CSCI 195 – Intro to Programming w/Python
Completing the course outline creator program

Overview

In these exercises, we'll first create the program as described in the DyKnow slides, and then modify it so that we can specify the names of the columns that we want to include as command line arguments to the program.

Instructions:

☐1. Open Chrome and go to http://cloud.sagemath.com

☐2. Click the link to open the course project, and then on the link to the folder named **CSCI195-assignments**.

☐3. Click the link to open the folder **course-outline-creator**.

☐4. Open the file **createCourseOutline-final.py.** This contains the code we have gone over in class, except that it doesn't include the code that prints out the tabs (January, February, March, etc.) or the code that prints out the individual month tables.

☐5. Line 68 gives a review of what the tabs should look like, while lines 81 through 99 have comments describing an individual line of code. You should write the code described by each of the comments on lines 81, 85, 88, 91, 96 and 99.

You can try to write the code on your own; or use the DyKnow slides as a reference.

You can test your code by executing:
`python createCourseOutline-final.py > outline.html`
from the terminal session associated with the project, and then examining the contents of the file outline.html

☐6. Lines 111-126 describe what needs to happen to output the actual tables containing the information for each month. In this case, there's only an outline of what needs to happen, rather than a step by step series of comments for each line.

Try to implement the code as described; again, use the DyKnow slides as a reference if you need to.

Now we'll add the ability to specify which columns should be included in the HTML output. We'll do this by allowing one of two options to be specified on the command line:

- If the `--include` argument is given, it will contain a comma separated list of column names to **include**, and those will be the only columns included in the output. For example, we might execute
  `python createCourseOutline-final.py --include "Day, Date, Topic"`
  to make it so that only the 3 specified columns are included in the HTML version.
- If the `--exclude` argument is given, it will contain a comma separated list of column names to **exclude,** and all columns **except** for those columns will be included in the output. For example,
  `python createCourseOutline-final.py --exclude "Class Notes, Due Date"`
  would make the HTML contain the columns *Day*, *Date*, *Topic*, *Student Preparation*, *Assign* and *Due Date*, but not the *Class Notes* and *Due Date* columns.

☐7. Add code at the top of the program that makes the `argparse` module available, to help with parsing command line arguments.

☐8. After the declaration of the `postscript` variable, add this line to create a parser object (don't include any line breaks anywhere)

```
parser = argparse.ArgumentParser(
  description='Create a web page version of an Excel-based course outline',
  epilog='Exactly one of the --include and --exclude options may be specified'
)
```

☐9. Add the following lines after the one you just added, to configure the parser object to look for `--include` and `--exclude` command line arguments:

```
parser.add_argument('--include',
  help='A comma separated list of column names to be included in the output'
)
parser.add_argument('--exclude',
  help='A comma separated list of column names to be excluded from the output'
)
```

☐10. Add text to be printed as part of the help message, after printing out information about the possible options:

☐11. Ask the parser to parse the command line arguments, storing the results in a variable named `options`

```
options = parser.parse_args ()
```

The options variable will have properties for each of the arguments that were passed on the command line. For example, if the user invoked the program as:

```
python createCourseOutline-phase5.py --include "Day,Date,Topic"
```

then `options.include` will be defined, but `options.exclude` will not.

☐12. Write an `if` statement that checks to ensure that at least one of the options `--include` or `--exclude` were given. If not, then call `parser.print_help()` and then `call sys.exit()`.

```
if not options.include and not options.exclude:
    parser.print_help()
    sys.exit()
```

☐13. Write a similar if statement that checks whether both include and exclude arguments were given. If so, print an error message and call `sys.exit()`.

