MODULE 5

NETWORKING IN PYTHON

Networking support is provided in python using in-built method called socket in the socket module. Sockets help to make network connections and retrieve data through them. A socket provides a two-way connection between programs. Data can be both read from and written into through the same socket.

Data can be communicated over the Internet through network protocols. HTTP is one such protocol which allows users to communicate and retrieve web pages from the Internet. Protocol is a set of rules that define how data is transmitted from one end to another over a communication medium.

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers.

Python program that makes a connection to a web server and follows the rules of the HTTP protocol to request a document and display what the server sends back.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

mysock.connect(('data.pr4e.org', 80))

 cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()

mysock.send(cmd)

while True:

        data = mysock.recv(20)

         if (len(data) < 1): break

        print(data.decode(),end=' ')

mysock.close()
```

Socket:

A network connection between two applications where the applications can send and receive data in either direction

socket method

The **socket()** call creates a socket in the specified domain and of the specified type.

Syntax:

s = socket(domain, type, protocol);

- s is the socket handle or file descriptor which is returned on establishing a socket connection.
- Domain -AF_INET defines Address Family belonging to Internet
- Type can be SOCK_STREAM or SOCK_DGRAM

Port:

A number that generally indicates which application you are contacting when you make a socket connection to a server. As an example, web traffic(HTTP) usually uses port 80 while email traffic uses port 25.

Output

HTTP/1.1 200 OK

Date: Wed, 25 Apr 2018 23:02:51 GMT

Server: Apache/2.4.18 (Ubuntu)

Last-Modified: Sat, 13 May 2017 11:22:22 GMT

ETag: "a7-54f6609245537"

Accept-Ranges: bytes

Content-Length: 167

Cache-Control: max-age=0, no-cache, no-store, must-revalidate

Pragma: no-cache

Expires: Wed, 11 Jan 1984 05:00:00 GMT

Connection: close

Content-Type: text/plain

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief

2. Retrieving web pages with urllib

```
import urllib.request
fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())
```

Output:

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief

Urlopen method returns a file object and hence the contents can be accessed normally like a file.The contents can be traversed using the file handle.

Urllib package prints only the body of the message and removes all headers during display

Retrieve the data for romeo.txt and compute the frequency of each word in the file

```
import urllib.request
fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)
```

3. Parsing HTML and scraping the web

Web scraping is retrieving web pages and examining the data in those pages looking for patterns. This can be done using regular expressions.

Example: Search engine:

The act of a web search engine retrieving a page and then all the pages linked from a page and so on until they have nearly all the pages on the Internet which they use to build their search index. **This is called Spidering**

Parsing HTML using regular expressions

<h1>The First Page</h1>

** If you like, you can switch to the **

Second Page.

** **

Program to extract links from the above HTML page

```
import urllib.request
import re
url = input('Enter - ')
html = urllib.request.urlopen(url).read()
links = re.findall(b'href="(http://.*?)"', html)
for link in links:
        print(link.decode())
```

output:

Enter - http://www.dr-chuck.com/page1.htm

http://www.dr-chuck.com/page2.htm

The regular expression looks for strings that start with "href="http://", followed by one or more characters (".+'), followed by another double quote. The question mark added to the".*?" indicates that the match is to be done in a "non-greedy" fashion instead of a "greedy" fashion. A non-greedy match tries to find the smallest possible matching string and a greedy match tries to find the largest possible matching string.

Parentheses is added to the regular expression to extract only the links.

4.Retrieving an image using HTTP

```
import socket import time
HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET,socket.SOCK_STREAM) mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0 #stores the length of the entire data
picture = 'b '#picture initialisation in bytes
while True:
        data = mysock.recv(5120)
```

```
        if (len(data) < 1): break
        time.sleep(0.25)
        count = count + len(data)
        print(len(data), count)
        picture = picture + data
mysock.close()
```

 # Look for the end of the header (2 CRLF)

```
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())    #prints header information
```

 # Skip past the header and save the picture data

```
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close() #stores image locally in a file
```

The Content-Type header indicates that body of the document is an image (image/jpeg). Once the program completes, the image data can be viewed by opening the file stuff.jpg in an image viewer.

As the program runs, 5120 characters are not received each time a call is made to the recv() method. In this example, either 1460 or 2920 characters are received each time a request up to 5120 characters of data is made. This depends on the speed of the connection at the moment the recv() call is made.

The results may be different depending on your network speed.

The successive recv() calls can be slowed down by the call to time.sleep(). This way, a quarter of a second wait is introduced after each call so that the server can "get ahead" and send more data before recv() is called again.

5. Retrieving binary files using urllib

Non tex(binary)t files such as image or video files can be transferred using the urllib.The pattern is to open the URL and use read to download the entire contents of the document into a string variable (img) then write that information to a local file.

