New York University
School of Continuing and Professional Studies
Management and Information Technology

Advanced Python

Homework, Session 2

Note: please consult the Code Quality document for organization and formatting hints.

Please choose between two sets of problems: **Option A** or **Option B**. Option B borrows from the Intro Python class and is intended for those of you new to some of these topics. Feel free to do either.

OPTION A are the 2 assignments (and 1 extra credit) that are listed below and were discussed briefly in class.

For this option you can refer to the topic summary document, new for this week.

2.1 Program argument validation and read from STDIN. The purpose of the assignment is to validate the arguments and process STDIN. As a bonus (not required), it could be used to send email. This program provides a simplified interface to the **sendmail** email progam (which is only available on Mac/Linux -- it could also be adapted to the **blat** email sender on Windows).

Argument validation: an email message requires two fields (**to_address** and **from_address**) and allows three optional fields (**cc, bcc** and **subject**). Write a script that allows the user to input these five fields, will exit with error if the required fields are missing, and will also exit with an error message and a usage: string if any of the specified fields are not valid (i.e., not among the five required and optional fields).

To read the arguments, take user input from the command line using **sys.argv** (please see the summary document for more information about **sys.argv**).

Read from STDIN: the message body should be read from a file. However, instead of using a filename to open the file for reading, the program should read from STDIN using **sys.stdin.** Thus the user will need to direct the text of the message body to the program either through file redirection or piping from another program.

The arguments can be entered in the below format, at the command line -- the following invocation shows the file redirection method of directing body text to STDIN:

```
$ ./email.py to=joe@wilson.com from=me@my.com cc=joe@joe.com bcc=mike@mike.com
subject='hey there!' < body_text.txt
```

Another method of directing text to STDIN is piping from another program (such as the

output of the **ls -l** file listing), which would look like this -- same arguments, but different STDIN input:

```
$ ls -l | ./email.py to=joe@wilson.com from=me@my.com cc=joe@joe.com
bcc=mike@mike.com subject='hey there!'
```

Another approach to piping could be to echo text and pipe its output to the program:

```
$ echo 'hey joe, what up?' | ./email.py to=joe@wilson.com from=me@my.com
cc=joe@joe.com bcc=mike@mike.com subject='hey there!'
```

Note on Windows commands: the dir listing replacement command for Unix **ls** in Windows is **dir**; the text output command for Unix **echo** in Windows is **type**.

Note: some servers are configured to block mail, however this is not a requirement of the assignment -- see note below.

a. Begin with the following stub:

```
import os
import sys

sendmail_prog = '/usr/sbin/sendmail'   # mac default.  use
                                       # 'which sendmail' to locate
                                       # sendmail on unix/linux

required_args = set(['to', 'from'])
valid_args = set(['to', 'from', 'cc', 'bcc', 'subject'])

args = sys.argv[1:]
```

b. Run the stub script with the arguments as shown in bold in the first example. After you run the program, note the value of **args**, which is a list of the arguments as collected from **sys.argv**.

c. Split out the argument keys and values and load the arguments as keys and values in a dictionary. Take these steps in as few lines as possible, and *without referring to the expected field names*. Use a **try**/**except** to detect errors in argument format as you convert to a dictionary (for example, what exception is raised if the user doesn't format the args as **this=that** pairs, and you attempt to split on **=** and load the results into a dict?). If an error is trapped by **except**, call **usage()** with a descriptive Usage: message. (See spec for **usage()** below.) **Please note that your except: must specify a specific exception.**

d. For extra credit (and a fair amount of satisfaction), can you use a list comprehension wrapped in a call to **dict()** to load the args into a dict?

e. Now use **set comparisons** to determine whether all specified fields are valid, and also to determine whether the two required fields are present. Keep in mind that *you will not have to loop through any of the containers to achieve this*, thanks to set comparisons.

f. Report errors: if any required arguments are not found in the argument keys, report them with a call to a custom function, **usage()**. If any argument keys are invalid

(i.e., not found in the **valid_args** list), report them with **usage()**.

g. Your **usage()** function should display a Usage: string (see examples in slides) that indicates how the program should be called.