☐14. Find the lines of code that read in the first line of the input file and create the dictionary `columnMap` from it. Right after that loop, write an if/else statement that does the following:

- if the `include` option was specified, create an **empty** list named `columnsToInclude`
- otherwise create a list named `columnsToInclude` with the values 0, 1, ..., up to the number of entries in `columnMap - 1`

☐15. Modify the `if` portion so that it does the following steps

- Breaks the value of `options.include` into a list of strings named `columns`, using a comma as a separator
- Loops over each column name `columnName` in the list `columns`, and
  - Calls the `strip` function on the variable `columnName`, storing the result back into `columnName`
  - Uses the `append` method to add the column number associated with the column name to the list `columnsToInclude`, by looking up the column name in the `columnMap` dictionary

☐16. Further modify the `if` portion of the code so that if a column name is encountered that **doesn't exist** in the `columnMap` dictionary, an error message is printed (which includes the name of the offending column name) and the program is terminated by calling `sys.exit()`.

You can test this aspect of your program by running:

```
python createCourseOutline-phase5.py --include "Day,Invalid,Topic"
```

☐17. Modify the else portion of the code you've been working on so that it

- Breaks the value of `options.exclude` into a list of strings named `columns`, using a comma as a separator
- Loops over each column name `columnName` in the list `columns`, and
  - Calls the `strip` function on the variable `columnName`, storing the result back into `columnName`
  - Uses the `remove` method to remove the column number associated with the column name to the list `columnsToInclude`, by looking up the column name in the `columnMap` dictionary

☐18. After the if/else, add the following to help you debug:

```
print "columnsToInclude is", columnsToInclude
sys.exit()
```

Test your program by invoking it with various options for `--include` and `--exclude` and verifying that the appropriate column numbers exist in the list `columnsToInclude`. For example

```
python createCourseOutline-phase5.py --include "Day,Date,Topic"
```

should output the list `[0, 1, 2]`.

☐19. Remove the code that you added above, since it's only for debugging purposes.

☐20. Find the code that prints out the table headers; it looks like this:

```
print "<tr>"
for header in columnHeaders:
    print "<th>{0}</th>".format(header)
print "</tr>"
```

☐21. Modify the above code so that it looks like this

```
print "<tr>"
columnNumber = 0
for header in columnHeaders:
    if columnNumber in columnsToInclude:
        print "<th>{0}</th>".format(header)
    columnNumber = columnNumber + 1
print "</tr>"
```

This code should only print those columns that have been requested. To verify, run

```
python createCourseOutline-phase5.py --include "Day,Date,Topic" > outline.html
```

and examine the contents of outline.html. You should see only the 3 requested headers in the table; the data will still be wrong at this point.

☐22. Find the code that creates the variable `tableRow`, just above the declaration of the variable `monthNames`.

☐23. Add code similar to what was done for the headers so that only the appropriate columns are added to the variable `tableRow`.

☐24. Rerun the program, and verify that the output is as expected.

Accessing information from Excel directly

Now we have the ability to take a tab separated file which was saved from an Excel workbook, and generate tab-based web page from you, with each tab representing a month of class. Pretty awesome! But what if we wanted to skip the intermediate step of saving the workbook into a tab-separated file? It seems like this would be a hard thing to do – but luckily a **module** already exists to make our lives easier.

Let's get to work modifying our program!

☐25. Identify the portions of the code that perform the following tasks, and write the line numbers down:

a. Open up the tab separated file
   Line(s): _____

b. Read the first line of the tab separated file
   Line(s): _____

c. Create the values in the dictionary `columnMap`
   Line(s): _____

d. Processes each row of the tab separated file, putting the entries into the appropriate list stored in the dictionary `months`
   Line(s): _____

Now we will install a Python module that will allow us to perform the same operations on the Excel file directly.

☐26. Open the terminal window inside the **course-outline-creator** folder.

☐27. Enter the command below into the terminal window:

```
pip install xlrd
```

`xlrd` is a module that allows us to read information from an Excel workbook; there is a corresponding `xlwt` module to write (create and edit) workbooks, but since we don't need that functionality, we'll just install the `xlrd` module.

☐28. Add an appropriate statement to the beginning of the program to give the program access to the functionality in the `xlrd` module.

