expressions.

You can request that the returned rows be sorted by one of the fields as follows:

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

To remove a row, you need a `WHERE` clause on an SQL `DELETE` statement. The `WHERE` clause determines which rows are to be deleted:

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

It is possible to `UPDATE` a column or columns within one or more rows in a table using the SQL `UPDATE` statement as follows:

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

The `UPDATE` statement specifies a table and then a list of fields and values to change after the `SET` keyword and then an optional `WHERE` clause to select the rows that are to be updated. A single `UPDATE` statement will change all of the rows that match the `WHERE` clause, or if a `WHERE` clause is not specified, it performs the `UPDATE` on all of the rows in the table.

These four basic SQL commands (INSERT, SELECT, UPDATE, and DELETE) allow the four basic operations needed to create and maintain data.

1.6 Spidering Twitter using a database

In this section, we will create a simple spidering program that will go through Twitter accounts and build a database of them. *Note: Be very careful when running this program. You do not want to pull too much data or run the program for too long and end up having your Twitter access shut off.*

One of the problems of any kind of spidering program is that it needs to be able to be stopped and restarted many times and you do not want to lose the data that you have retrieved so far. You don't want to always restart your data retrieval at the very beginning so we want to store data as we retrieve it so our program can start back up and pick up where it left off.

We will start by retrieving one person's Twitter friends and their statuses, looping through the list of friends, and adding each of the friends to a database to be retrieved in the future. After we process one person's Twitter friends, we check in our database and retrieve one of the friends of the friend. We do this over and over, picking an "unvisited" person, retrieving their friend list and adding friends we have not seen to our list for a future visit.

We also track how many times we have seen a particular friend in the database to get some sense of "popularity".

By storing our list of known accounts and whether we have retrieved the account or not, and how popular the account is in a database on the disk of the computer, we can stop and restart our program as many times as we like.

This program is a bit complex. It is based on the code from the exercise earlier in the book that uses the Twitter API.

Here is the source code for our Twitter spidering application:

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('''
CREATE TABLE IF NOT EXISTS Twitter
(name TEXT, retrieved INTEGER, friends INTEGER)''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    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

    url = twurl.augment(TWITTER_URL,
              {'screen_name': acct, 'count': '20'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    # print 'Remaining', headers['x-rate-limit-remaining']
    js = json.loads(data)
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )

    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()

cur.close()
```

Our database is stored in the file `spider.sqlite3` and it has one table named `Twitter` and each row in the `Twitter` table has a column for the account name, whether we have retrieved the friends of this account, and how many times this account has been "friended".

In the main loop of the program, we prompt the user for a Twitter account name or "quit" to exit the program. If the user enters a Twitter account, we retrieve the list of friends and statuses for that user and add each friend to the database if not already in the database. If the friend is already in the list, we add one to the `friends` field in the row in the database.

If the user presses enter, we look in the database for the next Twitter account that we have not yet retrieved and retrieve the friends and statuses for that account, add them to the database or update them and increase their `friends count`.

Once we retrieve the list of friends and statuses, we loop through all of the `user`

items in the returned JSON and retrieve the screen_name for each user. Then we use the SELECT statement to see if we already have stored this particular screen_name in the database and retrieve the friend count (friends) if the record exists.

```
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()
```

Once the cursor executes the SELECT statement, we must retrieve the rows. We could do this with a for statement, but since we are only retrieving one row (LIMIT 1), we can use the fetchone() method to fetch the first (and only) row that is the result of the SELECT operation. Since fetchone() returns the row as a **tuple** (even though there is only one field), we take the first value from the tuple using [0] to get the current friend count into the variable count.

If this retrieval is successful, we use the SQL UPDATE statement with a WHERE clause to add one to the friends column for the row that matches the friend's account. Notice that there are two placeholders (i.e. question marks) in the SQL, and the second parameter to the execute() is a two-element tuple which holds the values to be substituted into the SQL in place of the question marks.

If the code in the try block fails it is probably because no record matched the WHERE name = ? clause on the SELECT statement. So in the except block, we use the SQL INSERT statement to add the friend's screen_name to the table with an indication that we have not yet retrieved the screen_name and setting the friend count to zero.

So the first time the program runs and we enter a Twitter account, the program runs as follows:

```
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
```

Since this is the first time we have run the program, the database is empty and we create the database in the file spider.sqlite3 and add a table named Twitter

to the database. Then we retrieve some friends and add them all to the database since the database is empty.

At this point, we might want to write a simple database dumper to take a look at what is in our `spider.sqlite3` file:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur :
   print row
   count = count + 1
print count, 'rows.'
cur.close()
```

This program simply opens the database and selects all of the columns of all of the rows in the table `Twitter`, then loops through the rows and prints out each row.

