Ministry of Education, Singapore
Computing Teachers' Content Upgrading Course 2019

Practical Assessment 2

22 May 2019

Time allowed: 3 hours

Instructions to candidates:

1. This is an **open-book** exam.
2. Answer all **three** questions.
3. You may complete your solutions in any IDE first before copy them into this Jupyter Notebook for submission.
4. Input validation is not required
5. Submit this Jupyter Notebook online before test ends. You may submit multiple times, but only last submission before test end time will be accepted. https://driveuploader.com/upload/jX3ZAxA8cl/ (https://driveuploader.com/upload/jX3ZAxA8cl/)
6. Please note that the sample test program may not be enough to test your program. Your programs will be tested with other inputs and they should exhibit the required behaviours to get full credit.

Question 1

In a maze exploring game, you move step by step in one of the 4 directions, `east`, `west`, `south` and `north` to navigate around the maze. You use a `TimeMachine` to record direction of each step. This `TimeMachine` instructs you the direction to move if you need to backtrack your steps.

For example, the following sample code shows how the Time Machine works.

```
tm = TimeMachine()
tm.record('north')
tm.record('east')
tm.record('east')
tm.record('south')
tm.recall()       # return 'north'
tm.recall()       # return 'west'
tm.recall()       # return 'west'
tm.recall()       # return 'south'
```

A) Implement the `TimeMachine` class as following:

- Create an empty list `_moves` in initializer to record movements.
- Write a method `record()` which records direction of a step.
- Write a method `recall()` which backtracks previous step. It returns the direction to move, e.g. return `west` if previous step is `east`.
  - **Hint:** Declare a class variable `DIRECTIONS` dictionary to keep the 4 opposite directions pairs, i.e. `east` vs `west`, `south` vs `north`.

In [1]:

```
class TimeMachine:
    DIRECTIONS = {'east':'west', 'west':'east', 'south':'north', 'north':'south'}

    def __init__(self):
        self._moves = []

    def record(self, move):
        self._moves.append(move)

    def recall(self):
        move = self._moves.pop() if self._moves else None
        if move:
            return type(self).DIRECTIONS[move]
        else:
            return None
```

Test Case 1

In [2]:

```
tm = TimeMachine()
tm.record('north')
tm.record('east')
tm.record('south')
tm.record('west')
print(tm.recall())       # print out 'east'
print(tm.recall())       # print out 'north'
print(tm.recall())       # print out 'west'
print(tm.recall())       # print out 'south'
print(tm.recall())       # print out None
```

```
east
north
west
south
None
```

Expected output:

```
east
north
west
south
None
```

Test Case 2

In [3]:

```
tm = TimeMachine()
print(tm.recall())       # print out None
print(tm.recall())       # print out None
print(tm.recall())       # print out None
```

```
None
None
None
```

Expected output:

```
None
None
None
```

Test Case 3

In [4]:

```
tm = TimeMachine()
tm.record('south')
tm.record('south')
tm.record('south')
tm.record('east')
print(tm.recall())       # print out 'west'
print(tm.recall())       # print out 'north'
tm.record('west')
print(tm.recall())       # print out 'east'
```

```
west
north
east
```

Expected output:

```
west
north
east
```

B) Implement a class `TimeMachine2` which inherits from `TimeMachine` class and adds following functions.

- Write a method `peek()` which returns next direction to move **if** user would like to backtrack previous step.
  - **Note:** It does not perform actual backtrack.
- Write a method `size()` which returns current number of recorded steps in the list.
- Write a method `is_empty()` which returns `True` if there is no more recorded steps in the list, otherwise return `False` .

In [5]:

```
class TimeMachine2(TimeMachine):

    def peek(self):
        move = self._moves[-1] if self._moves else None
        if move:
            return type(self).DIRECTIONS[move]
        else:
            return None

    def size(self):
        return len(self._moves)

    def is_empty(self):
        return len(self._moves) == 0
```

Test Case 1

In [6]:

```
tm = TimeMachine2()
tm.record('north')
print(tm.size())         # print out 1
print(tm.peek())         # print out 'south'
tm.recall()
print(tm.size())         # print out 0
print(tm.peek())         # print out None
```

```
1
south
0
None
```

Expected output:

```
1
south
0
None
```

Test Case 2

In [7]:

```
tm = TimeMachine2()
print(tm.is_empty())     # print out True
```

```
True
```

