INTRODUCTION TO PROGRAMMING
WEEK 8 - FILE MANIPULATION AND EMAILING

FINAL EXAM

Your final exam is coming up in 3 short weeks! The two part exam will take place on 3/23.

To pass this class, you will need to:

- Get a grade of B or better on final exam
- Complete ALL homework assignments before 12:00AM PST on 3/29

If you are behind on homework, I would strongly recommend catching up now instead of waiting until the last minute, since you will also need time to study for the exam.

GETTING THE DEMO FILES

This week's class will move very fast, so I have prepared demo files for you to follow along with.

Please run the following from your week8 directory:

```
cp -r /home/alberthwang/python/week8 demo_files
```

All the examples below reference something in the starter files.

FILES

In this section, we'll explore file objects - opening files up for reading, writing, and appending.

open()

The `open` function in python takes two parameters, the 1st being the name of the file being opened, and the 2nd being the mode that the file is opened in. **The function returns a `File` object that has multiple methods such as `read()`, `write()`, and `close()`.**

```
test_file = open('test.txt', 'r')
```

In this line, we are opening the file, `test.txt` (located in the same directory as our script) for reading, as specified by the second parameter, `'r'`.

There are 3 primary modes that a file can be opened up for:

```
test_file = open('test.txt', 'r')
test_file = open('test.txt', 'w')
test_file = open('test.txt', 'a')
```

As you can probably guess - `r` (reading), `w` (writing), and `a` (appending) are independent modes

of each other. The difference between 'a' and 'w' is that in 'w' mode, the contents of the original file are replaced with whatever is newly written to it, while the 'a' mode allows the script to append the file with whatever text the script specifies without altering the original contents.

read()

`read()` is a method of `File` objects that returns the contents of a file as a string. Create a `test.txt` file in your directory and put the following contents in it:

```
line 1...uno
line2...dos
line3...tres
```

Create a python file called `read_example.py` and add this content:

```
def main():
  test_file = open('test.txt', 'r')
  print test_file.read()

if __name__ == '__main__':
  main()
```

Running `read_example.py` should print the content of the file out onto the terminal.

write()

The `write()` method takes 1 parameter, the string that will be written to the file object. Create a file called `write_example.py` and add the following content:

```
def main():
  out_str = 'This is my favorite file!'
  test_file = open('test.txt', 'w')
  test_file.write(out_str)

if __name__ == '__main__':
  main()
```

Now open up `test.txt`, what is inside of it?

Try running it again, this time with `out_str` being:

```
out_str = 'This is my favorite file!\nI also like other files as well...'
```

In text files, '`\n`' delineates new lines, so you can use it to create newlines.

appending

Now, change the mode of `open('test.txt')` to `'a'` and change `out_str` to

```
out_str = 'add this on!'
open('test.txt', 'a')
```

Run `write_example.py` again and now check the contents of `test.txt`

CSV Files (Comma Separated Values)

The file format CSV, or comma-separated values is one of the most used file formats for data processing in business. Other data-intensive file formats such as .xls (excel spreadsheets) can be easily exported to csv format and worked with. For this section, we will be using a csv file, `data.csv`, which looks like this:

ldap	name	department
alberthwang	Albert Hwang	People Operations
smadaan	Saurabh Madaan	Sales

CSV Module - DictReader

Python has its own **csv** module that provides several handy classes for accessing and manipulating data in CSV files. You can find full documentation on the csv module here:

http://docs.python.org/library/csv.html

For class, we are going to explore the `DictReader` class which, in my opinion, is the most useful csv reading class in the module.

The `DictReader` class creates `DictReader` objects that hold each row of a csv file as a separate dictionary. In each dictionary, the keys are the csv file's headers (the first row).

```
import csv #1

def main():
  data_file = open('data.csv', 'rU') #2
  dictReader_data = csv.DictReader(data_file) #3

  for row in dictReader_data: #4
    print row #5
```

```
if __name__ == '__main__':
  main()
```

When we run the script above, we get the following output:

```
{'department': 'People Operations', 'name': 'Albert Hwang', 'ldap': 'alberthwang'} #6
{'department': 'Sales', 'name': 'Saurabh Madaan', 'ldap': 'smadaan'}
```

Let's look at this script part by part (follow the numbered comments above)

1. In order to use the csv module, we have to first import it.
2. The first thing we need to do to utilize the `DictReader` class is create a File object by opening the `data.csv` file in read mode.
3. Next, we need to create a `DictReader` object, and in order to do that, we pass the `data_file` object we created in step 1 into its constructor.
4. a `DictReader` object is an iterable (much like a list) so we can use a for loop on its contents.
5. Each row in the csv is read into the DictReader as a separate dictionary
6. We can see here that the keys of each row dictionary are the header values from the first row of the CSV file.

