Python for Informatics

LESSON 7

Using Web Services

- In our previous lesson, we learned how to use Python to create a simple web browser application that we used to connect to web servers and parse web pages by using the HTTP protocol.
- Web pages are encoded in Hyper Text Manipulation Language (HTML).

Using Web Services

- HTML is not a programming language.
- HTML enables the decription of how a web page is formatted, and how it will appear.
- The ultimate consumer of an HTML file (web page) is a human being.

Using Web Services

- Relatively soon after the development of the Web, we realized that we needed a format definition language similar to HTML that was not limited to describing ***how things should appear***.
- The language would allow for the definition of ***what things are***, and ***what things mean.***
- The language would need to be ***extensible***, such that anything could be defined in whatever way is useful.

Using Web Services

- A language that can only define how things appear (HTML) is ultimately consumed by a viewer (a human being).
- A language that defines what things are and what they mean can be consumed by any information processing system (computational machine).

Using Web Services

- ***eXtensible Markup Language (XML)*** is such a language.
- ***XML*** supports the intercomputer exchange of diverse data descriptions.

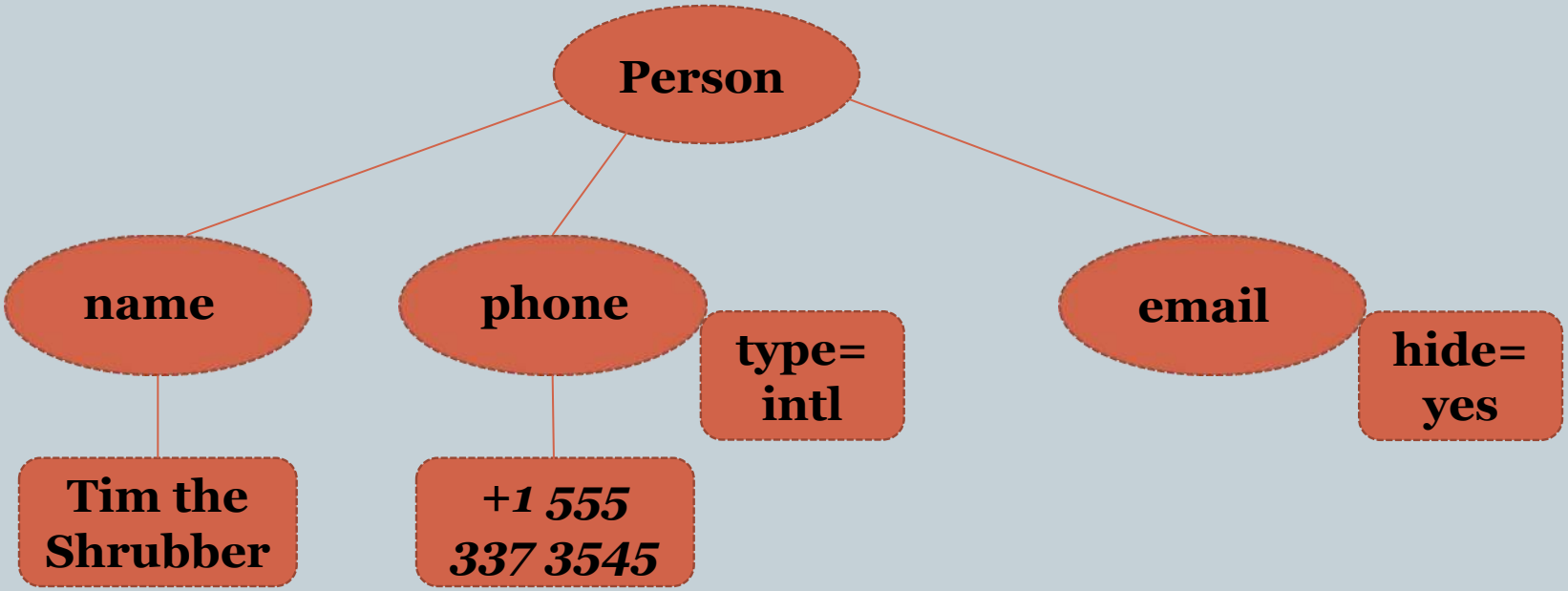
Using Web Services

- Here is an example of an ***XML document***:

```
<person>
   <name>Tim the Shrubber</name>
   <phone type="intl">
         +1 555 337 3545
   </phone>
   <email hide="yes"/>
<person>
```

Using Web Services

- It is useful to view an XML document as an upside-down tree-like structure:



Using Web Services

- Python has an XML parser named ***ElementTree***, that facilitates the navigation of XML structures and the extraction of data contained in those structures.
- The code in the following slide demonstrates how to use ***ElementTree*** to extract XML data.