Expected output:

```
True
```

Test Case 3

In [8]:

```
tm = TimeMachine2()
tm.record('north')
tm.record('south')
print(tm.size())          # print out 2
print(tm.peek())          # print out 'north'
tm.record('east')
print(tm.size())          # print out 3
print(tm.peek())          # print out 'west'
tm.recall()
tm.recall()
print(tm.size())          # print out 1
print(tm.peek())          # print out 'south'
print(tm.is_empty())      # print out False
tm.recall()
print(tm.size())          # print out 0
print(tm.peek())          # print out None
print(tm.is_empty())      # print out True
```

```
2
north
3
west
1
south
False
0
None
True
```

Expected output:

```
2
north
3
west
1
south
False
0
None
True
```

Question 2

The file `"percent-bachelors-degrees-women-usa-pivot.csv"` gives percentage of bachelor degree holders who are women in USA from year 1970 to year 2011.

A) Write a function `read_csv()` to read the file and returned records in a list.

- Each record (data point) in the list is a tuple of values, e.g. `('1970', 'Agriculture', '4.22979798')`.
- Do NOT include header row in the returned list.

In [9]:

```
import csv

def read_csv(file):
    result = []
    with open(file) as f:
        reader = csv.reader(f)
        header = next(reader)
        for r in reader:
            result.append(r)
    return result
```

Test Case 1

In [10]:

```
file_name = 'percent-bachelors-degrees-women-usa-pivot.csv'
table = read_csv(file_name)
print("Record count: ", len(table))
print(table[:4])
```

```
Record count:  714
[['1970', 'Agriculture', '4.22979798'], ['1970', 'Architecture', '11.9210053
9'], ['1970', 'Art and Performance', '159.7'], ['1970', 'Biology', '29.08836
297']]
```

Expected output:

```
Record count:  714
[('1970', 'Agriculture', '4.22979798'), ('1970', 'Architecture', '11.92100539'),
 ('1970', 'Art and Performance', '159.7'), ('1970', 'Biology', '29.08836297')]
```

Test Case 2

In [11]:

```
file_name = 'percent-bachelors-degrees-women-usa-pivot.csv'
table = read_csv(file_name)
print(table[1])
```

```
['1970', 'Architecture', '11.92100539']
```

Expected output:

```
('1970', 'Architecture', '11.92100539')
```

B) Write a function `list_majors()` to return the names of all bachelor degrees in the file.

- It returns a distinct list of bachelar degree names sorted in ascending order.

In [12]:

```
def list_majors(table):
    return sorted(list({r[1] for r in table}))
```

Test Case

In [13]:

```
file_name = 'percent-bachelors-degrees-women-usa-pivot.csv'
table = read_csv(file_name)
majors = list_majors(table)
print(majors)
```

```
['Agriculture', 'Architecture', 'Art and Performance', 'Biology', 'Busines
s', 'Communications and Journalism', 'Computer Science', 'Education', 'Engin
eering', 'English', 'Foreign Languages', 'Health Professions', 'Math and Sta
tistics', 'Physical Sciences', 'Psychology', 'Public Administration', 'Socia
l Sciences and History']
```

Expected output:

```
['Agriculture', 'Architecture', 'Art and Performance', 'Biology', 'Business', 'Com
munications and Journalism', 'Computer Science', 'Education', 'Engineering', 'Engl
ish', 'Foreign Languages', 'Health Professions', 'Math and Statistics', 'Physical
 Sciences', 'Psychology', 'Public Administration', 'Social Sciences and History']
```

C) The percentage value (column 3) in all records must be between 0 and 100. Write a function `find_invalid_records()` to print out all invalid data points.

- Note: NO need to remove invalid data points from table.

In [14]:

```
def find_invalid_records(table):
    for r in table:
        if float(r[2]) < 0 or float(r[2]) > 100:
            print(r)
```

Test Case

In [15]:

```
file_name = 'percent-bachelors-degrees-women-usa-pivot.csv'
table = read_csv(file_name)
find_invalid_records(table)
```

```
['1970', 'Art and Performance', '159.7']
['1970', 'Communications and Journalism', '-35.3']
['1971', 'Social Sciences and History', '-36.2']
['1973', 'Architecture', '114.7916134']
['2010', 'Physical Sciences', '140.2']
```

Expected output:

```
('1970', 'Art and Performance', '159.7')
('1970', 'Communications and Journalism', '-35.3')
('1971', 'Social Sciences and History', '-36.2')
('1973', 'Architecture', '114.7916134')
('2010', 'Physical Sciences', '140.2')
```

D) Write a function `calc_avg()` to find the average percentage of women who obtained a particular bachelor degree.

