Lab 4: Internet-Based Systems with Python

Due Friday September 30, 2015 @ 5pm PT on bCourses

Part 1: Python and the Internet

Exercise 1: Program Structure

Deliverables (0.5):

- In your lab report, respond to Questions.

Directions:

1. Open and run the ProgramStructure.py. Use Ctrl+C to close the program. Make sure you understand what the program is doing.
2. **Questions**:
    a. How is this program structure similar to the one used by Arduino?
    b. How is it different?

Exercise 2: Date and Time

Deliverables (1 pt):

- In your lab report, respond to Questions.

Directions:

1. Open and run DatetimeModule.py. Make sure you understand what the program is doing.
2. **Questions**:
    a. Using Python, convert the following to UNIX time:
        i. 2015-08-26T14:00:00.000000Z
        ii. 12 noon (UTC/GMT) on July 4th, 2010.
    b. Using Python, convert the following from UNIX time to an ISO 8601 UTC timestamp (YYYY-MM-DDTHH:MM:SS.mmmmmmZ). Your response must include the "T" and "Z":
        i. 0
        ii. 1400000000

Exercise 3: HTTP Requests

Deliverables (1 pt):

- Submit a file named MakingHTTPRequests.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.
- In your lab report, respond to Questions.

Directions:

1. In this exercise, we will use the Requests module to make HTTP GET requests to Internet servers. GET requests are most commonly used by web browsers to retrieve the source files of a webpage. In this exercise, we will also make GET requests to a Web API, allowing us to retrieve the information contained in a webpage without the visualizations intended for web browsers.
2. Open the MakingHTTPRequests _skeleton.py file, save it as MakingHTTPRequests.py.
3. We will start by making a GET request to google.com.

```
# Make a GET request for Google.com
address = 'https://www.google.com/'
r = requests.request('get',address)
# Print the content returned by Google
print r.text
```

If you run the program, you will see lots of text printed to the Python console. This text describes the structure, styling, and interactions that allow a web browser to display the Google.com homepage.

4. Next, we will use an HTTP GET request to run a Google search for "berkeley". We will specify the search term using the query string "?q=berkeley". This is effectively the same as opening your web browser and entering "https://www.google.com/search?q=berkeley".
Comment out the last block of code (using # or '''…''') and add the following block of code below it. To add a query string to the HTTP request, we will pass a dictionary to the request() method.

```
# Make a GET request to Google Search
address = 'https://www.google.com/search'
query = {'q':'berkeley'}
r = requests.request('get',address,params=query)
# Print the URL of the request
print 'Request URL:', r.url
# Print the content returned by Google
print r.text
```

Again, you will see lots of text printed to the console. When this information is visualized by a web browser, it is easy to identify the key information. When retrieving the data programmatically, it is much harder to parse the results. This limitation is solved by employing a Web API. The Web API defines a set of operations that a user can call and details the format in which information will be returned. This allows us to write a program that can make a request to a server and consistently parse the results.

5. Finally, we will use the Google Search Web API to search for "berkeley". The search results are returned as a JSON format string, which we can easily convert to a Python dictionary/list using the json module. Immediately, you should find it much easier to understand the response now that the page structure and styling details have been removed.
   Comment out the last block of code (using # or '''…''') and add the following block of code below it.

```
# Using an Application Programming Interface (API)
# Google Example: Results returned in JSON format
endpoint = 'http://ajax.googleapis.com/ajax/services/search/web'
query = {'v':'1.0','q':'berkeley'}
r = requests.request('get',endpoint,params=query)
# r.text is a JSON string.
print '\nJSON Response:\n', r.text
# Convert r.text to Python dictionary/list
content = json.loads( r.text )

results = content['responseData']['results']
# Print first result
print '\nFirst Result (as a Python object):\n',results[ 0 ]
# Open https://www.google.com/search?q=berkeley in a
# web browser and compare the results
```

6. **Questions**:
   a. How many search results are returned by the Google Search Web API?
   b. Do the Google Search Web API results for "berkeley" match the search results at https://www.google.com/search?q=berkeley?