☐29. The `xlrd.open_workbook` function works just like the `open` function does, except that it is able to open and parse Excel workbook files. It also works correctly with Python's `with` construct, which means we can drop it into our program without much effort.

Modify the line of the program that opened the tab separated text file using `open` with a call to `xlrd.open_workbook`, opening the file **outline.xlsx** instead of **outine.txt**.

☐30. Add the following line of code as the first line of the `with` block:

```
sheet = outline.sheet_by_name("Lecture")
```

This gives us a variable to refer to the worksheet in the workbook named *Lecture*.

Next we need to modify the process by which the column headers are stored into the dictionary named `columnMap`.

☐31. Replace the `while` loop with a `for` loop that iterates over the result of calling `sheet.row(0)`, using the variable `cell` as the loop variable.

The `row` method will give us a list of `cell` objects in the specified row.

☐32. Change the line that sets up the value in the `columnMap` dictionary from

```
columnMap[columnHeaders[column]] = column
```

to

```
columnMap[cell.value] = column
```

The `value` property gives us the actual contents of the cell.[1]

Now we need to change the portion of the code that iterates over the rows in the outline. We'll use the `nrows` property of the sheet object to help us iterate over the rows, and the `row` method of the sheet to access the information stored in each row.

☐33. Find the loop that iterates over the lines in the file (approximately line 70).

☐34. Replace that loop with

```
for rowNumber in range(1,sheet.nrows):
  columns = sheet.row(rowNumber)
```

Make sure you understand what is happening here before moving on.

---

[1] As far as I can see, there aren't any other properties available on a `cell`; I suspect that the `value` property exists to allow for future expansion in case other properties become necessary in the future.

☐35. Modify the code that creates the variable named `tableRow` to get the `value` property of the column, by changing

```
tableRow = tableRow + "<td>{0}</td>".format(column)
```

to

```
tableRow = tableRow + "<td>{0}</td>".format(column.value)
```

☐36. Modify the code that assigns a value to the variable `dateColumnValue` by adding `.value` at the end:

```
dateColumnValue = columns[columnMap['Date']].value
```

This change is needed to accommodate the `xlrd` module's way of storing the cell values.

☐37. Unfortunately, Excel stores dates as numeric values, rather than strings, so we'll need to do some work to get the month part of the date[2]. Replace the code

```
dateParts = dateColumnValue.split("/")
month = int(dateParts[0])
```

with

```
year, month, day, hour, minute, second =
xlrd.xldate_as_tuple(dateColumnValue, outline.datemode)
```

Note the line break above comes from Word's rendering of this code; you can't enter it into the document.

The function `xldate_as_tuple` allows us to convert between the numeric value and the individual components. You may remember that a **tuple** is similar to a list, except that its contents can't be modified.

[2] Specifically, it stores the date as a floating point number representing the number of days since January 0, 1900. Thus, a numeric value of 1 corresponds to January 1, 1900, 2 refers to January 2, 1900, and 1.5 would be noon on January 1, 1900. A value between 0 and 1 represents just a time, without a date component. Many computer systems store dates in this fashion to facilitate easy comparisons and computations involving dates.

☐38. Our final modification comes when printing out the table headers around line 126. Change the loop so that it iterates over `sheet.row(0)` instead of the list `columnHeaders`; you'll also need to add `.value` in an appropriate place when printing out the `<th>` element.

If you run the program now, storing the output in **outline.html** as we have been doing previously, it will be *almost* right. Printing the day and date values won't really work correctly; the days will be floating point values like 1.0 instead of just 1, and the dates will be printed as their numeric values.

I'll leave it as a challenge to you to handle this problem if you like.

I solved it by using the `ctype` property of a cell to determine if the cell value being output is a date. The `ctype` property will have the value `xlrd.XL_CELL_DATE` when the cell contains a date value, and it will have the value `xlrd.XL_CELL_NUMBER` when the value is a number.

When the value is a date, I used the `xldate_as_tuple` function to get the parts of the date, and then combined them together using format to get a string value that looks like `1/11/2016`.