Mapping DictReader objects to a List of Dictionaries

While `DictReader` objects are pretty awesome, they aren't especially flexible. They aren't indexed by position, for instance, so it is not easy to find data on the 50th row in a csv using only the DictReader object. They also aren't associatively indexed (like dictionaries), so it's also difficult to find data on a row given a certain quality (e.g. where the ldap is 'alberthwang').

To add flexibility to our dealings with CSV files, we must map the `DictReader` object to a much more familiar data structure, our friend the `list`.

Let's take a look at this code:

```
def main():
  data_file = open('data.csv', 'rU')
  dictReader_data = csv.DictReader(data_file)
  list_data = [] #1

  for row_dict in dictReader_data: #2
    list_data.append(row_dict) #3

  print 'This is the 1st row of data'
  print list_data[0] #4
  print ''

  print 'This is the 2nd row of data'
  print list_data[1] #4
  print ''

  result_str = list_data[0]['name'] + '\'s department is '
  result_str += list_data[0]['department'] #5
```

```
  print result_str
```

When we change main to this and run the script again, we see the following print out:

```
This is the 1st row of data
{'department': 'People Operations', 'name': 'Albert
Hwang', 'ldap': 'alberthwang'}

This is the 2nd row of data
{'department': 'Sales', 'name': 'Saurabh Madaan', 'ldap': 'smadaan'}

Albert Hwang's department is People Operations
```

Comment Explanations:

1. First, we want to create a list that will hold the row dictionaries that `DictReader` objects generate.
2. Again, as before, we iterate through the `DictReader` object and extract its dictionaries
3. This time, we insert each dictionary into the list which will hold the data
4. Now, we can reference the row data by its row number
5. Since each item in `list_data` is a dictionary, its values can be extracted just like any old dictionary.

Mapping a DictReader Object to a Dictionary of Dictionaries

Oftentimes, each row of your data will be unique, representing an employee's information or a single transaction, etc. For instance, in our example, what if I wanted to find the department for ldap alberthwang? I could reference the row where his data is, I guess, but what if I don't know the row number? To do this, we need to map the `DictReader` to a dictionary of dictionaries.

Let's look at this code:

```
def main():
  data_file = open('data.csv', 'rU')
  dictReader_data = csv.DictReader(data_file)
  dict_data = {}

  for row_dict in dictReader_data:
    dict_data[row_dict['ldap']] = row_dict

  print 'Here are the LDAPs!'
  for ldap in dict_data:
    print ldap

  print ''

  print 'Here is the data on alberthwang'
  print dict_data['alberthwang']
  print ''
```

```
  print 'Here is alberthwang\'s department'
  print dict_data['alberthwang']['department']
```

When we change main to this and run the script again, we see the following print out:

```
Here are the LDAPs!
alberthwang
smadaan

Here is the data on alberthwang
{'department': 'People Operations', 'name': 'Albert
Hwang', 'ldap': 'alberthwang'}

Here is alberthwang's department
People Operations
```

Writing to CSV Files

To write to csv files, we will use the `DictReader` class' sister class - `DictWriter`.

The same steps apply - create a `DictWriter` object using a file object in its constructor and then write a dictionary into the csv.

```
def main():
  data_file = open('output.csv', 'w') #1
  field_names = ['ldap', 'name', 'department'] #2
  data_writer = csv.DictWriter(data_file, field_names) #3

  header_row = {'ldap': 'ldap', #4
                'name': 'name',
                'department': 'department'}

  data_writer.writerow(header_row) #5
  print 'header row written successfully.'

  row_data = {'ldap': 'satishm', #6
              'name': 'Satish Musurunu',
              'department': 'Engineering'}

  data_writer.writerow(row_data) #7
  print 'row written successfully.'
```

Comment Explanations:

1. As with `DictReader`, the first thing we need to do is create a `File` object with the CSV we want to manipulate. This time we will open it in `write` mode.
2. Next, we create a list of the fieldnames. These are the headers that will be used when we write to the CSV.
3. Creating the `DictWriter` object, we need both the `File` object that represents our

CSV and the field names.
4. First we need to create a dictionary with the headers (stupid, I know...versions 2.7 and later have a writeheader() method)).
5. Next, we have to write the headers to the file
6. The data that is written into the CSV must be in dictionary form like so.
7. To actually write the row into the CSV, just call the `DictWriter` object's `writerow` method and pass the row data.

