Lecture 9: Files and List Comprehensions

Check-in and Reminders

- Pick up graded **Homework 2** from the box up front
- Reminder: Homework 3 is due in class on Monday
- Mon/Tues's lab is going to be a partnered lab
    - If you plan to start early, make sure to find a partner in the same lab section as you
    - In a team, one person's repository will be chosen an they will invite the other as a collaborator
    - Instructions are given in the Lab assignment
- Topics to review for lab: File reading and writing (today's lecture), Nested lists, looping over lists and maintaining state variables, etc.

Do You Have Any Questions?

Today’s Highlight

List comprehensions!

- Remember the very common operation you had to do over and over for this week’s lab?
  - Iterating over a list and accumulating some strings from it such as:
    - Return all words that have length n
    - Return all words that start with a vowel
    - Return canon of all the words
  - **What steps do we have to take for these operations?**
  - Today we will learn how to do them in one line!
  - Plus other things: reading CSV files and writing to files

Reading CSV Files

- **Analyzing CSV data**
  - A CSV (Comma Separated Values) file is a type of plain text file that stores **"tabular"** data.
  - Each row of a table is a line in the text file, with each column on the row separated by commas.
  - This format is the most common import and export format for spreadsheets and databases.

Name	Age
Harry	14
Hermoine	14
Dumbledor	60

CSV form:

```
Name,Age
Harry,14
Hermoine,14
Dumbledor,60
```

CSV Module

- Python's `'csv'` module provides an easy way to read and iterate over a CSV file.

Path to file on computer as a string

```
In [1]: import csv # the module must be explicitly imported
```

```
In [2]: with open('roster.csv') as myFile:
            csvf = csv.reader(myFile)
            print(csvf)
        # implicitly closes file
        # csvf is a  file object that can be iterated over
```

Variable name for your file object

```
<_csv.reader object at 0x10c1556d0>
```

- `csvf` is a csv file object that is **iterable**, that is, can we iterated over in a loop (similar to sequences)

Iterating over a CSV object

- When we iterate over a CSV object, the loop variable is a **list** and its items take the role of the contents of each row (in order).

```
In [3]: with open('roster.csv') as myFile:
            csvf = csv.reader(myFile)
            for row in csvf:
                print(row)
```

```
['Ahmad,Omar', '23AAA', '02 (Shikha)']
['Bennett,Zoe', '23AAA', '02 (Shikha)']
['Le,Long N.', '21AAA', '02 (Shikha)']
['Bal,Gabriella H.', '20AAA', '02 (Shikha)']
['Chen,Kary', '23AAA', '02 (Shikha)']
['Dean,Sarah R.', '23AAA', '02 (Shikha)']
['Eckerle,Jacob M.', '23AAA', '02 (Shikha)']
['Kilinc,Onder', '23AAA', '02 (Shikha)']
['Litton,William', '23AAA', '02 (Shikha)']
['Lynch,Lauren E.', '23AAA', '02 (Shikha)']
['McCarey,Lauren R.', '23AAA', '02 (Shikha)']
['Mohan,Avery E.', '23AAA', '02 (Shikha)']
['Mojarradi,Mohammad Mehdi', '23AAA', '02 (Shikha)']
['Peters,Maximilian E.', '23AAA', '02 (Shikha)']
['Robayo,Salvador', '23AAA', '02 (Shikha)']
['Ruschil,Evan U.', '23AAA', '02 (Shikha)']
```

Accumulating CSV rows: List of Lists

- We can iterate over a CSV file and accumulate all rows (each of which is a list) into a mega list, that is, a list of lists.

In [4]:
```
rosterList = []
with open('roster.csv') as myFile:
    csvf = csv.reader(myFile)
    for row in csvf:
        rosterList.append(row)
```

In [5]:
```
rosterList # lets see what is in the rosterList
```

Out[6]:
```
[['Ahmad,Omar', '23AAA', '02 (Shikha)'],
 ['Bennett,Zo   '23AAA', '02 (Shikha)'],
 ['Le,Long N.', '21AAA',
 ['Bal,Gabriella H.', '20
 ['Chen,Kary', '23AAA', '02 (Shikha)'],
 ['Dean,Sarah R.', '23AAA', '02 (Shikha)'],
```

Nested List!

Indexing Nested Lists

- We have 32 students in class, so 32 lists in our `rosterList`
- Lets see how we can index this list to access information of random students
- Python's random module helps generate random numbers
- To access the name field we can write `rosterList[index][0]`

```
In [13]: import random # import module to help generate random numbers

In [16]: randomIndex = random.randint(0, 31)
         # generates a random integer between 0 and 31

In [17]: rosterList[randomIndex]

Out[17]: ['Wolf,Samuel T.', '21AAA', '02 (Shikha)']
```

Reorganizing CSV data

- Let us write some helper functions that take as input a list (which is a row of the CSV file) and output a cleaned row as a tuple.
- The returned tuple must have three items: first name (string), last name (string), graduation year (as a two digit int)

In [31]:
```
def reorgData(rowList):
    """Takes a row of a CSV (as a list) and returns
    a tuple of student information"""
    # tuple assignment, splitting last name
    # and first(with middle) name
    lName, fmName = rowList[0].split(',')
    fName = fmName.split()[0]
    year = rowList[1]  # takes the form '23AAA'
    yy = int(year[:2])
    return fName, lName, yy
```