CSCI 195 – Intro to Programming w/Python
Performing sentiment analysis on the Bible

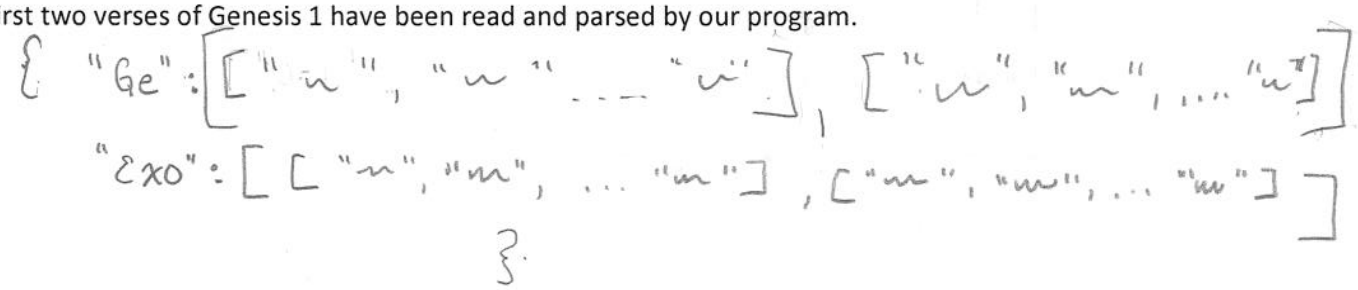
We have been given a version of the King James Bible whose contents have been encoded using a simple text format. The following is a sample of the file's contents.

Ge@1:1@In the beginning God created the heaven and the earth.
Ge@1:2@And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.
...
Ge@1:31@And God saw every thing that he had made, and, behold, it was very good. And the evening and the morning were the sixth day.
Ge@2:1@Thus the heavens and the earth were finished, and all the host of them.
...
Ge@50:26@So Joseph died, being an hundred and ten years old: and they embalmed him, and he was put in a coffin in Egypt.
Exo@1:1@Now these are the names of the children of Israel, which came into Egypt; every man and his household came with Jacob.
...

As you can see, each line of the file contains a single verse; at the beginning of the line is an abbreviation of the book, chapter and verse, encoded in the format **Book@Chapter:Verse Number@Verse Text**.

Our goal will be to create a **dictionary**, with the keys being strings representing the book. Each entry in the dictionary will be a list, with entry `c-1` in the list representing chapter `c` (e.g., entry 0 represents chapter 1, entry 1 represents chapter 2, and so on). The **contents** of each chapter will be a list of strings, containing the text of each verse. As with the chapters, entry `v-1` contains the text of verse `v`.

1. Using { } for dictionary, and [ ] for list, show what the contents of our dictionary will be after the first two verses of Genesis 1 have been read and parsed by our program.



2. Write a statement that declares an empty dictionary named `bible`.

```
bible = dict()
```

3. Write a statement that opens the file named **kjv.atv** using a `with` statement, and then iterates over each line in that file with a `for` loop. Use the variable `line` as the loop variable.

```
with open( kjv.atv) as file :
    for line en file :
```

4. Assume that `line` stores a single line from the file. Write a series of statements that (1) strips off any trailing whitespace from `line`, storing the stripped text back in `line`; (2) splits the variable `line` on the @ sign, storing the result into a variable named `parts`; and (3), stores the components of `parts` into variables named `book`, `reference` and `verse_text`, in that order.

```
line = line.rstrip()
parts = line.split("@")
book = parts[0]
reference = parts[1]
verse_text = parts[2]
```

5. Assuming that `reference` stores a chapter and verse combination (such as 1:1 or 2:50), write a series of statements that (1) splits `reference` based on the `:` character, storing the result into a variable named `parts`; (2) uses the `int` function to convert the *first* entry in `parts` into an integer, storing the result into a variable named `chapter`; and (3) uses the `int` function to convert the *second* entry in `parts` into an integer, storing the result into a variable named `verse`.

```
parts = reference.split(":")
chapter = int(parts[0])
verse = int(parts[1])
```