Using Web Services


```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Tim the Shrubber</name>
  <phone type="intl">
     +1 555 337 3545
   </phone>
   <email hide="yes"/>
</person>'''

tree = ET.fromstring(data)
print 'Name:',tree.find('name').text
print 'Attr:',tree.find('email').get('hide')
```

```
Name: Tim the Shrubber
Attr: yes
```

Using Web Services


```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Tim the Shrubber</name>
  <phone type="intl">
    +1 555 337 3545
  </phone>
  <email hide="yes"/>
</person>'''
```

- Above, we import the ElementTree module, and we use the as keyword to establish an alias for our module that is easier to type.
- Then we create a ***str*** object with XML data.
- Note that the triple quotes allow us to specify a ***str*** with embedded newline characters.

Using Web Services


```
tree = ET.fromstring(data)
print 'Name:',tree.find('name').text
print 'Attr:',tree.find('email').get('hide')
```

- The ***fromstring()*** method converts the str parameter and returns an ***XML tree*** of ***XML nodes***.
- The ***find()*** method traverses the ***XML tree*** and returns the ***XML node*** that matches the specified ***XML tag***.
- Each node can contain text, some attributes (such as ***hide***), and some child nodes.

Using Web Services

- Often we need to process multiple ***element nodes*** within an ***XML document***.
- This can be done easily by using the ***ElementTree findall()*** method, and then a ***for*** loop to iterate through the ***list*** that is returned.

Using Web Services


```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x='42'>
      <id>001</id>
      <name>Tim the Shrubber</name>
    </user>
    <user x='23'>
      <id>009</id>
      <name>Sir Galahad</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')
```

```
User count: 2
Name Tim the Shrubber
Id 001
Attribute 42
Name Sir Galahad
Id 009
Attribute 23
```

Using Web Services

- ***JavaScript Object Notation (JSON)*** is an alternative techology supporting the exchange of data between applications and between computers.
- Although ***JSON*** was developed for ***JavaScript***, it is now used as a data exchange standard across many languages.
- Since the structure of ***JSON*** was inspired by ***Python***, using ***JSON*** with ***Python*** seems natural and consistent.

Using Web Services

- Here is an example of a ***JSON encoding***:

```
{
   "name" : "Tim the Shrubber"
   "phone" : {
      "type" : "intl",
      "number" : "+1 555 337 3545"
   },
   "email" : {
       "hide" : "yes"
   }
}
```

Using Web Services

- Our sample ***JSON*** encoding differs from the sample ***XML*** encoding in that…
  - ***XML*** adds attributes like ***intl*** to the ***phone*** tag, whereas ***JSON*** simply uses ***key-value pairs***.
  - The person ***XML*** tag is replaced by a pair of ***enclosing curly braces***.
- ***XML*** is more complex, allowing deep and nuanced data definitions, and ***meta-definitions***.
- ***JSON*** is relatively simple, and maps directly to ***lists***, ***dictionaries***, and combinations thereof.
- Unless there's a need for the complexity of ***XML***, ***JSON*** is generally preferred.

Using Web Services

- To construct a JSON structure, we merely nest a combination of lists and dictionaries in the way that best suits our needs.
- For example, if we need a list of users, where each user is a set of key-value pairs, then we have a list of dictionaries.

Using Web Services


```
import json

input = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Chuck"
  }
]'''

info = json.loads(input)
print 'User count:', len(info)

for item in info:
    print 'Name', item['name']
    print 'Id', item['id']
    print 'Attribute', item['x']
```

Using Web Services

- JSON allows us to quickly translate our data into convenient native Python structures (a ***list of dictionaries***, in this case).
- Convenient simplicity is the strength of JSON.