Exercise 4: Cloud Optimization Solver

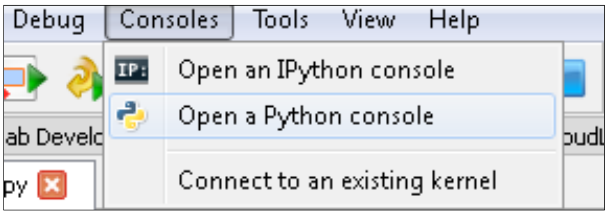
Deliverables (1 pt):

- In your lab report, respond to Questions.

Directions:

1. For this course, we have created a (simple) web-based linear program (LP) solver using the PuLP and Flask modules. We are calling it the **Cloud Optimization Solver**. The app could be deployed to an online platform such as Heroku, but for now, you will run the app locally from your computer.

2. The app requires the PuLP module for solving linear programs using Python. We can install the Pulp (and its dependencies) using pip.
    a. For Windows:
        i. Open cmd
        ii. Type: python –m pip install pulp
    b. For Mac/Linux:
        i. Open Terminal
        ii. Type: python –m pip install pulp
    c. You may need to restart Spyder.
3. To run the app, open the cloudlpapp.py file in Spyder (located in the Code->CloudLPApp folder of the Lab 4 Packet).



4. Open a new Python console (Consoles->Open A Python Console). You should now have 2 (or more) instances of Python running on your computer. We will refer to them as console #1 and console #2.
5. In Python console #2, run the cloudlpapp.py file to start the app on a local server (i.e. your computer). You should see a message similar to the one below. The url (http://127.0.0.1:5000/) is the address at which the app is running. Open the link in a web browser to view the Cloud Optimization Solver's API documentation.

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
INFO:werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

6. Read through the API documentation to learn how to send LP and MILP problems to the app using HTTP POST requests.
7. In Python console #1, run the example problem detailed at the bottom of the API documentation, CloudOptSolverExample.py.
8. **Questions**:
    a. To maximize profits, how many desks and chairs should be manufactured each week?
    b. What is the weekly profit?

Exercise 5: The Knapsack Problem

Deliverables (2.5 pts):

- Submit a file named knapsack.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.
- In your lab report, respond to Questions.

Directions:

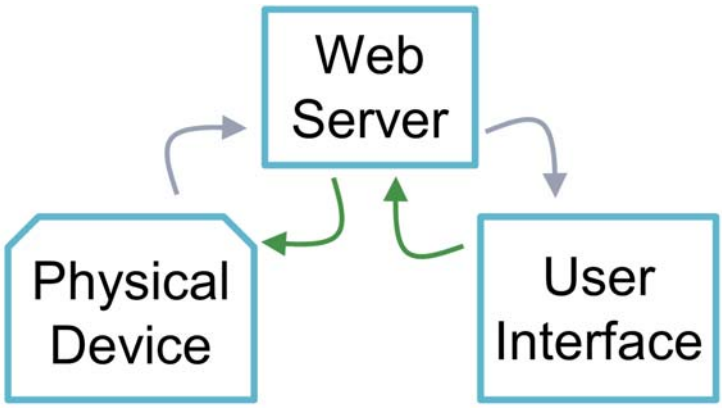
Alexander Supertramp is planning to hitchhike across the United States. He can carry one knapsack and must decide how much food and equipment to bring along. Mr. Supertramp does not have to bring any equipment, but he must carry at least 2 units of food. Also, since carrying equipment is exhausting, the units of equipment cannot be more than double the units of food. His knapsack has a finite volume and he needs your help to maximize the total "value" of the goods he will carry.

Fortunately, you have been working with a linear program solver app and can help him find the optimal solution to his problem.