```
import urllib.request, urllib.parse, urllib.error
img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
```

```
fhand.write(img)

fhand.close()
```

This program reads all of the data in at once across the network and stores it in the variable img in the main memory of your computer, then opens the file cover.jpg and writes the data out to your disk. This will work if the size of the file is less than the size of the memory of your computer.

However if this is a large audio or video file, this program may crash or at least run extremely slowly when your computer runs out of memory. In order to avoid running out of memory, we retrieve the data in blocks (or buffers) and then write each block to your disk before retrieving the next block. This way the program can read any size file without using up all of the memory you have in your computer.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')

fhand = open ('cover3.jpg', 'wb')

size = 0

while True:

        info = img.read(100000)

        if len(info) < 1: break

        size = size + len(info)

        fhand.write(info)

print(size, 'characters copied.')

fhand.close()
```

In this example, we read only 100,000 characters at a time and then write those characters to the cover.jpg file before retrieving the next 100,000 characters of data from the web.

Using Web Services

Accessing and transferring data across the web can be done and processed in two most common formats.The following are the two common data interchange format across the web

- **XML - Document style dat**
- **JSON - Dictionaries ,lists and other data types**

XML – Extensible Mark up Language-

XML is a standard for custom mark up languages for text documents.

It is entirely made of text

The text based tags are called mark up.

It is different from HTML since it uses customised tags for storing elements whereas HTML uses predefined tags.Hence it is called extensible.

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications. It is mainly used in webpages, where the data has a specific structure. XML creates a tree-like structure that is easy to interpret and supports a hierarchy.

It is a widely used standard for storing structured data as well as sharing data across different applications.

XML documents have sections, called *elements*, defined by a beginning and an ending *tag*. A tag is a markup construct that begins with < and ends with >. The characters between the start-tag and end-tag, if there are any, are the element's content. Elements can contain markup, including other elements, which are called "child elements".

- The largest, top-level element is called the *root*, which contains all other elements.
- Attributes are name–value pair that exist within a start-tag or empty-element tag. An XML attribute can only have a single value and each attribute can appear at most once on each element.

XML Parser Architectures

The entire XML file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document. The XML tree structure makes navigation, modification, and removal relatively simple.

Python has a built in library, ElementTree, that has functions to read and manipulate XMLs

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ElementTree has two classes for this purpose –

ElementTree -represents the whole XML document as a tree

Element represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the **ElementTree** level. Interactions with a single XML element and its sub-elements are done on the **Element** level

Element.find(match)

Finds the first subelement matching *match*. *match* may be a tag name or path. Returns an element instance or None.

Element.text()

accesses the element's text content

Element.get()

accesses the element's attributes:

get(key, default=None)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

Here is a sample of an XML document:

```
<person>
   <name>Chuck</name>
   <phone type="intl">
      +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

Python program to parse an XML file and print element text and attributes

```
import xml.etree.ElementTree as ET
data = '''
<person>
    <name>Chuck</name>
   <phone type="intl">
      +1 734 303 4456
    </phone>
  <email hide="yes"/>
</person>'''
tree = ET.fromstring(data) #creates an Elementinstance
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))
```

Calling fromstring converts the string representation of the XML into a “tree” of XML nodes. When the XML is in a tree, there are a series of methods that can be called to extract portions of data from the XML.

The find function searches through the XML tree and retrieves a node that matches the specified tag. Each node can have some text, some attributes (like hide), and some “child” nodes. Each node can be the top of a tree of nodes.

Looping through nodes

An XML tree can have any number of nodes and they can be traversed to process each of them separately.

