10.009 The Digital World

Term 3. 2017

Problem Set 4 (for Week 4)

Last update: January 13, 2017

- **Problems: Cohort sessions**: Following week: Monday 11:59pm.
- **Problems: Homework**: Same as for the cohort session problems.
- **Problems: Exercises**: These are practice problems and will not be graded. You are encouraged to solve these to enhance your programming skills. Being able to solve these problems will likely help you prepare for the midterm examination.

Objectives

1. Learn nested lists.
2. Learn how to create and use nested lists as tables.
3. Learn how to use looping to process lists.
4. Learn how to traverse sublists.
5. Learn tuples.
6. Learn to use the dictionary data structure.

Note: Solve the programming problems listed using your favorite text editor. Make sure you save your programs in files with suitably chosen names, **and try as much as possible to write your code with good style (see the style guide for python code)**. In each problem find out a way to test the correctness of your program. After writing each program, test it, debug it if the program is incorrect, correct it, and repeat this process until you have a fully working program. Show your working program to one of the cohort instructors.

Problems: Cohort sessions

1. *Lists*: The following problems test your knowledge of lists in Python; no need to write a program for these but you can verify your answers by writing programs.

   (a) Specify the value of `x[0]` after the following code snippet.

   ```
   x=[1,2,3]
   x[0]=0
   y=x
   y[0]=1
   ```

   (b) Specify the value of `x[0]` after the following code snippet.

   ```
   x=[1,2,3]
   def f(l):
     l[0]='a'
   f(x)
   ```

   (c) What is the value of `a[0][0][0][0]` after executing the following code snippet? Write 'E' if there are any errors.

   ```
   x=[1,2,3]
   y=[x]
   a=[y,x]
   y[0][0] = (1,2)
   ```

   (d) Specify the values of expressions (a), (b), (c) and (d) in the following code.

   ```
   x=[1,2,3]
   y1=[x,0]
   y2=y1[:]
   y2[0][0]=0
   y2[1]=1
   y1[0][0] # (a)
   y1[1] # (b)
   y2[0][0] # (c)
   y2[1] # (d)
   ```

   (e) Specify the values of expressions (a), (b), (c) and (d) in the following code.

   ```
   import copy
   x=[1,2,3]
   y1=[x,0]
   y2=copy.deepcopy(y1)
   y2[0][0]=0
   y2[1]=1
   y1[0][0] # (a)
   y1[1] # (b)
   y2[0][0] # (c)
   y2[1] # (d)
   ```

   (f) What is the value of `l` after steps (a), (b), (c) and (d) below?

   ```
   l=[1,2,3]
   l[2:3]=4 # (a)
   l[1:3]=[0] # (b)
   l[1:1]=1 #(c)
   l[2:]=[] # (d)
   ```

2. *Functions: Compound value:* Suppose you save \$100 each month into a saving account with an annual interest rate of 5%. Therefore, the monthly interest rate is 0.05/12=0.00417. After the first month, the value in the account becomes

   $100 * (1 + 0.00417) = 100.417$

   After the second month, the value in the account becomes

   $(100 + 100.417) * (1 + 0.00417) = 201.252$

   After the third month, the value in the account becomes

   $(100 + 201.252) * (1 + 0.00417) = 302.507$

   and so on. Write a function named `compound_value_months` that takes in a monthly saving, an annual interest rate, and the number of months (n), and returns the account value after the n^{th} month. Round the return value to 2 decimal places. Note that this problem is similar to one of the problem you did in the past. The only the different is that the number of months here can be any integer n, and therefore, you need to use loops.

```
>>> ans=compound_value_months(100,0.05,6)
>>> print ans
608.81
>>> ans=compound_value_months(100,0.03,7)
>>> print ans
707.04
>>> ans=compound_value_months(200,0.05,8)
>>> print ans
1630.29
>>> ans=compound_value_months(200,0.03,1)
>>> print ans
200.5
```