If we run this program after the first execution of our Twitter spider above, its output will be as follows:

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

We see one row for each `screen_name`, that we have not retrieved the data for that `screen_name` and everyone in the database has one friend.

Now our database reflects the retrieval of the friends of our first Twitter account (**drchuck**). We can run the program again and tell it to retrieve the friends of the next "unprocessed" account by simply pressing enter instead of a Twitter account as follows:

```
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
```

Since we pressed enter (i.e. we did not specify a Twitter account), the following code is executed:

```
    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
```

We use the SQL `SELECT` statement to retrieve the name of the first (`LIMIT 1`) user who still has their "have we retrieved this user" value set to zero. We also use the `fetchone()[0]` pattern within a try/except block to either extract a `screen_name` from the retrieved data or put out an error message and loop back up.

If we successfully retrieved an unprocessed `screen_name`, we retrieve their data as follows:

```
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 name = ?', (acct, ) )
```

Once we retrieve the data successfully, we use the `UPDATE` statement to set the `retrieved` column to one to indicate that we have completed the retrieval of the friends of this account. This keeps us from re-retrieving the same data over and over and keeps us progressing forward through the network of Twitter friends.

If we run the friend program and press enter twice to retrieve the next unvisited friend's friends, then run the dumping program, it will give us the following output:

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

We can see that we have properly recorded that we have visited `lhawthorn` and `opencontent`. Also the accounts `cnxorg` and `kthanos` already have two followers. Since we now have retrieved the friends of three people (`drchuck`, `opencontent` and `lhawthorn`) our table has 55 rows of friends to retrieve.

Each time we run the program and press enter, it will pick the next unvisited account (e.g. the next account will be `steve_coppin`), retrieve their friends, mark them as retrieved and for each of the friends of `steve_coppin`, either add them to the end of the database, or update their friend count if they are already in the database.

Since the program's data is all stored on disk in a database, the spidering activity can be suspended and resumed as many times as you like with no loss of data.

1.7 Basic data modeling

The real power of a relational database is when we make multiple tables and make links between those tables. The act of deciding how to break up your application data into multiple tables and establishing the relationships between the two tables is called **data modeling**. The design document that shows the tables and their relationships is called a **data model**.

Data modeling is a relatively sophisticated skill and we will only introduce the most basic concepts of relational data modeling in this section. For more detail on data modeling you can start with:

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

Let's say for our Twitter spider application, instead of just counting a person's friends, we wanted to keep a list of all of the incoming relationships so we could find a list of everyone who is following a particular account.

Since everyone will potentially have many accounts that follow them, we cannot simply add a single column to our `Twitter` table. So we create a new table that keeps track of pairs of friends. The following is a simple way of making such a table:

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

Each time we encounter a person who `drchuck` is following, we would insert a row of the form:

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

As we are processing the 20 friends from the `drchuck` Twitter feed, we will insert 20 records with "drchuck" as the first parameter so we will end up duplicating the string many times in the database.

This duplication of string data violates the best practices for **database normalization** which basically states that we should never put the same string data in the database more than once. If we need the data more than once, we create a numeric **key** for the data and reference the actual data using this key.

In practical terms, a string takes up a lot more space than an integer on the disk and in the memory of our computer and takes more processor time to compare and sort. If we only have a few hundred entries the storage and processor time hardly matters. But if we have a million people in our database and a possibility of 100 million friend links, it is important to be able to scan data as quickly as possible.

We will store our Twitter accounts in a table named `People` instead of the `Twitter` table used in the previous example. The `People` table has an additional column to store the numeric key associated with the row for this Twitter user. SQLite has a feature that automatically adds the key value for any row we insert into a table using a special type of data column (`INTEGER PRIMARY KEY`).

