Python IO and Database APIs

Christopher Simpkins
chris.simpkins@gatech.edu

Outline

- File I/O
  - Comma-Separated Value Files
- XML Parsing
  - SAX Parsers
  - DOM Parsers
  - ElementTree - the Pythonic way
- Working with databases
  - Connections
  - Cursors
  - Statements
  - Transactions

Text File IO

- File IO is done in Python with the built-in `File` object which is returned by the built-in `open` function
- Use the `'w'` open mode for writing

```
$ python
>>> f = open("hello.txt",'w') # opens hello.txt for writing, creating file if necessary
>>> f.write("Hello, file!\n") # writes string to file -- notice the \n line ending
>>> f.close()                 # closes the file, causing it to be written to disk
>>> exit()
$ cat hello.txt
Hello, file!
```

- Use the `'r'` open mode for reading

```
$ python
>>> f = open("hello.txt", "r") # open hello.txt for reading in text mode
>>> contents = f.read()        # read() slurps the whole file into memory
>>> contents
'Hello, file!\n'
>>> exit()
```

Reading Lines from Text Files

- Text files often have data split into lines
- the `readlines()` function reads all lines into memory as a list

```
>>> f = open('lines.txt', 'r')
>>> f.readlines()
['line 1\n', 'line 2\n', 'line 3\n']
```

- `readline()` reads one line at a time, then returns empty string when file is fully read
- re-open file or use `seek()` to go back to beginning of file

```
>>> f = open('lines.txt', 'r')
>>> f.readline()
'line 1\n'
>>> f.readline()
'line 2\n'
>>> f.readline()
'line 3\n'
>>> f.readline()
''
>>> f.seek(0)
>>> f.readline()
'line 1\n'
```

Processing Lines in a Text File

- Could use `readlines()` and iterate through list it returns

```
>>> f = open('lines.txt', 'r')
>>> for line in f.readlines():
...     print line
...
line 1
line 2
line 3
```

- Better to use the built-in file iterator

```
>>> for line in open('lines.txt', 'r'):
...     print line
...
line 1
line 2
line 3
```

Comma-Separated Value Files

- Say we have data in a comma-separated value file

```
$ cat capitals.dat # notice .dat extension instead of .txt - still text, but structured
Japan,Tokyo
France,Paris
Germany,Berlin
U.S.A.,Washington, D.C
```

- Can use line-by-line file reading with the split() function we saw earlier to process comma-separated value files

```
$ python
>>> capitals = {} # initialize a dictionary to hold our capitals data
>>> for line in open('capitals.dat', 'r'): # for each line in the data file,
...     k, v = line.split(',')             # split line into key and value
...     capitals[k] = v                    # add key: value to capitals dictionary
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: too many values to unpack
```

- Why didn't it work?

CSV Separator Characters

- We can troubleshoot in the Python interpreter

```
>>> for line in open('capitals.dat', 'r'):
...     print line.split(',')
...
['Japan', 'Tokyo\n']
['France', 'Paris\n']
['Germany', 'Berlin\n']
['U.S.A.', 'Washington', ' D.C\n']
```

- There’s a comma in Washington, D.C. that was taken as a separator
- So let’s change the capitals.dat file to use semicolons as the separators

```
$ cat capitals.dat
Japan;Tokyo
France;Paris
Germany;Berlin
U.S.A.;Washington, D.C
```

CSV Files in Practice

- Now our capitals.dat file is readable as a “comma”-separated value file

```
>>> capitals = {}
>>> for line in open('capitals.dat', 'r'):
...     k, v = line.split(';')
...     capitals[k] = v
...
>>> capitals
{'Japan': ' Tokyo\n', 'U.S.A.': ' Washington, D.C\n', 'Germany': ' Berlin\n', 'France': '
Paris\n'}
```

- But the values have leading whitespace and trailing ‘\n’ characters from the data file
- We can make our processor more robust with `strip()`, which removes leading and trailing whitespace and non-printing characters

```
>>> for line in open('capitals.dat', 'r'):
...     k, v = line.split(';')
...     capitals[k.strip()] = v.strip()
...
>>> capitals
{'Japan': 'Tokyo', 'U.S.A.': 'Washington, D.C', 'Germany': 'Berlin', 'France': 'Paris'}
```

XML Parsing

- CSV files are simply structured text-based data files
- XML files are more highly structured text-based data files, allowing nested, or “tree” structured data to be represented in text
- An XML Parser reads an XML file and extracts its structure. Three kinds of XML parsers:
  - SAX (Simple API for XML): a state machine that produces events for each element read from an XML file. Parser responds to events to process these elements
  - DOM (Document Object Model): the DOM standard specifies an object-tree representation of an XML document
    - Tip: In Google Chrome you can see the DOM tree of any web page by clicking View->Developer->Developer Tools in the menu and clicking the Elements tab in the lower pane. Expand elements of the DOM tree and cursor over them to see the rendered parts of the page they represent. In FireFox you can install a plug-in to do the same thing
- We’ll use ElementTree, which is essentially a Python-specific DOM parser

ElementTree

- An `ElementTree` is a representation of an XML document as a tree of `Element` objects with easy to use methods:
  - `parse("fileName")` reads an XML file and create an ElementTree representation of its document
  - `find("elementName")` gets a single child of an element
  - `findall("elementName")` gets an iterator over the like-named children of an element
- That's it. Really. It's that simple.