Using Web Services

- ***JSON*** structures have no way of describing themselves the way that ***XML*** can.
- Therefore, we need to know ahead of time, what the structures are that we are working with—our code is written with fore-knowledge of the input data structures.
- The lack of meta-description is a weakness of ***JSON***.

Using Web Services

- Instead of using ***HTTP*** with ***HTML*** data, we now know enough to consider how we might use ***HTTP*** with ***XML*** or ***JSON*** data.
- Establishing a contract between applications as to how to exchange data becomes the basis for a set of services.

Using Web Services

- A ***contract*** between a ***service application*** and a ***client application*** is known as an ***Application Program Interface (API)***.
- The approach of using a set of services that applications provide to each other based on their respective responsibilities is known as ***Service-Oriented Architecture (SOA)***.

Using Web Services

- The idea of ***SOA*** is to code your application such that it makes use of the sets of services provided by other applications.
- Examples of ***SOA*** include web sites that assist in the booking of air travel, hotels, and rental cars.
- Although you may think that you are using only one site, the ***SOA application*** is pulling and pushing information from/to many other sites via the ***web services*** that they provide.

Using Web Services

- Amazon offers a ***web service*** that allows its inventory to be queried programmatically.
- Google offers a ***web service*** that provides access to its vast database of geographic information.
- To explore the use of SOA and web services, we will make use of Google's free ***geocoding web service***.

Using Web Services

- Although Google provides its ***geocoding web service*** for free, it puts a rate limit on it.
- Please do not abuse the use of this web service, as doing so might motivate Google to curtail its availability.

Using Web Services

- You can read more about this web service at the following URL:

  https://developers.google.com/maps/documentation/geocoding/intro

- To try the service out, quickly and easily, just click on this web link:

  http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=San+Diego+CA

Using Web Services

- Let's look at a simple Python app that prompts the user for a search str, calls the Google geocoding API, and extracts data from the returned JSON data structures.

```
import urllib
import json

serviceurl =
'http://maps.googleapis.com/maps/api/geocode/json?'

while True:
    address = raw_input('Enter location: ')
```

Using Web Services


```
if len(address) < 1 : break

   url = serviceurl + urllib.urlencode({'sensor':'false', 'address': address})
   print 'Retrieving', url
   uh = urllib.urlopen(url)
   data = uh.read()
   print 'Retrieved',len(data),'characters'

   try: js = json.loads(str(data))
   except: js = None
   if '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
```

Using Web Services

- The program accepts a search ***str*** as input from the user, and uses ***urllib*** to retrieve ***JSON*** formatted text from the Google ***geocoding API***.
- It then parses the retrieved JSON by using the ***json*** library, performs some simple validation, and then extracts the data we are looking for.

Using Web Services


```
Enter location: San Diego, CA
Retrieving
http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=San
+Diego%2C+CA
Retrieved 1738 characters
{
   "status": "OK",
   "results": [
      {
         "geometry": {
            "location_type": "APPROXIMATE",
            "bounds": {
               "northeast": {
                  "lat": 33.114249,
                  "lng": -116.90816
               },
               "southwest": {
                  "lat": 32.534856,
                  "lng": -117.2821665
               }
```

Using Web Services


```
   },
   "viewport": {
      "northeast": {
         "lat": 33.114249,
         "lng": -116.90816
      },
      "southwest": {
         "lat": 32.534856,
         "lng": -117.2821665
      }
   },
   "location": {
      "lat": 32.715738,
      "lng": -117.1610838
   }
},
```

Using Web Services


```
"address_components": [
          {
            "long_name": "San Diego",
            "types": [
              "locality",
              "political"
            ],
            "short_name": "San Diego"
          },
          {
            "long_name": "San Diego County",
            "types": [
              "administrative_area_level_2",
              "political"
            ],
            "short_name": "San Diego County"
          },
```

Using Web Services


```
{
            "long_name": "California",
            "types": [
               "administrative_area_level_1",
               "political"
            ],
            "short_name": "CA"
         },
         {
            "long_name": "United States",
            "types": [
               "country",
               "political"
            ],
            "short_name": "US"
         }
      ],
```

Using Web Services


```
 "place_id": "ChIJSx6SrQ9T2YARed8V_fohOgo",
          "formatted_address": "San Diego, CA, USA",
          "types": [
             "locality",
             "political"
          ]
       }
    ]
}
lat 32.715738 lng -117.1610838
San Diego, CA, USA

Enter location:
```

Using Web Services

- ***API keys*** are often required while using a ***web service*** so the vendor can track who is using their service and how much they are using it.
- Using an assigned ***API key*** might require supplying it as part of POST data or as a parameter within the ***URL*** when the ***web service*** is invoked.

