Python Refresher and the Jupyter Notebook

Due 11:59pm Thursday, September 5

Objective

A pre-requisite for this course is prior experience with Python at the level of 15-112 or 95-888. This assignment provides an opportunity to brush up on your Python knowledge.

- Start the notebook with `% jupyter notebook` (where `%` is the prompt)
- Start a new notebook and call it `a1-python-review`. The suffix of `ipynb` will be automatically added
- Recall that an *expression* evaluates to a value and a *statement* does something. E.g., `3+4` evaluates to `7` whereas `print(8)` prints `8` on the screen. When you have a sequence of expressions in a cell and evaluate them you only see the value of the last expression. For example, if
  ```
  4+8
  2+5
  ```
  were in a cell and it was evaluated we would only see 7. If you want to see both you need to print the values:
  ```
  print(4+8)
  print(2+5)
  ```
- Use list comprehensions where possible

Write the solutions to the following Python exercises in the Python notebook conforming to the following instructions:

1. Write your full name and andrewID in Markdown cell at the top of your notebook.
2. Precede each code cell with a Markdown cell which briefly describes the problem you are solving in the cell below. Just 2-3 lines will suffice
3. For ease of grading, use the specified function names (e.g., `zeller` etc.)

This is an individual assignment. You are allowed to discuss high level ideas, but the final work needs to be your own. Software tools will be used to check submissions for authenticity. You are welcome (and expected) to use online resources (stackoverflow etc.) to seek answers to specific Python related questions.

Be sure that your submission conforms to the expectations in our Python style guide. Non-conformance will result in points off.

Exercises

1. [parse_hms, hms_to_seconds, seconds_to_hms, sum_hms] For this problem you will write 4 functions that deal with manipulating time.

i. **`parse_hms`** takes a string representation of *a time duration* and returns a triple of hours, minutes, and seconds.

```
parse_hms('2:34:16')-> (2, 34 16)
parse_hms('14:21:56') -> (14, 21, 56)
parse_hms('45:00') -> (0, 45, 0)
parse_hms('23') -> (0, 0, 23)
```

While you don't have to do any error checking (e.g., are the minutes < 60 etc.), your code needs to handle flexible input --- not all three components (hours, minutes, seconds) need to be present in the string representation: `'45'` means 45 seconds; `'2:34'` means 2 minutes and 34 seconds; finally if all 3 components are present `'3:23:15'` means 3 hours, 25 minutes, and 15 seconds. For appropriate credit, do not use a FOR loop. Rather use a list comprehension.

ii. **`hms_to_seconds`** takes three arguments, hours, minutes, and seconds and returns the equivalent total number of seconds. This is a very easy function—a 1 liner.

```
hms_to_seconds(1,23,24) -> 5004
hms_to_seconds(0,33,37) -> 2017
```

iii. **`seconds_to_hms`** is the converse of `hms_to_seconds`. It takes a number of seconds returns the equivalent in hours, minutes, and seconds.

```
seconds_to_hms(2017) -> (0, 33, 37)
seconds_to_hms(0) -> (0,0,0)
hms_to_seconds(*seconds_to_hms(1234)) -> 1234
```

Note the use of the destructuring operator '' to convert the tuple into 3 distinct arguments.*

```
h, m, s = seconds_to_hms(54321)
hms_to_seconds(h,m,s)-> 54321
```

iv. **`sum_hms`** takes an array of strings representing time durations and adds them. You will use the previously defined three functions to define `sum_hms`:

```
sum_hms(['0:1:24',  '2:34:45']) -> (2, 36, 9)
sum_hms(['3:00:00', '23:00', '45']) -> (3, 23, 45)
sum_hms([]) -> (0,0,0)
```

For credit, you should NOT have any explicit loops in `sum_hms`. All iterations should be expressed with list comprehensions. In the most common implementation you will have two list comprehensions. Recall that the python `sum` function can add the elements of an array `sum([2,5,3]) -> 10`

2. [Zeller's congruence] This question requires you to do some background reading. *Zeller's congruence* is an intriguing formula for calculating the day of week given the day month and year. Read more about it at the Wikipedia page https://en.wikipedia.org/wiki/Zeller%27s_congruence . Write the Zeller function which takes a year, month, and day and returns a number from 0 to 6 representing the day of the week. The detailed formula is given on the Wikipedia page (we currently use the Gregorian calendar). Few things to note:

i. The formula does an adjustment when the month is either Jan or Feb. You will have code along the structure of

```
if month <= 2:
   # do something
else:
   # do something else
```

- You will have to use integer division (//) and the modulo operator (%). Python is one of the few languages that does the mathematical modulo i.e., `-2 % 7 = 5`, which is what the Zeller congruence needs.
- When given a formula like this, it is a good idea to use the same variable names in the formula in your code

Test your code on some known dates e.g., today; your birthday; new year of 2019 falls on a Tuesday and `zeller(2019,1,1)` should return 3.