A Complete XML Parsing Example

- Say we have an XML file named `people.xml`

```
<?xml version="1.0"?>

<people>
  <person>
    <firstName>Alan</firstName>
    <lastName>Turing</lastName>
    <profession>Computer Scientist</profession>
  </person>
  <person>
    <firstName>Stephen</firstName>
    <lastName>Hawking</lastName>
    <profession>Physicist</profession>
  </person>
</people>
```

- Our XML document will have a root element named `people` and child elements named `person`
- Each person element will have three child elements named `firstName`, `lastName`, and `profession`

Parsing with ElementTree

- Parsing `people.xml` with `ElementTree` is this easy:

```
>>> import xml.etree.ElementTree
>>> people = xml.etree.ElementTree.parse("people.xml")
>>> persons = people.findall("person")
>>> for person in persons:
...     print person.find("firstName").text
...     print person.find("lastName").text
...     print person.find("profession").text
...
Alan
Turing
Computer Scientist
Stephen
Hawking
Physicist
>>>
```

Working with Databases

- Connection objects represent a connection to a database
  - provide transaction methods `rollback()` and `commit()`
  - provide a cursor object via `cursor()` to access the database
- Cursor object is a pointer to a part of the database
  - Connection's `cursor()` method returns a pointer to the database itself on which we can then execute statements
- SQL Statements are submitted to the `execute()` method of a database cursor
  - the execute method returns a cursor to its result
  - If the statement was a select, then the cursor can return the rows as a Python list of tuples via its `fetchall()` method

SQLite Databases

- Say we have the following database schema in `people-create.sql`:

```
create table if not exists person (
  person_id integer primary key autoincrement,
  first_name text,
  last_name text,
  profession text
);
```

- And we create an empty SQLite database with it:

```
$ sqlite3 people.sqlite3
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .read people-create.sql
sqlite> .exit
$ ls
people.sqlite3    people-create.sql
```

Inserting Data into an SQLite Database

- We can insert data into the database from within a Python program
- Use the cursor object to execute an SQL insert statement
- The insert statement uses ? markers for the values, and the values are supplied with a tuple

```
>>> import sqlite3
>>> conn = sqlite3.connect("people.sqlite3")
>>> curs = conn.cursor()
>>> curs.execute("insert into person values (?, ?, ?, ?)", (1, "Andy", "Register", "EE"))
<sqlite3.Cursor object at 0x1004dd1e8>
>>> curs.rowcount
1
>>> conn.commit()
```

- The `rowcount` attribute of the cursor object indicates how many rows of the database were affected
- The `commit()` method on the connection object causes the data to be written to the database. `rollback()` would undo all changes since the last `commit()`
- Notice we had to supply the primary key even though SQLite could auto-generate it. The Python database module doesn't know about autoincrement fields

Getting Data Out of an SQLite Database

- We can get data with an SQL select query
- After executing a select, the cursor's `fetchall()` method returns the results as a list of tuples

```
>>> curs.execute("insert into person values (?, ?, ?, ?)", (2, "Carlee", "Bishop", "Sys
Eng"))
<sqlite3.Cursor object at 0x1004dd1e8>
>>> conn.commit()
>>> curs.execute("select * from person")
<sqlite3.Cursor object at 0x1004dd1e8>
>>> for row in curs.fetchall():
...     print(row)
...
(1, u'Andy', u'Register', u'EE')
(2, u'Carlee', u'Bishop', u'Sys Eng')
>>>
```

A Complete Example: Reading XML Data into a Database

- Say we have a system that receives data externally in the form of XML files and inserts the data from them into a database
- We can read the data from the XML file using ElementTree parsing shown earlier
- We can insert the data into our database using the database APIs we just saw
- Assume we have an empty database created with the people-create.sql file we saw earlier
- Let’s write a program to read an XML file, extract the data from it, and insert the data into the database

Inserting Data into the Database

- First, we'll write a function to insert persons into the database in `people.py`

```
import sqlite3

def insert_person(db_conn, person_id, first_name, last_name, profession):
    curs = db_conn.cursor()
    curs.execute("insert into person values (?, ?, ?, ?)",
                 (person_id, first_name, last_name, profession))
    conn.commit()
```