Symbol	Description	Value
x_1	Units of food (Decision Variable)	---
x_2	Units of equipment (Decision Variable)	---
c_1	Value per unit of food	1
c_2	Value per unit of equipment	3
v_1	Volume per unit of food	2.5
v_2	Volume per unit of equipment	4
K	Maximum volume of knapsack	30

1. **Questions**:
    a. By hand, formulate a mixed-integer linear program (MILP) to solve the knapsack problem. Use the notation from the table above and express the program in the "standard" form. Include the cost function and constraints. Label each constraint to explain its meaning.
    b. Rewrite the MILP formulation in the "matrix" form.
2. Open the knapsack_skeleton.py file and save it as knapsack.py.
3. Complete the program to solve Mr. Supertramp's problem using the Cloud Optimization Solver App. ***You will need to reference the Cloud Optimization Solver API document to determine how to properly interact with the app and to interpret the results***.
4. **Questions**:
    a. What is the integer solution (i.e. x* if x_1 and x_2 are integers)?
    b. What is the relaxed solution (i.e. x* if x_1 and x_2 are floats)?

Part 2: Internet-Based Systems



Exercise 6: Arduino to Python

Items:

- 1x Arduino Uno Boards with USB Cables

Deliverables (1 pt):

- Submit a file named SendData.ino containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.
- Submit a file named ReceiveData.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.

Directions:

1. In this exercise, we will use a serial connection to move data from an Arduino microcontroller to a Python program running on your laptop. Note that we could use wireless transceivers to communicate between the Arduino and your laptop, but for simplicity, we will employ a physical USB connection.
2. To begin, write an Arduino program called SendData.ino that sends data to the hardware serial port. This could be sensor data, a sine wave, etc. Be sure that the data is sent no more than once every 5 seconds. Also, we will use the newline character to indicate the end of a message. You can send the newline symbol by using Serial.println(value), which sends the newline character after the value, or by using Serial.print('\n'). Upload the file and open the Arduino IDE's serial monitor to check that data is being sent.
3. Note the name of the serial port to which the Arduino is connected. In the Arduino IDE, the serial port can be found under Tools->Port.
4. Next, we will write a Python program to receive the incoming data. If you have not already done so, install the pySerial module for communicating with a device over USB.
    a. For Windows:

i. Open cmd
ii. Type: python –m pip install pyserial

b. For Mac/Linux:
i. Open Terminal
ii. Type: python –m pip install pyserial

c. You may need to restart Spyder.

5. Open the ProgramStructure.py file and save it as ReceiveData.py.
6. At the top of the program, import the serial module, specify the port to which the Arduino is connected, and create a serial port object.
   a. For Windows:

```
import serial
# Change the port name to match the port
# to which your Arduino is connected.
serial_port_name = 'COM1' # for Windows
ser = serial.Serial(serial_port_name, 9600, timeout=1)
```

   b. For Mac/Linux:

```
import serial
# Change the port name to match the port
# to which your Arduino is connected.
serial_port_name = '/dev/tty.usbserial-A9007UX1' # for Mac
ser = serial.Serial(serial_port_name, 9600, timeout=1)
```

7. Next, edit the close() function to properly close the serial port when the program is terminated.

```
# Run once at the end
def close():
    try:
        print "Close Serial Port"
        ser.close()
    except:
        print "Close Error"
```

8. Finally, edit the loop() function to receive and print the incoming data.

```
# Run continuously forever
def loop():
    # Check if something is in serial buffer
    if ser.inWaiting() > 0:
        try:
            # Read entire line
            # (until '\n')
            x = ser.readline()
            print "Received:", x
            print "Type:", type(x)

        except:
```

```
            print "Error"

        # 100 ms delay
        time.sleep(0.1)
        return
```

9. Note that the data is being sent by the Arduino and received by the Python program as a character string. If you know that the data is a float or an integer, you can convert x using float() or int(), respectively.