6. Write an `if/else` statement that checks to see if there is an entry in the dictionary `bible` for the string variable `book`.
   - If `book` **does exist** as a key in the `bible` dictionary, set the variable named `book_chapters` to be the value in `bible` associated with `book`.
   - If `book` **does not exist** as a key in `bible`, (1) set a variable `book_chapters` to be an **empty list**, and then store `book_chapters` as the value for the key `book` in the `bible` dictionary.

```
if book in bible:
    book_chapters = bible[book]
else:
    book_chapters = []
    bible[book] = book_chapters
```

7. Write an `if/else` statement that checks whether the number of entries in the list named `book_chapters` is **at least as big** as the variable `chapter`, which we assigned a value to in step 5. Fill in the body of the `if/else` according to the following logic:
   - If the test is **true**, we know that there is already an entry in `book_chapters` corresponding to the chapter represented by `chapter`, and so we can append the variable `verse_text` to the list associated with `chapter` (remember chapter numbers are 1 based, but list indexes are 0 based)
   - If the test is **false**, this is the first verse we've seen from this chapter. So we need to add a new list containing only `verse_text` to the end of the `book_chapters` list.

```
if len(book_chapters) >= chapter - 1:
    bookchapters[chapter - 1] append(verse_text)

else:
    bookchapters . append(verse_text)
```

8. Take the code from the previous steps and enter it into a new file named **parse_bible.py** in the assignment named **sentiment_analysis** (Start a new file by clicking the New button from the Files tab, enter the name **parse_bible.py** and then press Enter).

   Add a print statement at the end as follows:

   ```
   print bible["Ge"][0][0]
   ```

9. Execute the program, checking for any syntax errors:

   ```
   python parse_bible.py
   ```

   If all goes well, you should see the contents of Genesis 1:1. If you don't, check the logic of each of the statements that you've entered. Remember that you can use

   ```
   python -m pdb parse_bible.py
   ```

   to start the program in a debugger. You can then use `n` followed by `Enter` to execute the program one line at a time, and use statements like `print book` to see the values of variables.

10. Add a second print statement to print out the contents of the last verse:
    `print bible["Rev"][21][20]`
    and ensure the output is as expected

11. Now that we know our data structure is complete, we'll add some interactivity. First, write a statement that asks the user to enter a book in the bible and stores the user's response as a variable named `book`, and then sets the variable `book_chapters` to be the list of chapters in the dictionary `bible` associated with the book entered by the user.

12. Assume the variable `book_chapters` has been set using the code you wrote above. Determine what the following series of statements does:

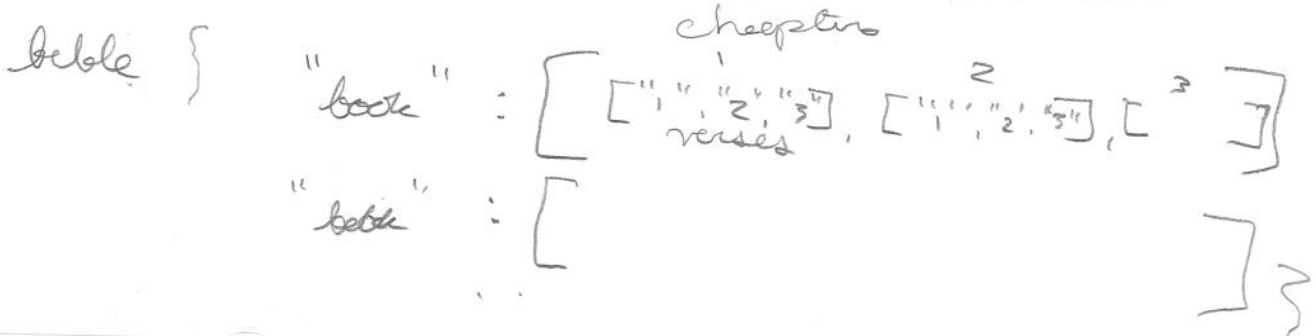
```
list = []
for c in book_chapters:
  list.append(len(c))

print "{0}: {1}".format(book, max(list))
```