```
import xml.etree.ElementTree as ET

input = '''
 <stuff>
    <users>
       <user x="2">
          <id>001</id>
          <name>Chuck</name>
       </user>
       <user x="7">
          <id>009</id>
          <name>Brent</name>
       </user>
    </users>
 </stuff>'''
stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))
for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get("x"))
```

The findall method retrieves a Python list of subtrees that represent the user structures in the XML tree. Then we can write a for loop that looks at each of the user nodes, and prints the name and id text elements as well as the x attribute from the user node.

Output:

User count: 2

Name Chuck

Id 001

Attribute 2

Name Brent

Id 009

Attribute 7

JAVASCRIPT OBJECT NOTATION(JSON)

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language.The JSON format was inspired by the object and array format used in the JavaScript language

```
{
"name" : "Chuck",
"phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456" },
 "email" : {
    "hide" : "yes" }
}
```

JSON codes data as key- value pairs. JSON has the advantage that it maps directly to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python's dictionaries and lists, JSON is a very natural format to have two cooperating programs exchange data.

JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

Parsing JSON

JSON can be constructed by nesting dictionaries (objects) and lists as needed. In this example, a list of users are represented where each user is a set of key-value pairs (i.e., a dictionary). So it's a list of dictionaries.

Python provides built in json library to parse the JSON and read through the data. Comparing this closely to the equivalent XML data and code, the JSON has less detail, getting a list and

that the list is of users and each user is a set of key-value pairs. The JSON is more succinct (an advantage) but also is less self-describing (a disadvantage)

```
import json

data = '''
[
        { "id" : "001",
          "x" : "2",
          "name" : "Chuck" } ,
        { "id" : "009",
          "x" : "7",
          "name" : "Chuck" }
]'''
info = json.loads(data)
print('User count:', len(info))
for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])
```

If the code to extract data from the parsed JSON and parsed XML is compared ,it can be seen that json.loads() returns a Python list which we traverse with a for loop, and each item within that list is a Python dictionary. Once the JSON has been parsed, we can use the Python index operator to extract the various bits of data for each user. We don't have to use the JSON library to dig through the parsed JSON, since the returned data is simply native Python structures.Hence using JSON data format for data exchange is easier and can be processed much better and faster than XML.

The output of this program is the same as the XML version above.

```
User count: 2
 Name Chuck
 Id 001
Attribute 2
 Name Brent
 Id 009
```

Attribute 7

In general, there is an industry trend away from XML and towards JSON for web services. Because the JSON is simpler and more directly maps to native data structures we already have in programming languages, the parsing and data extraction code is usually simpler and more direct when using JSON. But XML is more self descriptive than JSON and so there are some applications where XML retains an advantage. For example, most word processors store documents internally using XML rather than JSON.

Web Services/Application Programming Interface

Web service:

When an application makes a set of services in its API available over the web, we call these web services.

API

An application to application interface that's allows data exchange between web services is called an Application Programming Interface.

The general name for these application-to-application contracts is Application Program Interfaces or APIs. When we use an API, generally one program makes a set of services available for use by other applications and publishes the APIs (i.e., the "rules") that must be followed to access the services provided by the program.

When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a **Service-Oriented Architecture** or SOA. A SOA approach is one where our overall application makes use of the services of other applications. A non-SOA approach is where the application is a single standalone application which contains all of the code necessary to implement the application

Example of SOA:

Web provides a number of applications that follow SOA. We can go to a single web site and book air travel, hotels, and automobiles all from a single site. The data for hotels is not stored on the airline computers. Instead, the airline computers contact the services on the hotel computers and retrieve the hotel data and present it to the user. When the user agrees to make a hotel reservation using the airline site, the airline site uses another web service on the hotel systems to actually make the reservation. And when it comes time to charge your credit card for the whole transaction, still other computers become involved in the process

A Service-Oriented Architecture has many advantages

(1) we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit)

(2) the owners of the data can set the rules about the use of their data.

Google geocoding web service

Geocoding is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coordinates (like latitude 37.423021 and longitude -122.083739), which you can use to place markers on a map or position the map.

Google has an excellent web service that allows us to make use of their large database of geographic information. We can submit a geographical search string like “Ann Arbor, MI” to their geocoding API and have Google return its best guess as to where on a map we might find our search string and tell us about the landmarks nearby.The geocoding service is free but rate limited so it cannot be used to make unlimited calls to the API in a commercial application.

The following is a simple application to prompt the user for a search string, call the Google geocoding API, and extract information from the returned JSON.