Exercise 7: A Simple Web App

Deliverables (1 pt):

- Submit a file named NameServer.py containing the program described below. Additionally, in your lab report, copy & paste your Python code into a code box.

Directions:

1. Any Internet connected computer can become a web server. This makes it very easy to prototype an Internet-based system using an Arduino microcontroller and your laptop. In this exercise, we will create a simple web application using the Flask Web Framework. See the Flask Quickstart page for an overview of the framework.
2. Open the NameServer_skeleton.py file and save it as NameServer.py.
3. Currently, the app is very simple. At the top, we import Flask and request from the flask module and then create the **app** object. Next, we have code that routes all HTTP requests to the base URL to the root() function, which simply returns the string “Hello”. Finally, at the bottom of the program, we run the app.
4. Run the app in a Python console. You should see a message like the one below.

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
INFO:werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

5. Open a web browser and navigate to http://127.0.0.1:5000/. You should see a webpage containing “Hello”. In the Python console, you will see messages that someone has made GET requests to the server.
    a. Note: The server is being hosted locally at the IP address 127.0.0.1 and is only reachable from your laptop. The port 5000 is the default port for Flask apps.
    b. Use Ctrl+C to close the server.
6. Next, we will add a new “endpoint” to the app. This endpoint will only accept HTTP GET requests and will include a URL variable, **firstname**. For example, if someone navigates to http://127.0.0.1:5000/firstname/Harry, the string “Harry” will be passed to the app as the variable **firstname**.

```
    # Route with URL variable
    @app.route('/firstname/<firstname>', methods=['GET'])
    def firstname(firstname):
        return "Hello "+firstname
```

7. Next, we will add an endpoint with URL variables **firstname** and **lastname**. The endpoint will also accept a query string (key=value pairs separated by &) with optional parameters mi (middle initial) and yr (year). Update the program and try navigating to http://127.0.0.1:5000/firstname/Harry/lastname/Potter?mi=J&yr=1998.

```
# Route with URL variable and query string
# Query parameters: mi, yr
@app.route('/firstname/<firstname>/lastname/<lastname>', methods=['GET'])
def firstlastname(firstname,lastname):

    # Get middle initial
    mi = request.args.get('mi',None)
    if mi is None:
        return "Hello "+firstname+" "+lastname

    # Get class year
    yr = request.args.get('yr',None)
    if yr is None:
        return "Hello "+firstname+" "+mi+'. '+lastname

    return "Hello "+firstname+" "+mi+'. '+lastname+", Hogwarts Class of "+yr
```

8. Finally, we will create a couple endpoints for storing and retrieving names from a database. In this exercise, we will use a SQLite database. SQLite is a serverless SQL engine which stores the contents of a database in a single file. These characteristics are very useful for embedded systems and for prototyping web apps. However, to run an app on a real Internet server, it is generally necessary to use a more powerful SQL database like MySQL or PostgreSQL.

   If you have never worked with or even heard of an SQL database, don't worry. They are very easy to use. Basically, SQL databases store data in tables (rows and columns). Data is added, modified, retrieved, or removed using SQL queries.

   For example, if this exercise, we want to create a database table called "names" which has 2 columns, "firstname" and "lastname". The SQL query to create this table is:

```
CREATE TABLE names (firstname TEXT, lastname TEXT)
```

It is common practice to capitalize SQL terms like CREATE and TABLE. User-defined terms, like names and firstname, are lowercase. When creating a table, we must also specify the data type of each column. The basic data types are INTEGER, REAL, and TEXT.

We can add data (i.e. add a row) to the "names" table using the query below.

```
INSERT INTO names VALUES ('Harry', 'Potter')
```

Finally, to retrieve all of the data from the "names" table, we can use the query below.

```
SELECT * FROM names
```

The three queries above are the very basics of working with any SQL database. In this exercise, we will use the sqlite3 Python module to interact with our SQLite database.