Beware of Empty Cells in CSV

- We used a relatively clean and complete CSV file in our example
- In general CSVs, some information might be missing and thus some cells might be empty
- To make sure your code is trying to index a string which could be missing from some data that you include a check for it
- Easy way to check a string or list is not empty?

```
if len(seq):
     # word is not empty
     # statements using word
```

Accumulation in Lists

- Let's get to know our class better! We will write a function `yearList` which takes in two arguments `rosterList` (list of lists) and `year (int)` and returns the list of students in the class with that graduating year

In [29]:
```
def yearList(classList, year):
    result = []
    for sList in rosterList:
        # tuple assignment:
        fName, lName, yy = cleanData(sList)
        if yy == year:
            result.append(fName + ' ' +lName)
    return result
```

Analyze File with Sequence Tools

- Who has the most number of vowels in their name?
- First write a `getName` helper function that takes studentInfo as a tuple and returns a string which is firstName (space) lastName

```
In [30]: def mostVowelName(classList):
             currentMax = 0 # initialize max value
             persons = []  # initialize list for names
             for sInfo in classList:
                 name = getName(sInfo)
                 numVowels = countAllVowels(name)
                 if numVowels > currentMax:
                     # found someone whose name as more vowels
                     # than current max update person, currentMax
                     currentMax = numVowels
                     persons = [name] # reupdate
                 elif numVowels == currentMax:
                     # is someone's name as long as currentMax?
                     persons.append(name)
             return persons, currentMax
```

Writing to Files

- We can write all the results that we are computing into a file
- The following code will create a new file named `studentFacts.txt` and write the formatted strings to it
- **Note.** Writing to a file in write mode deletes all previous contents of the file. Mode only for creating and writing to new files.

```
In [35]: with open('studentFacts.txt', 'w') as sFile:
             sFile.write('Fun facts about CS134 students.\n')# need newlines
             sFile.write('No. of first years in CS134: {}\n'.format(len(yearList(rosterList, 23))))
             sFile.write('No. of sophmores in CS134: {}\n'.format(len(yearList(rosterList, 22))))
             sFile.write('No. of juniors in CS134: {}\n'.format(len(yearList(rosterList, 21))))
             sFile.write('No. of seniors in CS134: {}\n'.format(len(yearList(rosterList, 20))))
```

- Open your current directory to see the new file and its contents!

Appending to Files

- To append to an existing file we open it in the append mode `'a'`
- Notice the `format` function is helpful in writing to files as well, not just printing

In [52]:
```
with open('studentFacts.txt', 'a') as sFile:
    sFile.write('Name with most vowels: {}\n'.format(mostVowelName(rosterList)))
    sFile.write('Name with least vowels: {}\n'.format(leastVowelName(rosterList)))
```

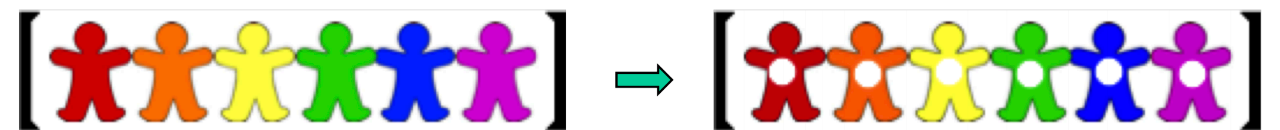
```
Fun facts about CS134 students.
No. of first years in CS134: 18
No. of sophmores in CS134: 8
No. of juniors in CS134: 3
No. of seniors in CS134: 3
Name with most vowels: ('Marika Massey-Bierman', 8)
Name with least vowels: ('Long Le', 2)
```

List Patterns: Map & Filter

```
people = ['Hermione Granger', 'Harry Potter', 'Ron Weasley', 'Luna Lovegood']
```

- **Mapping.** return a new list that results from performing an operation on each element of a given list. E.g. Return a list of the first names in `people`

  `['Hermione', 'Harry', 'Ron', 'Luna']`



- **Filtering.** return a new list that results from keeping those elements of a given list that satisfy some condition E.g. Return a list of names with last names ending in 'er' in `people`

  `['Granger', 'Potter']`

Image Source: (http://cs111.wellesley.edu/spring19)

List Comprehensions

List Comprehension for mapping

```
newSequence = [expression for item in sequence]
```

List Comprehension for filtering

```
newSequence = \
[item for item in sequence if conditional]
```

To notice:

- List comprehension starts with an expression (note that a variable like item is an expression), for example, **`x*2`** or **`n`**.
- Never use **append** in this position. We are using list comprehension to avoid creating a list with **append**.

Image Source: (http://cs111.wellesley.edu/spring19)

List Comprehensions: Mapping & Filtering

```
newSequence = \
[expression for item in sequence if conditional]
```

The example below shows a list comprehension that extracts the even numbers from a range object and creates a list of their squares. The code to the right is analogous and shows the same process with iteration.

```
result = []
for n in range(10):
    if n%2 == 0:
        result.append(n**2)
result
```

List Comprehension
for filtering and mapping

```
result = [n**2 for n in range(10) if n%2 == 0]
```

Image Source: (http://cs111.wellesley.edu/spring19)

Assert

- Python's `assert` statement is a debugging aid that tests a condition.
- If the condition is true, it does nothing and your program just continues to execute.
- But if the `assert` condition evaluates to false, it raises an `AssertionError` exception with an optional error message
- Assertions are internal self-checks for your program

```
assertStatement = "assert" exp1 [","  exp2]
```

- `exp1` is the condition we test, and the optional `exp2` is an error message that's displayed if the assertion fails.

Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/