We can create the `People` table with this additional `id` column as follows:

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

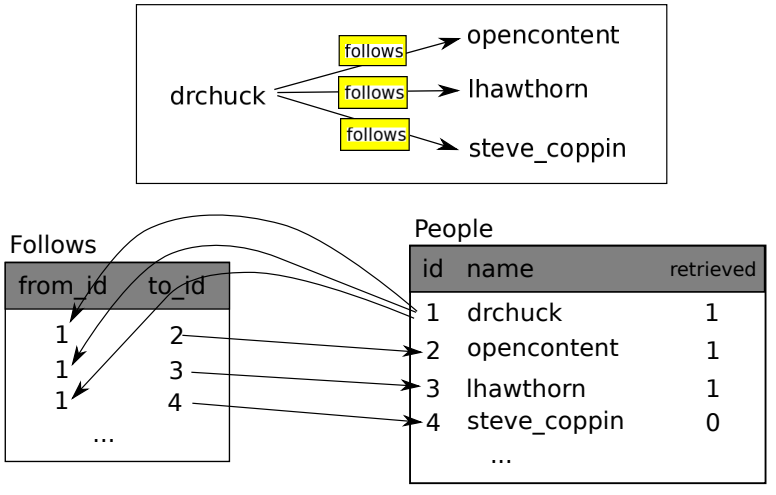
Notice that we are no longer maintaining a friend count in each row of the `People` table. When we select `INTEGER PRIMARY KEY` as the type of our `id` column, we are indicating that we would like SQLite to manage this column and assign a unique numeric key to each row we insert automatically. We also add the keyword `UNIQUE` to indicate that we will not allow SQLite to insert two rows with the same value for `name`.

Now instead of creating the table `Pals` above, we create a table called `Follows` with two integer columns `from_id` and `to_id` and a constraint on the table that the *combination* of `from_id` and `to_id` must be unique in this table (i.e. we cannot insert duplicate rows) in our database.

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

When we add `UNIQUE` clauses to our tables, we are communicating a set of rules that we are asking the database to enforce when we attempt to insert records. We are creating these rules as a convenience in our programs as we will see in a moment. The rules both keep us from making mistakes and make it simpler to write some of our code.

In essence, in creating this `Follows` table, we are modelling a "relationship" where one person "follows" someone else and representing it with a pair of numbers indicating that (a) the people are connected and (b) the direction of the relationship.



1.8 Programming with multiple tables

We will now re-do the Twitter spider program using two tables, the primary keys, and the key references as described above. Here is the code for the new version of the program:

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlitesqlite3')
cur = conn.cursor()

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))''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
        cur.execute('''SELECT id, name FROM People
            WHERE retrieved = 0 LIMIT 1''')
        try:
            (id, acct) = cur.fetchone()
        except:
            print 'No unretrieved Twitter accounts found'
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (acct, ) )
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( acct, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',acct
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL,
       {'screen_name': acct, 'count': '20'} )
    print 'Retrieving account', acct
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    print 'Remaining', headers['x-rate-limit-remaining']

    js = json.loads(data)
```

```
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        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
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
            VALUES (?, ?)''', (id, friend_id) )
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

This program is starting to get a bit complicated, but it illustrates the patterns that we need to use when we are using integer keys to link tables. The basic patterns are:

1. Creating tables with primary keys and constraints.
2. When we have a logical key for a person (i.e. account name) and we need the `id` value for the person. Depending on whether or not the person is already in the `People` table, we either need to: (1) look up the person in the `People` table and retrieve the `id` value for the person or (2) add the person to the `People` table and get the `id` value for the newly added row.
3. Insert the row that captures the "follows" relationship.

We will cover each of these in turn.

1.8.1 Constraints in database tables

As we design our table structures, we can tell the database system that we would like it to enforce a few rules on us. These rules help us from making mistakes and introducing incorrect data into out tables. When we create our tables:

```
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))''')
```

We indicate that the `name` column in the `People` table must be `UNIQUE`. We also indicate that the combination of the two numbers in each row of the `Follows` table must be unique. These constraints keep us from making mistakes such as adding the same relationship more than once.

We can take advantage of these constraints in the following code:

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

We add the `OR IGNORE` clause to our `INSERT` statement to indicate that if this particular `INSERT` would cause a violation of the "`name` must be unique" rule, the database system is allowed to ignore the `INSERT`. We are using the database constraint as a safety net to make sure we don't inadvertently do something incorrect.

Similarly, the following code ensures that we don't add the exact same `Follows` relationship twice.

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

Again we simply tell the database to ignore our attempted `INSERT` if it would violate the uniqueness constraint that we specified for the `Follows` rows.

1.8.2 Retrieve and/or insert a record

When we prompt the user for a Twitter account, if the account exists, we must look up its `id` value. If the account does not yet exist in the `People` table, we must insert the record and get the `id` value from the inserted row.

This is a very common pattern and is done twice in the program above. This code shows how we look up the `id` for a friend's account when we have extracted a `screen_name` from a `user` node in the retrieved Twitter JSON.