h. Read the body of the message from **sys.stdin**. If no text is found there, issue a warning to **sys.stderr** and continue with an empty message body.

i. When the arguments have been validated, print out an email header string with the values inserted into a "header" template. This is the format needed for the **sendmail** program to send a mail message.

```
From: me@my.com
To: joe@joe.com
Subject: hey there!
Cc: mike@mike.com
```

Use a **triple-quoted string** to create the header template, and **str.format()** to insert the values. Print the header.

j. Optional: for fun and excitement (on **mac** and linux/unix where **sendmail** is available, windows users can attempt to use **blat**, although I have no information on how to use it) actually send your email message, using the following additional code:

```
msg = header + argdict['body']
sendmail = os.popen(sendmail_prog + " -t", "w")
sendmail.write(msg)
sendmail.close()
```

(As mentioned, if **sendmail** can't be found on your mac/linux/unix system, you can use the command **which sendmail** to see where it may be located.)

(Oops! **gmail** was too smart for my email send and placed my message in **Spam**! If you're not seeing an error but also not seeing your sent message, check your **Spam** folder.)

(Second oops! some home Internet Service Providers or networks may block direct email sending and may require that you use the ISP's **smtp** server, in which the above sendmail code would have to be modified to work properly. See your ISP's documentation or email me to discuss troubleshooting email sending.

2.2 Write a "file list" program that reads a directory (or optionally, a directory tree) and then presents the "top n" files sorted by one of three criteria: filename, file size or last modification time.

Please note that this assignment requires building a *dict of dicts* or *dict of lists* complex structure -- the solution should not sort using the OS's file list utility.

a. From the command line (using **argparse** or **sys.argv**), take user input for directory location, sort criteria ('size', 'mtime' or 'name'), # of results and sort direction ('ascending' or 'descending').

b. Make sure to test (or make sure **argparse** will test) to see that the sort criteria is one of the three options and that the sort direction is one of the two directions. Also make sure that all required options are specified. You can make some options optional with a default to be used if not specified.

c. Use **try/except** to ensure that the directory path is correct (i.e., capture the error if the directory can't be read).

d. Use **os.listdir()** to read a directory, or **os.walk()** to traverse a directory tree. For each file, find out its size and last modification time. Also, use **os.path.basename()** to grab the bare filename (without the path info), and store that within the dict.

e. Add each file into a dictionary of dictionaries (or you may use a dictionary of lists) so that each file in the tree has three values associated with it: bare filename, file size and last modification time.

f. Allow a 'help' message to be displayed indicating the command line options and how they work (the argparse module provides this automatically). Here is how the arguments should look if you use the **argparse** approach to argument handling:

```
./list_files.py --dir='testdir' --by=mtime
                --results=4 --direction='descending'


test2.txt:  37 bytes.  Last modified Tue Oct 13 11:56:22 2015
test4.doc:  44 bytes.  Last modified Tue Oct 13 11:56:03 2015
test3.py:  0 bytes.  Last modified Tue Oct 13 11:55:50 2015
test1.txt:  0 bytes.  Last modified Tue Oct 13 11:55:46 2015
```

2.3 (optional extra credit): **bitly** is a URL shortening and forwarding service. Anyone can submit a long URL to bitly and receive a shortened bit.ly URL, which they can then use as a website link. Visitors to the shortened bit.ly URL are duly forwarded to the submitter's long URL. Shortened links are useful in place where a long URL would be difficult to use, for example in Twitter posts.

For example, if you wanted to tweet a link, but the link was http://www.dothis.com/oh/my/gah/this/is/so/long.html?and=params&too=what&a=drag, you would register this long URL with bitly, bitly would link the URL with a new, shortened url (for example, http://bit.ly/QtQEt), and you could use this shortened link in its place. Users who click the bitly link would be redirected to the long URL through bitly's servers.

Since bitly forwards millions of links each day, it can aggregate user data to derive information about its users and their browsing habits -- i.e. where they are in the world, what browser they are using, what links they are clicking on, etc.

