15.9: CONSTRAINTS IN DATABASE TABLES

Contributed by Chuck Severance
Clinical Associate Professor (School of Information) at University of Michigan

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.