Writing Multiple Rows to CSV

If you have multiple rows that you want to write into a CSV, you can use the `writerows` method of `DictWriter` objects and pass it a list of dictionaries.

```
row1 = {'ldap': 'satishm',
        'name': 'Satish Musurunu',
        'department': 'Engineering'}

row2 = {'ldap': 'alberthwang',
        'name': 'Albert Hwang',
        'department': 'People Operations'}

data = [header_row, row1, row2]
data_writer.writerows(data)
```

Of course, the dictionaries are usually created programmatically and not just hard-coded into the script.

Appending to CSV Files

What if we didn't want to over-write all the data in the CSV with our script? What if we just wanted to add another row underneath our current data? Then we would just need to open the file in `append` instead of `write` mode.

```
data_file = open('data.csv', 'a')
```

XLS Files (Excel)

Excel files (.xls) are often used in place of CSV in most business settings. To read/write excel files, we will need a separate python library that has been built to work with this somewhat odd data file type:

Reading XLS Files

To read XLS files, we'll need to use the xlrd library. Keep in mind that this library is 3rd party and does not come with most python installs. Luckily for you, xlrd is installed on gswitch already.

To start, let's create the XLS files with the same data in `data.csv`. Let's call this file `data.xls`. Here is some sample code to work with xlrd:

```
import xlrd #1

def main():
  xls_file = xlrd.open_workbook('data.xls') #2

  # Get the sheetnames as a list of strings
  print 'List of sheet names:'
  print xls_file.sheet_names() #3
  print ''

  # Get the specific sheet you want to work with
  sheet1 = xls_file.sheet_by_name('Sheet1') #4

  # Print the number of rows in test sheet1
  print 'Number of rows in Sheet1:'
  print sheet1.nrows #5
  print ''

  # Print out the data in each row
  print 'Each Row as a List:'
  for row_num in range(sheet1.nrows):
    print sheet1.row_values(row_num) #6

if __name__ == '__main__':
  main()
```

Running the script above will print:

```
List of sheet names:
[u'Sheet1', u'Sheet2', u'Sheet3']

Number of rows in Sheet1:
3

Each Row as a List:
[u'ldap', u'name', u'department']
[u'alberthwang', u'Albert Hwang', u'People Operations']
[u'smadaan', u'Saurabh Madaan', u'Sales']
```

Comment Explanations:

1. To use `xlrd`, you need to first import it.
2. Use the `open_workbook()` function to create a **workbook object**.
3. Once you have a **workbook object**, you can get a list of sheet names that belong to that workbook with its `sheet_names()` method.
4. To get at the data in a sheet, you need to first create a **sheet object** which you do by invoking a workbook object's `sheet_by_name()` method.
5. Once you have a **sheet object**, you can get the number of rows of data with the sheet object's `nrows` property.
6. To get a specific row of data from the sheet as a list, you can invoke the **sheet object**'s `row_values()` method.

*The ‘u’ in front of all the strings just make the strings unicode string that have a different encoding. For our purposes, though, we can treat these strings the same way we have been treating all other strings

Reading XLS files by column

Sometimes when working with data files, we aren’t especially interested in individual row data, but in aggregate data in a single column. To work with column data, we’ll want to use cell() method of sheet objects to obtain a specific cell value from the sheet. Usage is like this:

```
cell_value = <sheet_object>.cell(<row_index>, <col_index>).value
```

Let’s take at a script to get the data form the 3rd column of an excel sheet.

```
import xlrd

def main():
  xls_file = xlrd.open_workbook('data.xls')
  col3_data = [] #1

  # Get the specific sheet you want to work with
  sheet1 = xls_file.sheet_by_name('Sheet1')

  # Iterate through rows, getting 3rd column of data
  for row_num in range(sheet1.nrows):
    col3_data.append(sheet1.cell(row_num, 2).value) #2

  print col3_data

if __name__ == '__main__':
  main()
```

Comment Explanations:

1. In addition to creating a workbook and a sheet object, we also create an empty list object to hold the column 3 data.
2. For each row, we want to get the data in its 3rd column and append it to the col3_data list.

*Often, when working with numeric data, you’ll want to do some aggregate functions on the data such as sum, max, min, mean, etc. To do this, you can utilize some of Python’s built in aggregate functions for lists.

Writing to XLS Files

To write data to XLS files, we need to use a different library, the xlwt library.