3. An early probability question. Some consider the following to be one of the early questions that led to the creation of the field of probability. Which of the following two options (events) has a higher probability of occurrence:

i. roll a six in 4 throws of a dice
ii. roll two sixes in 24 throws of a pair of dice

Write the following functions:

a. `dice()` which randomly returns a number from [1, 6]
b. `six_in_4()` / simulates the successive rolling of a single die 4 times (which is equivalent to rolling 4 dice simultaneously once) and checks for the occurrence of a 6, returns `True` if a 6 occurs in the simulation and `False` otherwise
c. `two_six_in_24()` / simulates the successive rolling of two dice 24 times and checks for the occurrence of two sixes. Returns `True` if a pair of 6s occur in the simulation and `False` if not.

Both `six_in_4` and `two_six_in_24` do a single simulation of that event. Finally, write the function `prob_question(n=1000)` which will perform both of these simulations n times and returns the probability of occurrence of both events as a pair of numbers in a tuple in the order of (i, ii) where i is the probability of event i (see

above). Note that `prob_question` uses a keyword argument. Just `prob_question()` will perform the simulation 1000 times.

4. File I/O and dictionaries. Download the movies lens 100k data set from https://grouplens.org/datasets/movielens/100k/ . A description of the various files is given in http://files.grouplens.org/datasets/movielens/ml-100k-README.txt . You will only need the files `u.data` and `u.item`. Copy these into the folder where your ipynb file is. The format of these files is as follows:

```
u.data:  user id   item id  rating  timestamp
u.item: movie id | movie title | …
```

The goal of this assignment is to merge information from these two files.

Item id and movie id refer to the same movie. Note that the field separator for `u.data` is white space and that for `u.item` is the vertical bar |. The Python string method `split` divides a string returns an array of constituent pieces. E.g., `"a|b|c|d".split('|') => ['a', 'b', 'c', 'd']`. Write the following functions.

i. `read_u_data(data_file='u.data', encoding='utf-8')` returns a dictionary where the key is the `itemid` (same as `movieid`) and the value is a *list of all ratings* for that `itemid`.

ii. `calculate_avg_u_data_rating(d)` takes the output of `read_u_data` and returns a dictionary with the *average of the list of ratings*: `{ movieid : average_rating, ... }`. The returned dictionary will be of the form `{23 : 4.35, 12 : 3.12, …}`

iii. `read_u_item(data_file='u.item', encoding='latin-1')` reads the contents of the file `u.item` and returns a dictionary with movie_id as the key and movie_title as the value.

iv. `write_ratings(u_data_avg, u_item, fname='movie-ratings.txt')` takes the dictionaries created in (ii) and (iii), performs the equivalent of an SQL JOIN and writes output to fname in the following format:

```
movieid   avg-rating  movie-title
```

The output produced by write_ratings should be sorted in descending order based on rating. Movies that have the same rating should be sorted in alphabetical order. When printed, ratings should have only 2 decimal digits e.g., 3.26 or 1.73. `write_ratings` also returns the sorted list of movie ratings. (the same list which is printed to the file).

When the equivalent of the following is executed

```
d = calculate_avg_rating()
write_ratings(d)[:10]          # return the first 10 items of the list
```

You should see as output in your notebook:

```
[('1536', 5.0, 'Aiqing wansui (1994)'),
 ('1653', 5.0, 'Entertaining Angels: The Dorothy Day Story (1996)'),
 ('814', 5.0, 'Great Day in Harlem, A (1994)'),
 ('1201', 5.0, 'Marlene Dietrich: Shadow and Light (1996) '),
 ('1189', 5.0, 'Prefontaine (1997)'),
 ('1467', 5.0, 'Saint of Fort Washington, The (1993)'),
 ('1500', 5.0, 'Santa with Muscles (1996)'),
 ('1599', 5.0, "Someone Else's America (1995)"),
 ('1293', 5.0, 'Star Kid (1997)'),
 ('1122', 5.0, 'They Made Me a Criminal (1939)')]
```

And the file `movie-ratings.txt` created in your directory with the full correct output. [How many lines should there be in the file?]