Since over time it will be increasingly likely that the account will already be in the database, we first check to see if the `People` record exists using a `SELECT` statement.

If all goes well[2] inside the `try` section, we retrieve the record using `fetchone()` and then retrieve the first (and only) element of the returned tuple and store it in `friend_id`.

If the `SELECT` fails, the `fetchone()[0]` code will fail and control will transfer into the `except` section.

[2] In general, when a sentence starts with "if all goes well" you will find that the code needs to use try/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
```

If we end up in the `except` code, it simply means that the row was not found so we must insert the row. We use `INSERT OR IGNORE` just to avoid errors and then call `commit()` to force the database to really be updated. After the write is done, we can check the `cur.rowcount` to see how many rows were affected. Since we are attempting to insert a single row, if the number of affected rows is something other than one, it is an error.

If the `INSERT` is successful, we can look at `cur.lastrowid` to find out what value the database assigned to the `id` column in our newly created row.

1.8.3 Storing the friend relationship

Once we know the key value for both the Twitter user and the friend in the JSON, it is a simple matter to insert the two numbers into the `Follows` table with the following code:

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

Notice that we let the database take care of keeping us from "double-inserting" a relationship by creating the table with a uniqueness constraint and then adding `OR IGNORE` to our `INSERT` statement.

Here is a sample execution of this program:

```
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
```

We started with the drchuck account and then let the program automatically pick the next two accounts to retrieve and add to our database.

The following is the first few rows in the People and Follows tables after this run is completed:

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

You can see the id, name, and visited fields in the People table and you see the numbers of both ends of the relationship Follows table. In the People table, we can see that the first three people have been visited and their data has been retrieved. The data in the Follows table indicates that drchuck (user 1) is a friend to all of the people shown in the first five rows. This makes sense because the first data we retrieved and stored was the Twitter friends of drchuck. If you were to print more rows from the Follows table, you would see the friends of user two and three as well.

1.9 Three kinds of keys

Now that we have started building a data model putting our data into multiple linked tables, and linking the rows in those tables using **keys**, we need to look at some terminology around keys. There are generally three kinds of keys used in a database model.

- A **logical key** is a key that the "real world" might use to look up a row. In our example data model, the name field is a logical key. It is the screen name for the user and we indeed look up a user's row several times in the program using the name field. You will often find that it makes sense to add a UNIQUE constraint to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.

- A **primary key** is usually a number that is assigned automatically by the database. It generally has no meaning outside the program and is only used to link rows from different tables together. When we want to look up a row

in a table, usually searching for the row using the primary key is the fastest way to find a row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly. In our data model, the `id` field is an example of a primary key.

- A **foreign key** is usually a number that points to the primary key of an associated row in a different table. An example of a foreign key in our data model is the `from_id`.

We are using a naming convention of always calling the primary key field name `id` and appending the suffix `_id` to any field name that is a foreign key.

1.10 Using JOIN to retrieve data

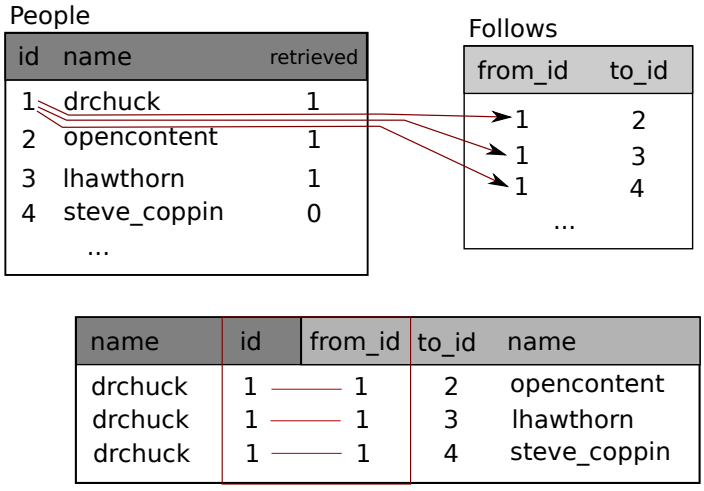
Now that we have followed the rules of database normalization and have data separated into two tables, linked together using primary and foreign keys, we need to be able to build a `SELECT` that re-assembles the data across the tables.

SQL uses the `JOIN` clause to re-connect these tables. In the `JOIN` clause you specify the fields that are used to re-connect the rows between the tables.

The following is an example of a `SELECT` with a `JOIN` clause:

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

The `JOIN` clause indicates that the fields we are selecting cross both the `Follows` and `People` tables. The `ON` clause indicates how the two tables are to be joined. Take the rows from `Follows` and append the row from `People` where the field `from_id` in `Follows` is the same the `id` value in the `People` table.



The result of the JOIN is to create extra-long "meta-rows" which have both the fields from `People` and the matching fields from `Follows`. Where there is more

than one match between the `id` field from `People` and the `from_id` from `People`, then JOIN creates a meta-row for *each* of the matching pairs of rows, duplicating data as needed.

The following code demonstrates the data that we will have in the database after the multi-table Twitter spider program (above) has been run several times.

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print 'People:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('SELECT * FROM Follows')
count = 0
print 'Follows:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('''SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 2''')
count = 0
print 'Connections for id=2:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.close()
```