Note: We are going to be using a quick and easy (i.e. "incorrect") approach for adding a database to an app. For a more "correct" approach for integrating SQLite with Flask, see http://flask.pocoo.org/docs/0.10/patterns/sqlite3/#sqlite3.

9. At the top of the program, import sqlite3 and add code to connect to the NameServer database and to create the "names" table.

```
import sqlite3

# Connect to the database
# If .sqlite file does not exist, it will be created.
conn = sqlite3.connect('NameServer.sqlite')
c = conn.cursor()
# Create a table, if it does not exist
c.execute('''CREATE TABLE IF NOT EXISTS names (firstname TEXT, lastname
TEXT)''')
# Save (commit) the changes
conn.commit()
# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

10. Next, add a new endpoint to the app for storing names in the database. This endpoint will only accept HTTP POST requests and will parse a JSON payload. The names will be inserted into the database. If anything goes wrong (e.g. the payload is empty), the app will return {"code":500}.

```
# Route for storing a name in the database
@app.route('/storename', methods=['POST'])
def storefirstlastname():
    try:
        # Parse the JSON payload/body/data
        payload = request.get_json()
        first = payload['firstname']
        last = payload['lastname']

        # Connect to the database
        conn = sqlite3.connect('NameServer.sqlite')
        c = conn.cursor()
        # Add a row to the table
        c.execute('''INSERT INTO names VALUES (?, ?)''', (first,last) )
        # Save (commit) the changes
        conn.commit()
        # We can also close the connection if we are done with it.
        # Be sure any changes have been committed or they will be lost.
        conn.close()

        return jsonify( {"code":200} )

    except:

        return jsonify( {"code":500} )
```

11. Add another endpoint for retrieving names from the database. The endpoint will only accept HTTP GET requests and will return up to 5 names as a JSON string. In the SQL query, “ORDER BY ROWID DESC LIMIT 5” can be interpreted as “find the last 5 rows”.

```
# Route for retrieving names from the database
@app.route('/retrievename', methods=['GET'])
def retrievefirstlastname():
    # Connect to the database
    conn = sqlite3.connect('NameServer.sqlite')
    c = conn.cursor()
    # Get the 5 most recently added names from the table
    c.execute('''SELECT * FROM names ORDER BY ROWID DESC LIMIT 5''' )
    query_results = c.fetchall()
    # Save (commit) the changes
    conn.commit()
    # We can also close the connection if we are done with it.
    # Be sure any changes have been committed or they will be lost.
    conn.close()

    # Convert tuples to lists
    results = {'names':[list(i) for i in query_results]}
    return jsonify( results )
```

12. Now that we are done writing the NameServer app, we want to use it to populate the database. We could setup a webpage with a form that POSTs to the app, one name at a time. Or we could write a Python program to POST to the app using the Requests module. In this exercise, we will use Python. Open the PostToNameServer.py file and run the program in a second Python console (not the console running the NameServer app). Make sure you understand what the program is doing.
13. There are a number of free programs for managing a SQLite database. Install one of the following and view the contents of the NameServer.sqlite file. Note that the NameServer is not capable of modifying or deleting any of the data. This must be done manually using a database manager.
    a. SQLite Manager Firefox Add-On (Windows/Mac, requires Firefox)
    b. SQLite Studio (Windows Executable, no install required)
    c. SQLite Browser (Windows/Mac)
14. On a final note, if you want to make the app reachable by other computers on your local network (e.g. your home WiFi or the AirBears2 network), change the host to 0.0.0.0 when starting the app. You will also need to know the IP address of your computer within the local network. This IP address can (almost always) be found using the socket module. Update the app and try sending names to it from a lab partner's laptop.

```
# Start the app
if __name__ == '__main__':
    import socket
    my_ip = str(socket.gethostbyname(socket.gethostname()))
    print
    print "* Running on the local network at http://"+my_ip+":5000/"
    app.run(host='0.0.0.0')
```