5. Plot Stock Prices.

For this part you do not need to write a function. Rather just write code in a cell, which when evaluated will produce a line plot.

Historical stock price information is available at Yahoo finance. For example, information about Apple is available at http://finance.yahoo.com/quote/AAPL/history?ltr=1 .

i. Setup matplotlib with:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

ii. Pick a company of your choice and download stock information for the whole calendar year 2018 (Jan 1 2018 to Dec 31 2018).
iii. Save the stock info in a csv file with the company's symbol e.g., `aapl.csv` or `goog.csv`. The structure of the csv file is given in the header: `Date,Open,High,Low,Close,Volume,Adj Close`
iv. We are interested only in the last column 'Adj Close'. Read each line of the file (you will hard code the name of your csv file) and store the 'Adj Close' values in a list called 'y'. Search the web for "python destructuring assignment" or "python unpacking" for an elegant way of reading a line of text and extracting names components.
v. Ensure that the values you are going to process are in chronologically increasing order. Some stocks may have had a split; don't worry about that. Just plot the values you get from yahoo.com
vi. We determine the x-axis values as `x = range(1, len(y)+1)`
vii. Plot the Adj close values (y) against the day of the year (x) with `plt.plot(x,y)`

Look up the documentation for matplotlib and determine how to set the xlabel, ylabel, and title of the plot. Your values should be:
xlabel: Days
ylabel: Adj. Close Price
title: "Stock price of AAPL in 2018" [naturally, use the symbol of the stock you picked.]

6. One hot encoding. A recommended way of encoding nominal values (e.g, colors, models of cars etc) as numbers is termed one hot encoding. The idea is simple: if you have 5 nominal values, then each value is encoded by 5 bits, where only 1 bit is set for each unique value. Following is the expected behavior of the function:

```
>>> one_hot_encoding(['alpha','beta', 'gamma'])

[['alpha', [1, 0, 0]], ['beta', [0, 1, 0]], ['gamma', [0, 0, 1]]]

>>> one_hot_encoding(['ford','honda', 'toyota', 'mazda', 'subaru'])

[['ford', [1, 0, 0, 0, 0]],
 ['honda', [0, 1, 0, 0, 0]],
 ['toyota', [0, 0, 1, 0, 0]],
 ['mazda', [0, 0, 0, 1, 0]],
 ['subaru', [0, 0, 0, 0, 1]]]
```

Note: Python provides a convenient way of creating copies of the elements of a list. `[0]*5 -> [0,0,0,0,0]`

7. Dense to Sparse matrix conversion. Many big data applications work with large sparse matrices. Large in that they will have several thousand (sometimes even a million) rows and columns. Sparse in that most of the entries will be 0 (typically only 10% of all entries will be non-zero). Storing such a large sparse matrix in a traditional format would be a huge waste of space as we would be storing a whole bunch of zeros. There are several alternative representations for such sparse matrices which are space efficient. One representation is to explicitly list the row index, column index, and value as a list of triples. For example, conside the following dense (traditional) representation of a matrix:

```
   [[1, 0, 2]
    [0, 0, 3]]
```

In this sparse representation we only store the non zero values as a list of triples (i, j, value). Indexing of the rows and columns starts with 0.

```
[(0,0,1), (0,2,2), (1,2,3)]
```

Write a function that will take a dense matrix and return the equivalent sparse representation.

```
dense2sparse(
   [ [1, 0, 2],
     [0, 0, 3]
   ])
```

Returns `[(0,0,1), (0,2,2), (1,2,3)]`. Note that an interesting property of this representation for sparse matrices is that the order of the triples doesn't matter.

The main learning objective of this question is (i) practice in list comprehensions and (ii) exposure to the python function `enumerate`. Hence your solution needs to be of the form:

```
def dense2spares(m):
    ans = < a nested list comprehension using enumerate twice >
    return ans
```

Of course, you will specify a doc string, as always. Read online about the `enumerate` function.

What to submit:

A zip bundle, `a1.zip`, with (1) your jupyter notebook `a1-python-review.ipynb` and (2) you csv file e.g., `aapl.csv` etc.