Quiz 7

To get credit for this quiz, use the Quiz tool at eee.uci.edu to enter your answers, within the Sunday-to-Tuesday quiz period.

Problem 1 (10 points) **Topic: List processing (with application to files)**

Suppose we wish to process text files that contain some "commented out" lines. That is, the file has a line that starts `"BBEGIN COMMENT"` and a line that starts `"END COMMENT"`; we want to keep all the lines *except* the ones between those two lines. Let's say that we have read the file into a list of strings and that we are guaranteed that the `"BEGIN COMMENT"` and `"END COMMENT"` lines will occur in the file.

Complete the definition of `def remove_commented_out_lines` below, consistent with its header, doc-string, and assertions. [Note that no actual file-handling commands are required for this solution.]

```
def remove_commented_out_lines(linelist: 'list of str') -> 'list of str':
    ''' Return input list, excluding lines between "BEGIN COMMENT" and
        "END COMMENT".  See examples below.
    '''
  result = []
  in_comment = False
  for line in linelist:
    if line.startswith("BEGIN COMMENT"):
      in_comment = True
      continue
    if line.startswith("END COMMENT"):
      in_comment = False
      continue
    if in_comment:
      continue
    result += [line]   # or result.append(line)
  return result

    # Other solutions are possible
```

```
test_list = ["Include this line",
             "and this one, too",
             "BEGIN COMMENT: Exclude this",
             "Also exclude this",
             "END COMMENT",
             "But include this last one."]

assert remove_commented_out_lines(test_list) == ['Include this line',
                            'and this one, too', 'But include this last one.']

assert remove_commented_out_lines(test_list[2:]) == ["But include this last one."]

assert remove_commented_out_lines(test_list[1:-1]) == ["and this one, too"]

assert remove_commented_out_lines(test_list[2:-1]) == []
```

Problem 2 (16 points) **Topic: Computation with if/elif/else, list processing, file processing**

Suppose we have these definitions from a previous quiz:

```
from collections import namedtuple
Date = namedtuple('Date', 'year month day')
```

where all three fields are numbers, so that November 12, 2015 would be represented as `Date(2015, 11, 12)`.

```
DrivingRecord = namedtuple('DrivingRecord', 'name license age tickets')
```

where `name` is a string representing a driver's name, `license` is a string representing his or her driver's license number, `age` is the driver's age, and `tickets` is a (possibly empty) list of Date objects containing the dates on which the driver has received a traffic ticket (i.e., was cited by a police officer for violating a driving law).

(a) (7 points) Complete the definition of `Date_is_earlier` below, consistent with its header, docstring, and assertions:

```
def Date_is_earlier(date1: Date, date2: Date) -> bool:
    ''' Return True if date1 is earlier than date2 (and False otherwise---
        for a boolean function, this goes without saying; it has to return
        either True or False)
    '''
```

```
if date1.year < date2.year:
    return True
elif date1.year > date2.year:
    return False
elif date1.month < date2.month:
    return True
elif date1.month > date2.month:
    return False
elif date1.day < date2.day:
    return True
else:
    return False
```

One way to approach this problem is to look at the assertions. There's no guarantee that every set of assertions on an exam will be comprehensive, but this one is: It lays out the various cases to check:
Are the years different (d1 < d2, or d1 > d2 actually)?
If the years are equal, are the months different (d1 < d2 or d1 > d2)?
If not, are the days different (d1 < d2)?

```
assert(Date_is_earlier(Date(2012, 1, 1),  Date(2013, 1, 1)))
assert(Date_is_earlier(Date(2012, 1, 1),  Date(2012, 2, 1)))
assert(Date_is_earlier(Date(2012, 1, 1),  Date(2012, 1, 2)))
assert(Date_is_earlier(Date(2013, 1, 14),  Date(2013, 2, 1)))
assert(not Date_is_earlier(Date(2013, 1, 14),  Date(2013, 1, 1)))
assert(not Date_is_earlier(Date(2013, 5, 1),  Date(2013, 1, 1)))
assert(not Date_is_earlier(Date(2013, 5, 14),  Date(2013, 1, 1)))
assert(not Date_is_earlier(Date(2013, 5, 1),  Date(2013, 1, 14)))
assert(not Date_is_earlier(Date(2012, 1, 1),  Date(2012, 1, 1)))
assert(not Date_is_earlier(Date(2013, 1, 2),  Date(2013, 1, 1)))
assert(not Date_is_earlier(Date(2012, 2, 1),  Date(2012, 1, 1)))
assert(not Date_is_earlier(Date(2013, 1, 1),  Date(2012, 1, 1)))
assert(not Date_is_earlier(Date(2013, 1, 1),  Date(2012, 2, 1)))
```

(b) (9 points) For this problem, assume that you have the following function already defined; you do not have to define it:

```
def Date_is_weekend(d: Date) -> bool:
    ''' Return True if d is a Saturday or Sunday '''
```

Complete the definitions below of the functions `total_tickets`, `total_weekend_tickets`, and `weekend_ticket_percentage`.

```
def total_tickets(DRL: 'list of DrivingRecord') -> int:
    ''' Return the total number of tickets issued to all drivers in DRL
    '''
  total = 0
  for dr in DRL:
    total += len(dr.tickets)
  return total
```