Using Web Services

- Some ***web services*** require greater security, such as the use of cryptographically signed messages with shared keys and secrets.
- ***OAuth*** is an internet resource that signs ***secure signature requests***.

http://www.oauth.net

Using Web Services

- ***Twitter*** is an especially popular and unique web service that requires ***OAuth*** signatures for each request.
- As an example of the programmatic use of the ***Twitter API***, locate the textbook files ***twurl.py***, ***hidden.py***, ***oauth.py***, and ***twitter1.py*** and place them in a common directory.

Using Web Services

- To run the example, you must have a Twitter account, and you must authorize your Python code as an application.
- Important parameter values you must supply include: ***key***, ***secret***, ***token***, ***token secret***.

Using Web Services

- ***hidden.py*** must be edited to contain your parameter values. For example:

```
# Keep this file separate

def oauth() :
    return { "consumer_key" : "h7Lu...Ng",
        "consumer_secret" :
"dNKenAC3New...mmn7Q",
        "token_key" : "10185562-
eibxCp9n2...P4GEQQOSGI",
        "token_secret" :
"HoycCFemmC4wyf1...qoIpBo" }
```

Using Web Services

- You then make use of ***oauth.py***, ***twurl.py***, and ***twitter1.py*** to access the ***Twitter web service***.

- See our textbook for further guidance and details.

Databases and Structured Query Language

- ***Databases*** are a bigger topic than regular expressions.
- We will barely scratch the surface of what can be learned regarding ***databases***.
- We will learn enough, however, to be able to connect to and query a ***database*** with Python.

Databases and Structured Query Language

- ***Databases*** are structured in a way that is similar to Python dictionaries in that they provide ***key-value mappings***.
- Whereas ***dictionaries*** reside in volatile memory, databases reside on a persistent storage medium, usually disk storage.

Databases and Structured Query Language

- ***Database system software*** is designed and implemented to be extremely fast.
- The insertion and retrieval of data is highly optimized.
- The use of ***indexes*** is an important aspect of how database systems are made for speed.

Databases and Structured Query Language

- Well-known examples of ***Relational Database Systems*** include:
  - Oracle
  - MySQL
  - Microsoft SQL Server
  - PostgreSQL
  - SQLite
- We will be using SQLite.

Databases and Structured Query Language

- SQLite is provided as a Python built-in.
- As its name implies, SQLite has a small form factor—it is designed to be embedded inside of other applications.
- The Firefox web browser has SQLite as an interal data management resource.

Databases and Structured Query Language

- You can explore the documentation on SQLite at this URL:

  http://sqlite.org/

- SQLite can easily be used to support informatics applications such as the Twitter spidering application that we will soon develop.

Databases and Structured Query Language

- A database is composed of one or more tables.
- Each table contains zero or more rows.
- Each row contains one or more columns.

Databases and Structured Query Language

- Database theory is based upon Math.

- There are theoretical terms that are also commonly used.
  - A ***table*** is also known as a ***relation***.
  - A ***row*** is also known as a ***tuple***.
  - A ***column*** is also known as an ***attribute***.