The dataset (located on the **source data** page linked on the course website home page) was collected on bitly links forwarded to **.gov** domains only for one day -- July 8 2011, the day of NASA's final shuttle launch. (This was all drawn from the usa.gov datasets on github)

The fields collected here are:

timestamp (the date and time; can be converted
using **import time; time.ctime(**timestamp**)**)
user_agent (info about the browser and platform accessing this link)
referring_url (the website and page the user was on before coming here)
short_url_cname (the bit.ly shortened URL)
long_url (the URL to which users are being forwarded)
geo_city_name
country_code
geo_region
accept_language (languages the browser is prepared to display)
timezone
lat (latitude)
long (longitude)

Additional data that can be derived from the above fields (using string manipulations or inspections like **str.split()** and **in**):

From the **long_url** field:
the **machine name** (e.g. **sanders.senate.gov**)
the **domain** (e.g. **nasa.gov**)

From the **user_agent** field:
the user's **platform** -- roughly speaking we can identify many user platforms by searching for the following terms within the **user_agent** field:

Android
BlackBerry
Windows NT
iPad
iPhone
iPad
iPod
Linux
Macintosh
PLAYSTATION 3

(Please keep in mind that platform analysis isn't meant to be at all exhaustive or comprehensive, and there are likely to be errors in the results when making these assumptions (e.g., that searching for "Windows NT" within the string will yield all Windows users); instead, this is part of an attempt to offer a real-world problem within the practicality of a reasonably simple solution. The complexity of the notations within the UserAgent string would make it too time consuming to make a comprehensive analysis.)

The assignment: please do at least one and as many of these as possible. Reading

the **bit.ly** tab-separated data (in the directory linked off of the **source data** link on the course home page), please write a program that can calculate one or more of the following based on a user-selected field in the table:

a. A set of unique cities represented in the data, sorted by name.

b. The "top ten" **country_code** values (use a dictionary sorted by value).

c. The "top ten" **machine_name** values (derived from the **long_url** field).

OPTION B are 3 assignments listed below. **Please note that if you choose this option, all 3 assignments are required.**

2.1 Create a sort function that will cause **sorted()** to sort the file lines in **revenue.txt** by the float value at the end of each line. **Do not build a new container of values -- sort the lines as they are!** Simply supply the **by_floatval** sort function to do this.

When you get the sort function to work, convert it to a lambda.

Starter Code (this assumes you are in **python_scripts** which is a folder next to **python_data**:

```
fh = open('../python_data/intro_data/revenue.txt')
lines = fh.readlines()
slines = sorted(lines, key=by_floatval)
```

2.2 Write a single list comprehension that returns the list of ids from student_db.txt. Print the IDs list.

Expected Output:

```
['jk43', 'axe99', 'jab44', 'ak9', 'ap172', 'jb23', 'jb29']
```

2.3 THIS ASSIGNMENT IS REQUIRED IF YOU ARE DOING OPTION "B" Reading through **stock_prices.csv**, build a dict of lists in which the key is the stock ticker and the value is a list of closing prices for that ticker. Sort the tickers by the difference between the highest and lowest close prices for each stock ticker for the year, and report as shown below. For extra credit, render the sort function as a lambda.

Brief discussion:

As you loop line by line, the operative code line is this:

```
ticker_prices[ticker].append(close)
```

where you are appending the closing price to a list of prices associated with a particular ticker (AAPL, GOOG, etc.)

However, as discussed, you will need to check the dict ahead of time to see if the ticker key is already there. If not, set the key and value in the dict for that line.

```
ticker_prices[ticker] = []
```

Please avoid looping through the file more than once. Please avoid using a separate list or set to hold any of the values. A dictionary of lists is all you need! This exercise helps you see how you can build and sort a multidimensional structure.

Once the structure is built, your sort function will sort each key in the dictionary by the difference between the highest and lowest close prices recorded. Inside the sort function, you can use the max() and min() functions to get highest and lowest; these functions take a list argument and return the greatest or least value of each.

Expected Output:

```
GOOG:  180.38 difference (672.93-492.55)
LNKD:  100.82 difference (270.76-169.94)
AAPL:  36.74 difference (133.0-96.26)
FB:  25.76 difference (98.39-72.63)
MSFT:  9.32 difference (49.61-40.29)
```

This assignment also features a discussion document.