Of course you need to call the previously-defined functions where appropriate; it's never full credit to duplicate code.

```
def total_weekend_tickets(DRL: 'list of DrivingRecord') -> int:
    ''' Return the total number of tickets issued on Saturday or Sunday.
        You may write a second function (a "helper function") to break this task down.
    '''
  # Alternative (without a helper function, using a nested loop):
  #             total = 0
  #             for dr in DRL:
  #               for d in dr.tickets:
  #                 if Date_is_weekend(d):
  #                   total += 1
  #             return total
  total = 0
  for dr in DRL:
    total += total_weekend_tickets_on_ticketlist(dr.tickets)
  return total

def total_weekend_tickets_on_ticketlist(ticketlist: 'list of Date') -> int:
  ''' Take a list of Dates (when tickets were issued) and count the
    number of tickets that were issued on Saturday or Sunday '''
  total = 0
  for d in ticketlist:
    if Date_is_weekend(d):
      total += 1
  return total
def weekend_ticket_percentage(DRL: 'list of DrivingRecord') -> float:
    ''' Return the percentage of all tickets issued that were issued
        on a Saturday or Sunday (value from 0 to 100)
    '''
  return total_weekend_tickets(DRL) / total_tickets(DRL) * 100
  # It would be better coding practice to check that total_tickets returns at least 1 since you can't divide by zero.
  # Also, note that to get a percentage you need to multiply by 100.
```

One aspect of this question is being able to USE a function whose header and docstring are supplied, even if you don't have the body of the function. This is a common programmer's experience---any time you use help() or look up the functions and methods in a library, you do this.

Problem 3 (9 points) **Topic: Reading from and writing to files**

(a) (5 points) Suppose we have a file of driving records as shown below:

```
John Jones    111222333     24         12/24/11,1/31/12,6/30/12
Jane Johnson  222333444     25
Joe Jenkins   333444555     18         4/5/12
Jill Jefferies    444555666           55    2/24/01,10/18/05
```

The four fields are separated by tabs; the list of ticket dates is separated by commas; each date is in mm/dd/yy format. The following code creates a list of DrivingRecords from a file like the one above.

```
def mmddyy_to_Date(mmddyy: str) -> Date:
    ''' Return Date from mm/dd/yy '''
    parts = mmddyy.split('/')
    return Date(2000+int(parts[2]), int(parts[0]), int(parts[1]))

## ALTERNATIVE 1
infile = open('records.txt', 'r')
inputlist = infile._________________()
DRL = [ ]
for dr in inputlist:
    fields = dr.split('\t')
    if len(fields) == 3:
        ticketlist = [ ]
    else:
        ticketlist = fields[3].strip().split(',')
    Datelist = []
    for d in ticketlist:
        Datelist.append(mmddyy_to_Date(d))
    record = DrivingRecord(fields[0], fields[1], int(fields[2]), Datelist)
    DRL.append(record)
print(DRL)
infile.close()

## ALTERNATIVE 2
infile = open('records.txt', 'r')
DRL = [ ]
for _________________ in _________________:
    fields = line.split('\t')
    if len(fields) == 3:
        ticketlist = [ ]
    else:
        ticketlist = fields[3].strip().split(',')
    Datelist = []
    for d in ticketlist:
        Datelist.append(mmddyy_to_Date(d))
    record = DrivingRecord(fields[0], fields[1], int(fields[2]), Datelist)
    DRL.append(record)
print(DRL)
infile.close()

## ALTERNATIVE 3
infile = open('records.txt', 'r')
inputstring = infile._________________
inputlist = inputstring.split('\n')
DRL = [ ]
for dr in inputlist:
    fields = dr.split('\t')
    if len(fields) == 3:
        ticketlist = [ ]
    else:
        ticketlist = fields[3].strip().split(',')
    Datelist = []
    for d in ticketlist:
        Datelist.append(mmddyy_to_Date(d))
    record = DrivingRecord(fields[0], fields[1], int(fields[2]), Datelist)
    DRL.append(record)
print(DRL)
infile.close()
```

There are five different ways to read text files in Python:

1. `read()`
2. `read(n)`
3. `readline()`
4. `readlines()`
5. `for line in file`

Match each of the ways listed above with one of the following statements:

A. Could be used in Alternative 1.

B. Could be used in Alternative 2.

C. Could be used in Alternative 3.

D. Could be used with a while loop to check whether the last line we (tried to) read is empty.

E. Would be more effective if the input file were organized into fixed-width columns.

read(): C. Could be used in Alternative 3.
read(n): E. Would be more effective if the input file were organized into fixed-width columns.
readline(): D. Could be used with a while loop to check whether the last line we (tried to) read is empty.
readlines(): A. Could be used in Alternative 1.
for line in file: B. Could be used in Alternative 2.

(b) (4 points) Below is come code to write a list of DrivingRecords to a file in the format described above in part (a). Fill in each blank with one Python identifier, operator, or constant:

```
def Date_to_mmddyy(D: Date) -> str:
    ''' Return a string in the form mm/dd/yy from the Date
    '''
    return str(D.month) + "/" + str(D.day) + "/" + str(D.year)[2:4]

def DRList_to_file(DRL: 'list of DrivingRecord', file_name: str) -> None:
    ''' Write a list of DrivingRecords to the named file (tab-delimited, with dates
        in the form mm/dd/yy
    '''
    outfile = open(_________________, 'w')    file_name
    for dr in DRL:
        output_line = dr.name + "\t" + str(dr.license) + "\t" + str(dr.age) + "\t"
        for d in dr._________________:  tickets
            output_line += _________________(d) + ","    Date_to_mmddyy
        output_line = output_line[:-1] + "\n"   # Remove trailing comma, add \n
        _________________.write(output_line)    outfile
    outfile.close()
```