- We can test our program in progress by importing it into the Python interactive shell

```
>>> import people
>>> import sqlite3
>>> conn = sqlite3.connect("people.sqlite3")
>>> people.insert_person(conn, 1, "George", "Burdell", "Student")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "people.py", line 7, in insert_person
    conn.commit()
NameError: global name 'conn' is not defined
```

- Oops. It's db_conn, not conn

Debugging Cycle

- So we fix our function:

```
import sqlite3

def insert_person(db_conn, person_id, first_name, last_name, profession):
    curs = db_conn.cursor()
    curs.execute("insert into person values (?, ?, ?, ?)",
                 (person_id, first_name, last_name, profession))
    db_conn.commit()
```

- Recreate our empty database:

```
$ rm people.sqlite3; sqlite3 people.sqlite3 < people-create.sql
```

- And try our (hopefully) fixed function:

```
$ python
>>> import people
>>> import sqlite3
>>> conn = sqlite3.connect("people.sqlite3")
>>> people.insert_person(conn, 1, "George", "Burdell", "Student")
>>> conn.cursor().execute("select * from person").fetchall()
[(1, u'George', u'Burdell', u'Student')]
```

- Now insert_person() works. On to XML extraction

person_data_from_element

- Now let’s add a function to extract person data from an XML element:

```
import xml.etree.ElementTree

def person_data_from_element(element):
    first = element.find("firstName").text
    last = element.find("lastName").text
    profession = element.find("profession").text
    return first, last, profession
```

- And test in the Python shell:

```
$ python
>>> import people
>>> import xml.etree.ElementTree
>>> peeps = xml.etree.ElementTree.parse("people.xml")
>>> first_person = peeps.findall("person")[0]
>>> people.person_data_from_element(first_person)
('Alan', 'Turing', 'Computer Scientist')
```

- Note that we called the XML element `peeps` so it wouldn’t clobber the `people` module name

The Complete Program

- Now we add the main loop to get all the persons from the XML file and our program is complete

```
import sqlite3
import xml.etree.ElementTree

def insert_person(db_conn, person_id, first_name, last_name, profession):
    curs = db_conn.cursor()
    curs.execute("insert into person values (?, ?, ?, ?)",
                 (person_id, first_name, last_name, profession))
    db_conn.commit()

def person_data_from_element(element):
    first = element.find("firstName").text
    last = element.find("lastName").text
    profession = element.find("profession").text
    return first, last, profession

if __name__ == "__main__":
    conn = sqlite3.connect("people.sqlite3")
    people = xml.etree.ElementTree.parse("people.xml")
    persons = people.findall("person")
    for index, element in enumerate(persons):
        first, last, profession = person_data_from_element(element)
        insert_person(conn, index, first, last, profession)
```

Our Program in Action

```
$ rm people.sqlite3; sqlite3 people.sqlite3 < people-create.sql
$ python people.py
$ sqlite3 people.sqlite3
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from person;
0|Alan|Turing|Computer Scientist
1|Stephen|Hawking|Physicist
sqlite>
```

- It works!
- But it's lame to have to manually insert primary keys

Determining a Safe Primary Key

- If we insert a record with a duplicate key, it will fail
- If we know keys are integers, we can get the integers and add one to the highest one
  - Note: can't just use size of table - key values may be greater due to inserts and deletes
- Here's how we can do it in Python:

```
>>> import sqlite3
>>> conn = sqlite3.connect("people.sqlite3")
>>> curs = conn.cursor().execute("select person_id from person")
>>> results = curs.fetchall()
>>> results
[(0,), (1,)]
>>> keys = [tuple[0] for tuple in results]
>>> keys
[0, 1]
>>> keys.sort()
>>> largest_key = keys[len(keys) - 1]
>>> largest_key
1
```

A More Robust Program

- This version doesn't require an empty database

```
import sqlite3
import xml.etree.ElementTree

def insert_person(db_conn, first_name, last_name, profession):
    curs = db_conn.cursor()
    curs.execute("select person_id from person")
    keys = [t[0] for t in curs.fetchall()]
    keys.sort()
    if len(keys) > 0:
        largest_key = keys[len(keys) - 1]
    else:
        largest_key = 0
    curs.execute("insert into person values (?, ?, ?, ?)",
                 (largest_key + 1, first_name, last_name, profession))
    db_conn.commit()

def person_data_from_element(element):
    first = element.find("firstName").text
    last = element.find("lastName").text
    profession = element.find("profession").text
    return first, last, profession

if __name__ == "__main__":
    conn = sqlite3.connect("people.sqlite3")
    people = xml.etree.ElementTree.parse("people.xml")
    persons = people.findall("person")
    for element in persons:
        first, last, profession = person_data_from_element(element)
        insert_person(conn, first, last, profession)
```

Conclusion

- We learned
  - Basic text file I/O
  - How to work with comma-separated value files
  - How to parse XML
  - How to work with databases
  - How to connect XML data and databases using Python programming
- Lots more to know, but we saw the basics
- Consult Python docs or books for more details