- Remember to remove invalid data points before calculation.

In [16]:

```
def calc_avg(table, major):
    perc= [float(r[2]) for r in table if r[1]==major and float(r[2])>=0 and float(r[2])
    return sum(perc) / len(perc)
```

Test Case

In [17]:

```
file_name = 'percent-bachelors-degrees-women-usa-pivot.csv'
table = read_csv(file_name)
major = 'Agriculture'
result = calc_avg(table, major)
print(major, result)
```

```
Agriculture 33.848165147547626
```

Expected output:

```
Architecture 34.146367113902436
```

Question 3

ALWAYS2 is a online store which sells all items at a fixed price of $2. User can add multiple items to his shopping cart. Each item is represented by a product name and quanity. You are to implement class `Item` and `ShoppingCart` to simulate its operations.

A) Implement the class `Item` as following:

- It has 2 instance variables, `_name` and `_qty`, representing product name and quantity respectively.
- Its initializer method takes in 2 parameters to initialize above two instance variable. Use `None` and `0` as default value for `_name` and `_qty` respectively.
- Use 2 properties `name` and `qty` to encapsulate its instance variables `_name` and `_qty` respectively. Both properties are readable and writable. The setter for qty should ensure that an item's quantity is never negative. Any attempt to set a negative value for qty should be ignored.
- Implement its `__repr__()` method which returns a string in the format of `_name(_qty)`, e.g. "Apple(5)".

In [18]:

```
class Item:

    def __init__(self, name=None, qty = 0):
        self._name = name
        self._qty = qty

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, val):
        self._name = val

    @property
    def qty(self):
        return self._qty

    @qty.setter
    def qty(self, val):
        if val > 0:
            self._qty = val

    def __repr__(self):
        return '{}({})'.format(self._name, self._qty)

```

Test Case 1

In [19]:

```
i = Item('Apple', 5)
print(i)
```

```
Apple(5)
```

Expected output:

```
Apple(5)
```

Test Case 2

In [20]:

```
i = Item('Apple', 5)
i.name = 'Orange'
i.qty = -10
print(i)
```

```
Orange(5)
```

Expected output:

`Orange(5)`

B) Implement the class `ShoppingCart` as following.

- It uses list to keep items in shopping cart, which is initially empty.
- It defines a class variable `PRICE` which has a value of 2.
- Implement its `add()` method which takes in a `item` parameter and appends it to the end of the list.
- Implement its `remove()` method which takes in a `name` parameter and removes only first item matching the name from the list. It returns `True` if an item is removed, else it returns `False`.
- Implement its `item_count()` method which returns the total count of items in the list.
- Implement a class method `amount_due()` which calculates the amount to be paid using formula `amount_due = item_count * PRICE`.
- Implement its `__str__()` method which returns a string representation of its item list. E.g. `[A(2), C(20), B(10)]`

In [21]:

```
class ShoppingCart:
    PRICE = 2

    def __init__(self):
        self._items = []

    def add(self, item):
        self._items.append(item)

    def remove(self, name):
        for i, x in enumerate(self._items):
            if x.name == name:
                del self._items[i]
                return True
        return False

    def item_count(self):
        count = 0
        for x in self._items:
            count = count + x.qty
        return count

    @classmethod
    def amount_due(cls, item_count):
        return item_count * cls.PRICE

    def __str__(self):
        return str(self._items)

```

Test Case 1

In [22]:

```
cart = ShoppingCart()
cart.add(Item('A', 2))
cart.add(Item('B', 5))
cart.add(Item('C', 20))
cart.add(Item('B', 10))
cart.remove('B')
print(cart)          #[A(2), C(20), B(10)]
item_count = cart.item_count()
print(ShoppingCart.amount_due(item_count))      #64
```

```
[A(2), C(20), B(10)]
64
```

Expected output:

```
[A(2), C(20), B(10)]
64
```

Test Case 2

In [23]:

```
cart = ShoppingCart()
cart.add(Item('Apple', 11))
cart.add(Item('Orange', 7))
print(cart.item_count())
print(ShoppingCart.PRICE)
```

```
18
2
```

Expected output:

```
18
2
```