In this program, we first dump out the `People` and `Follows` and then dump out a subset of the data in the tables joined together.

Here is the output of the program:

```
python twjoin.py
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
```

```
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, u'drchuck', 1)
(2, 28, 28, u'cnxorg', 0)
(2, 30, 30, u'kthanos', 0)
(2, 102, 102, u'SomethingGirl', 0)
(2, 103, 103, u'ja_Pac', 0)
20 rows.
```

You see the columns from the `People` and `Follows` tables and the last set of rows is the result of the `SELECT` with the `JOIN` clause.

In the last select, we are looking for accounts that are friends of "opencontent" (i.e. `People.id=2`).

In each of the "meta-rows" in the last select, the first two columns are from the `Follows` table followed by columns three through five from the `People` table. You can also see that the second column (`Follows.to_id`) matches the third column (`People.id`) in each of the joined-up "meta-rows".

1.11 Summary

This chapter has covered a lot of ground to give you an overview of the basics of using a database in Python. It is more complicated to write the code to use a database to store data than Python dictionaries or flat files so there is little reason to use a database unless your application truly needs the capabilities of a database. The situations where a database can be quite useful are: (1) when your application needs to make small many random updates within a large data set, (2) when your data is so large it cannot fit in a dictionary and you need to look up information repeatedly, or (3) you have a long-running process that you want to be able to stop and restart and retain the data from one run to the next.

You can build a simple database with a single table to suit many application needs, but most problems will require several tables and links/relationships between rows in different tables. When you start making links between tables, it is important to do some thoughtful design and follow the rules of database normalization to make the best use of the database's capabilities. Since the primary motivation for using a database is that you have a large amount of data to deal with, it is important to model your data efficiently so your programs run as fast as possible.

1.12 Debugging

One common pattern when you are developing a Python program to connect to an SQLite database will be to run a Python program and check the results using the

SQLite Database Browser. The browser allows you to quickly check to see if your program is working properly.

You must be careful because SQLite takes care to keep two programs from changing the same data at the same time. For example, if you open a database in the browser and make a change to the database and have not yet pressed the "save" button in the browser, the browser "locks" the database file and keeping any other program from accessing the file. In particular, your Python program will not be able to access the file if it is locked.

So a solution is to make sure to either close the database browser or use the **File** menu to close the database in the browser before you attempt to access the database from Python to avoid the problem of your Python code failing because the database is locked.

1.13 Glossary

attribute: One of the values within a tuple. More commonly called a "column" or "field".

constraint: When we tell the database to enforce a rule on a field or a row in a table. A common constraint is to insist that there can be no duplicate values in a particular field (i.e. all the values must be unique).

cursor: A cursor allows you to execute SQL commands in a database and retrieve data from the database. A cursor is similar to a socket or file handle for network connections and files respectively.

database browser: A piece of software that allows you to directly connect to a database and manipulate the database directly without writing a program.

foreign key: A numeric key that points to the primary key of a row in another table. Foreign keys establish relationships between rows stored in different tables.

index: Additional data that the database software maintains as rows are inserted into a table designed to make lookups very fast.

logical key: A key that the "outside world" uses to look up a particular row. For example in a table of user accounts, a person's e-mail address might be a good candidate as the logical key for the user's data.

normalization: Designing a data model so that no data is replicated. We store each item of data at one place in the database and reference it elsewhere using a foreign key.

primary key: A numeric key assigned to each row that is used to refer to one row in a table from another table. Often the database is configured to automatically assign primary keys as rows are inserted.

relation: An area within a database that contains tuples and attributes. More typically called a “table”.

tuple: A single entry in a database table that is a set of attributes. More typically called “row”.