```
import urllib.request, urllib.parse, urllib.error
import json
serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'
while True:
    address = input('Enter location: ')
    if len(address) < 1:
        break
    url = serviceurl + urllib.parse.urlencode( {'address': address})
    print('Retrieving', url)
    uh = urllib.request.urlopen(url)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')
    try:
        js = json.loads(data)
    except:
        js = None
    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue
    print(json.dumps(js, indent=4))
```

```
    lat=js["results"][0]["geometry"]["location"]["lat"]
lng = js["results"][0]["geometry"]["location"]["lng"]
print('lat', lat, 'lng', lng)
location = js['results'][0]['formatted_address']
print(location)
```

The program takes the search string and constructs a URL with the search string as a properly encoded parameter and then uses urllib to retrieve the text from the Google geocoding API. Unlike a fixed web page, the data we get depends on the parameters we send and the geographical data stored in Google's servers.

Once we retrieve the JSON data, we parse it with the json library and do a few checks to make sure that we received good data, then extract the information that we are looking for.

Spidering Twitter using Twitter API

Social media provide a number of web services in the form of API's which can be made use of in order to analyse text content available on social media to perform aplications such as opinion mining and sentiment analysis.

Twitter spidering and scraping is one such application that provides API's which can be used to extract information such as number of friends/followers of a particular account, their timeline information, the communication messages and tweets.

Twitter authentication

Twitter restricts usage of its API's by providing proper authentication to a particular user who holds a twitter account.

A twitter account holder should create an application through the account,by which each user is given the following keys and tokens for authentication

Consumer Key

Consumer Token

Token key

Token Secret

These keys/tokens are updated in the hidden.py file to do twitter spidering.

Files to be available in the twitter package, hidden.py,twurl.py,oauth.py

This program retrieves the timeline for a particular Twitter user and returns it to us in JSON format in a string. We simply print the first 250 characters of the string:

```
import urllib.request, urllib.parse, urllib.error
```

```
import twurl
import ssl
# https://apps.twitter.com/ # Create App and get the four strings, put them in hidden.py
TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'
# Ignore SSL certificate errors
 ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
while True:
        print('')
        acct = input('Enter Twitter Account:')
        if (len(acct) < 1):
                break
        url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '2'})
        print('Retrieving', url)
        connection = urllib.request.urlopen(url, context=ctx)
        data=connection.read().decode()
        print(data[:250])
        headers = dict(connection.getheaders())
        # print headers
        print('Remaining', headers['x-rate-limit-remaining'])
```

Along with the returned timeline data, Twitter also returns metadata about the request in the HTTP response headers. One header in particular, x-rate-limit remaining, informs us how many more requests we can make before we will be shut off for a short time period. You can see that our remaining retrievals drop by one each time we make a request to the API.

The following program retrieves a user's Twitter friends, parse the returned JSON, and extracts some of the information about the friends

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
 import ssl
# https://apps.twitter.com/ # Create App and get the four strings, put them in hidden.py
```

```
TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'
# Ignore SSL certificate errors
 ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    js = json.loads(data)
    print(json.dumps(js, indent=2))
    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])
    for u in js['users']:
      print(u['screen_name'])
      if 'status' not in u:
          print(' * No status found')
          continue
      s = u['status']['text']
      print(' ', s[:50])
```

Database Connectivity with Python

A **database** is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends. Because a database is stored on permanent storage, it can store far more data than a dictionary, which is limited to the size of the memory in the computer.

Like a dictionary, database software is designed to keep the inserting and accessing of data very fast, even for large amounts of data. Database software maintains its performance by building indexes as data is added to the database to allow the computer to jump quickly to a particular entry.

There are many different database systems which are used for a wide variety of purposes including: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SQLite

Creating a database table

The following code shows how a database table can be created using sqlite3

```
import sqlite3

conn = sqlite3.connect('music.sqlite')

cur = conn.cursor()

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

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

conn.close()
```

The connect operation makes a "connection" to the database stored in the file music.sqlite3 in the current directory. If the file does not exist, it will be created. The reason this is called a "connection" is that sometimes the database is stored on a separate "database server" from the server on which we are running our application. In our simple examples the database will just be a local file in the same directory as the Python code we are running.