Databases and Structured Query Language

- One surprisingly accurate way of visualizing what a ***database table*** is like is to think of it as being an ***Excel spreadsheet***.
- Although we will be using Python to operate upon a ***database***, it is often useful to use a ***database manager application*** to help us create, configure, and initialize a database.
- The ***Firefox*** browser has a simple ***SQLite database manager add-on***. You can get from here:

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

Databases and Structured Query Language

- Creating and defining a ***database table*** is more involved than creating a Python ***dictionary***.
- We must specify the name of each column, and the data type of the values that will be stored in each column.
- The data types available in SQLite are described here:

  http://www.sqlite.org/datatype3.html

Databases and Structured Query Language

- Although defining ***table*** structures is a bit of a pain, the payoff is that you will enjoy fast storage and retrieval of your data.
- In Python, we can create a ***database file*** with a ***table*** named ***Tracks*** with two ***columns*** with the following code:

Using Web Services


```
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()
```

Databases and Structured Query Language

- The code works much as you would expect by reading it.
- The ***connect()*** method establishes a ***connection*** to the ***database file*** by the given name.
- If the file doesn't exist, it will be created.
- Our database will be local.
- Databases are often set up to run on remote servers.

Databases and Structured Query Language

- The ***cursor()*** method returns a ***cursor*** object, which acts like a ***file handle*** does with a file.
- Our ***cursor*** gives us the means to start issuing commands to our database.
- The commands we will issue are part of a standardized language especially designed for relational databases known as ***Structured Query Language (SQL).***
  http://en.wikipedia.org/wiki/SQL

Databases and Structured Query Language


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

conn.close()
```

- The first command (***DROP TABLE***) deletes the ***Tracks*** table if it exists.
- The second command (***CREATE TABLE***) creates a ***table*** named ***Tracks*** with two columns, a ***text column*** named ***title*** and an ***integer column*** named ***plays***.
- The ***close()*** method closes our ***connection*** to the ***database***.

Databases and Structured Query Language

- Now that we have our ***Tracks*** table created, we can insert some data into it by means of the ***INSERT*** command.
- The ***INSERT*** command specifies the ***table*** and the various ***field*** values of the ***row*** we want to insert.
- Of course, we need to first establish a ***connection*** with the ***database*** and obtain our ***cursor***.

Databases and Structured Query Language


```
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()
```

Databases and Structured Query Language

- The ***INSERT*** commands insert two new ***rows*** into our ***Tracks table***.

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

- The ***fields*** are specified, and then the ***values***. We specify "***?***" for the ***values*** to indicate that the succeeding ***tuple*** contains the actual ***values***.
- The ***COMMIT*** command writes the data to the ***database***.

title	plays
Thunderstruck	*20*
My Way	*15*

Databases and Structured Query Language

- The ***SELECT*** command retrieves the specified ***fields*** of all ***rows*** from the ***Tracks table***.
- The data returned by the ***SELECT*** command is called a ***result set***.
- We use our ***cursor*** to iterate through the ***result set***, and print each ***row***.

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

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

- The **u'** indicates that the strings are unicode, which supports an extensive non-Latin character set.

Databases and Structured Query Language

- ***SQL*** provides a where clause that allows us to specify that operations are done only when a certain condition is true.

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

- The above ***SQL*** statement will ***delete*** all of the ***rows*** in the ***tracks table*** that have a ***plays*** value that is less than 100.

Databases and Structured Query Language

- Becoming adept with ***SQL*** takes time and practice.
- Here are some great resources on the web that can facilitate your learning.

http://www.w3schools.com/sql/

https://www.digitalocean.com/community/tutorials/a-basic-mysql-tutorial

http://www.elated.com/articles/mysql-for-absolute-beginners/

http://sqlzoo.net/wiki/SELECT_basics

http://www.yolinux.com/TUTORIALS/LinuxTutorialMySQL.html

http://www.tutorialspoint.com/mysql/mysql-where-clause.htm

http://www.devshed.com/c/a/MySQL/Beginning-MySQL-Tutorial/

Databases and Structured Query Language

- Understanding ***databases*** requires more than understanding ***SQL***.
- To design a ***database*** solution properly, we need to understand ***data modeling***.

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

- A ***relational database*** solution typically involves the creation of multiple ***tables***.
- The **tables** are ***linked*** together by means of ***keys***.

Databases and Structured Query Language

- There are three kinds of ***keys***:
  - ***Logical key*** – A real world ***key*** used to uniquely identify a ***row*** within a ***table***.
  - ***Primary key*** – A unique ***key*** that has no real world meaning, but serves only to identify a ***row*** within a ***table***.
  - ***Foreign key*** – A ***key*** that identifies an associated ***row*** within another ***table***.