13. Consider the following "transcript" of an interactive session, with user input shown in **bold**:

```
python parse_bible.py
Enter the desired book: John
Enter the desired chapter [1 - 21]: 3
Enter the desired verse [1 - 36]: 16
John 3:16 For God so loved the world, that he gave his only begotten
Son, that whosoever believeth in him should not perish, but have
everlasting life.
```

Add code at the end of **parse_bible.py** to implement the functionality shown above. Make note of how the program prompts the user with the appropriate limits for chapter number and verse number.



14. Add code to the end of your program that prints out the name of the book containing the **fewest** chapters. Here's a process you can use to figure this out:

    Create a variable named `fewest_chapters` and set its value to the **length** of the list associated with the string key `Ge` (it actually doesn't really matter which book you pick here)

    Create a second variable named `book_with_fewest_chapters`, and set it to the string value `Ge` (must be the same as what you picked above).

    Loop through the keys in the `bible` dictionary; each time through the loop:
    Compare the length of the list associated with the current key to `fewest_chapters`
    If that length is less than the value of `fewest_chapters`,
    update the values of both `fewest_chapters` and `book_with_fewest_chapters`

    After the loop is done executing, print out the values of `book_with_fewest_chapters` and `fewest_chapters`.

    The correct answer is Obad (Obadiah) with a single chapter.

According to the web site bibleresources.org/how-to-study-bible, the "principle of first mention" indicates that:

> "It is important to look for the place in the Bible that a subject, attitude or principle is mentioned for the first time, and see what it meant there."

We'll want to modify our program so that it asks the user to enter a word, and the reference and first where that word first occurs is printed. For example:

```
Enter a word you wish to find the first occurrence of: grace
First mention of grace is in Ge 6:8
```

15. In order to do this, we're going to have to keep track of the ordering of the books. Rather than entering this information ourselves, we can keep track of it while we parse the bible. To do this:
    - Initialize an empty list named `books` before opening the file *kjv.atv*
    - When you come across a book that is not yet in the dictionary named `bible` , append it to `books`
16. Add the appropriate `raw_input` statement to the end of your program to ask the user for the word to search for.

17. Add code to find the first occurrence of the given word. I did this by writing 3 nested loops:
    Loop over each book in the books list
      Loop over the chapters in the current book
        Loop over the verses in the current chapter
          Test to see if the desired word is contained within the current verse, using the `in` operator

    I used variables **`found`** (Boolean, initially False and set to True when the desired word has been found), `chapter_num` and `verse_num` to help me in my implementation

```
1   from random import random
2
3   sentiment = dict()
4
5   scores = {
6     "positive": 1,
7     "neutral": 0,
8     "negative": -1,
9     "weakneg": -1
10  }
11
12  multipliers = {
13    "weaksubj": 1,
14    "strongsubj": 2
15  }
16
17  with open("sentiment-dictionary.txt") as sentimentFile:
18      for line in sentimentFile:
19          line = line.rstrip()
20          parts = line.split(" ")
21          entry = dict()
22
23          for part in parts:
24              key = part.split("=")[0]
25              value = part.split("=")[1]
26              entry[key] = value
27
28
29          pos = entry["pos1"]
30          word = entry["word1"]
31
32          polarity = entry["priorpolarity"]
33          if polarity == "both":
34                polarity = "positive" if random() < 0.5 else "negative"
35
36          type = entry["type"]
37
38          score = scores[polarity]
39          multiplier = multipliers[type]
40
41          if pos not in sentiment:
42            pos_dict = {}
43            sentiment[pos] = pos_dict
44          else:
45            pos_dict = sentiment[pos]
46
47          pos_dict[word] = score * multiplier
48
```