The xlwt library is somewhat unusual. Essentially, each time you write to an XLS file, you need

to first create a workbook object and then save it to the XLS file. This basically only permits over-writes, but you can use a combination of xlrd and xlwt to read the old data out of a file, make some modifications, and write it back.

Let's take a look at a code sample.

```
import xlwt

def main():
  # Create a target XLS workbook object(WB) to write to
  target_wb = xlwt.Workbook() #1

  # Create a sheet in the WB to work with
  target_sheet = target_wb.add_sheet('Sheet 1') #2

  # Data to be written #3
  data = [[1, 5, 7, 8, 9],
          ['albert', 'sauwei', 'hwang'],
          [3.6, 7.7, 8.8, 2032.6]]

  # Write the data into the sheet
  for row_num, row_data in enumerate(data): #4
    for col_num, col_value in enumerate(row_data): #5
      target_sheet.write(row_num, col_num, col_value) #6

  # Save the WB object to the destination file
  target_wb.save('test.xls') #7

  print 'done.'

if __name__ == '__main__':
  main()
```

Comment Explanations

1. First, you need to create a Workbook object to work with
2. Create a Sheet object
3. Get the data to be written as a list of lists (each element of the outer list is a row, each element of a nested list is a column value in that specific row)
4. To write the data, first we need to iterate through the rows
5. Within each row (list) we iterate through the column values of that row
6. We use the Sheet object's write method, specifying a row number and column number along with the value to write
7. Finally, save the WB object to the destination file

SENDING EMAIL

To send email through a python script, you must first import the smtplib module which

provides several classes and methods for connecting to web servers to send mail. SMTP stands for Simple Mail Transfer Protocol which is an internet protocol for emailing (it is usually only used for outgoing mail transfer). You also need to import the `email` module which is used for creating email objects and help manage things like text type, attachments, headers, etc. Let's get right to it.

The following script will send a basic email from me to myself (**feel free to copy the SendEmail function from the notes for the homework and for your own purposes**):

```
import smtplib #1

from email.mime.text import MIMEText #2

def SendEmail(sender, recipients, subject, body):
  msg = MIMEText(body) #3, #5

  msg['Subject'] = subject #6
  msg['From'] = sender
  msg['To'] = recipients

  smtp_instance = smtplib.SMTP('localhost') #4
  smtp_instance.sendmail(sender, recipients.split(‘,’),
                         msg.as_string()) #7
  smtp_instance.quit()

  print 'sent.'

def main():
  SendEmail(sender='alberthwang@google.com',
            recipients='alberthwang@google.com',
            subject='I2P Test Email',
            body='Just testing things...')

if __name__ == '__main__':
  main()
```

A few things to notice here:

1. 2 crucial modules were imported - `smtp` and `email`
2. from the `email` module, we drilled into its `mime` directory and then its `text` directory to import the module `MIMEText`.
3. Inside the `MIMEText` module, there is a class called `MIMEText` and we created an object of that class with the line: `msg = MIMEText(body)`
4. From the `smtp` module, we created an SMTP object which is essentially a socket connection to a server which will send the email. In our case, we pass it the parameter,

`localhost` which tells the script to use the current server that the script is running off of (gswitch). It would be possible here to specify a different server that might have more capacity or specialize in smtp emails, but for our purposes, `localhost` will do.

5. `MIMEText` allows us to create an email object. MIME itself stands for Multi-purpose Internet Mail Extensions which is a protocol that allows emails to contain other kinds of content besides plain text.
6. Here we are altering the contents of an email's header.
7. To send an email, we need to call the SMTP object's `sendmail` method which takes as parameters - the sender, recipient, and the message object.

Sending multiple emails in one script

When sending multiple emails in one script, you only need 1 SMTP object. Just continually invoke its `sendmail()` method for each email sent. See a code sample here:

/home/alberthwang/python/week8/email/email_multiple.py

Sending to multiple recipients

To send to multiple recipients, you simply need to write a list of the recipients, separated by commas in this part of the code:

```
recipients = 'alberthwang@google.com, ryanvukelich@google.com'
```

Sending emails with HTML

Let's say you want to send an email with some nice formatting, like bolding, italics, etc. To do this, you will need to write the email body in HTML and then add an extra parameter to the `MIMEText` Object constructor that specifies the content type:

```
html_body = '<b>hello world!</b>' # bolds 'hello, world!'

msg = MIMEText(html_body, 'html')
```

There is much more to emailing with python than we cover here. If you're interested in learning how to add attachments to your emails, etc. I encourage you to do some Googling and investigate!