A cursor is like a file handle that we can use to perform operations on the data stored in the database. Calling cursor() is very similar conceptually to calling open() when dealing with text files

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

The command creates a table named Tracks with a text column named title and an integer column named plays.

Now that we have created a table named Tracks, we can put some data into that table using the SQL INSERT operation:

The SQL INSERT command indicates which table we are using and then defines a new row by listing the fields we want to include (title, plays) followed by the VALUES we want placed in the new row. We specify the values as question marks (?, ?) to indicate that the actual values are passed in as a tuple ('My Way', 15) as the second parameter to the execute() call.

```
import sqlite3
```

```
conn = sqlite3.connect('music.sqlite') 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')

cur.close()
```

First we INSERT two rows into our table and use commit() to force the data to be written to the database file.

Then we use the SELECT command to retrieve the rows we just inserted from the table. On the SELECT command, we indicate which columns we would like (title, plays) and indicate which table we want to retrieve the data from. After we execute the SELECT statement, the cursor is something we can loop through in a for statement. For efficiency, the cursor does not read all of the data from the database when we execute the SELECT statement. Instead, the data is read on demand as we loop through the rows in the for statement.

The output of the program is as follows:

```
Tracks:

('Thunderstruck', 20)

 ('My Way', 15)
```

Our for loop finds two rows, and each row is a Python tuple with the first value as the title and the second value as the number of plays.

At the very end of the program, we execute an SQL command to DELETE the rows we have just created so we can run the program over and over. The DELETE command shows the use of a WHERE clause that allows us to express a selection criterion so that we can ask the database to apply the command to only the rows that match the criterion.

Spidering Twitter using a database

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

one person's Twitter friends are processed, the database is checked and one of the friends of the friend is retrieved. 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.

It is also used to track how many times a particular friend in the database has been seen to get some sense of their "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.

```
from urllib.request
import urlopen
import urllib.error
import twurl
import  json
import sqlite3
import ssl
TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'
conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute(''' CREATE TABLE IF NOT EXISTS Twitter (name TEXT,
retrieved INTEGER, friends INTEGER)''')
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
while True:
    acct = 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': '5'})
print('Retrieving', url)
connection = urlopen(url, context=ctx)
data = connection.read().decode()
headers = dict(connection.getheaders())
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. 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 1 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, 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.

Basic data modeling

The real power of a relational database is when we create 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 tables is called data modeling. The design document that shows the tables and their relationships is called a data model.

In Twitter spider application, instead of just counting a person's friends, if 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 one of 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 order to avoid redundancy of data, the twitter information can be stored in a table called People.

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)

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.

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.

Programming with multiple tables

```
import urllib.request, urllib.parse, urllib.error

import twurl

import json

 import sqlite3

 import ssl

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

conn = sqlite3.connect('friends.sqlite')

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))''')

# Ignore SSL certificate errors
```

```
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
while True:
    acct = 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': '100'}) print('Retrieving account', acct)
try:
     connection = urllib.request.urlopen(url, context=ctx)
except Exception as err:
      print('Failed to Retrieve', err)
```

```
        break
data = connection.read().decode()
headers = dict(connection.getheaders())
print('Remaining', headers['x-rate-limit-remaining'])
try:
    js = json.loads(data)
except:
    print('Unable to parse json')
    print(data)
    break
print(json.dumps(js, indent=4))
if 'users' not in js:
            print('Incorrect JSON received')
            print(json.dumps(js,  indent=4))
            continue
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)

print('Remaining', headers['x-rate-limit-remaining'])

conn.commit()

cur.close()
```

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))''')
```

The name column in the People table must be UNIQUE. 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.

These constraints can be made use of as follows:

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

The OR IGNORE clause is added to the 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.

Three kinds of 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 the 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.

Using JOIN to retrieve data

SQL uses the JOIN clause to reconnect tables. In the JOIN clause specifies the fields that are used to reconnect 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 "metarows" 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 metarow for each of the matching pairs of rows, duplicating data as needed.

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')

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.to_id = People.id WHERE Follows.from_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()
```