3. *Loops:* Write a function named `find_average` that takes a list of lists as input. Each sublist contains numbers. The function returns a list of averages of each sublist and the overall average. For example, if the input list is $[[3,4],[5,6,7],[-1,2,3]]$ then the program returns the list $[3.5, 6.0, 1.333]$ and the overall average 3.625 calculated by summing all numbers in all sublists and dividing by the total count of the numbers.

```
>>> ans=find_average([[3,4],[5,6,7],[-1,2,8]])
>>> print ans
([3.5, 6.0, 3.0], 4.25)

>>> ans=find_average([[13.13,1.1,1.1],[],[1,1,0.67]])
>>> print ans
([5.11, 0.0, 0.89], 3.0)

>>> ans=find_average([[3.6],[1,2,3],[1,1,1]])
>>> print ans
([3.6, 2.0, 1.0], 1.8)
```

```
>>> ans=find_average([[2,3,4],[2,6,7],[10,5,15]])
>>> print ans
([3.0, 5.0, 10.0], 6.0)
```

4. *Lists and nested loops*: Use of a nested list in Python allows you to implement a data structure as a 2-dimensional matrix along with various matrix operations. Write a function named `transpose_matrix`, which takes a n x m integer matrix (i.e. a list with n items each of which is a list of m integer items) as argument, and returns its transposed matrix. For example:

```
>>> a = [[1,2,3], [4,5,6], [7,8,9]]
>>> transpose_matrix(a)
[[1,4,7], [2,5,8], [3,6,9]]
>>>
```

   Use a nested for-loop (i.e. a for loop inside a for loop) and swapping the list item values appropriately to implement this.

5. *Dictionary:* Write a function named `get_details` that takes in a name, a key search, and a list. The list contains a list of phone books entries with each entry is a dictionary. For example

```
>>> phonebook=[{‘name’:‘Andrew’, ‘mobile_phone’:9477865, ‘
    office_phone’:6612345, ‘email’:‘andrew@sutd.edu.sg’},{‘name’:‘
    Bobby’,‘mobile_phone’:8123498, ‘office_phone’:6654321, ‘email’: ‘
    bobby@sutd.edu.sg’}]
```

   The function returns the value of the key search requested for that particular name. It should return `None` if either the name of the key is not found. For example:

```
>>> print get_details(‘Andrew’, ‘mobile_phone’, phonebook)
9477865
>>> print get_details(‘Andrew’, ‘email’, phonebook)
andrew@sutd.edu.sg
>>> print get_details(‘Bobby’, ‘office_phone’, phonebook)
6654321
>>> print get_details (‘Chokey’, ‘office_phone’, phonebook)
None
```

6. *Dictionary:* Write a function named `get_base_counts` that takes a DNA string as input. The input string consists of letters A, C, G, and T (upper case only). The function returns in the form of a dictionary, the count of the number of times each of the four letters A, C, G, and T appear in the input string. For any input string with letters other than A, C, T, and G or lower case letters, the function will return ‘The input DNA string is invalid’.

   Test Cases:

Test case 1
Input: 'AACCGT'
Output: {'A': 2, 'C': 2, 'G': 1, 'T': 1}

Test case 2
Input: 'AAB'
Output: 'The input DNA string is invalid'

Test case 3
Input: 'AaCaGT'
Output: 'The input DNA string is invalid'

Problems: Homework

1. *Lists and loops*: One may use an approximate formula for quickly converting Fahrenheit (F) to Celsius (C) degrees: $C \approx \tilde{C} = (F - 30)/2$. Write three functions, one named `f_to_c` that returns the exact value in Celsius given a temperature in Farenheit; the second named `f_to_c_approx` that returns the approximate value in Celsius given a temperature in Farenheit; and the third named `get_conversion_table` to build a table (ie. a nested list) `conversion`. You should use your `f_to_c` and `f_to_c_approx` functions in your `get_conversion_table` function.

   (a) Build the table such that `conversion[i]` holds a row: `[F, C, CApprox]` where `F` is the temperature in Farenheit; `C` is the corresponding temperature in Celsius, and `CApprox` is the approximate temperature in Celsius. The first column of the table, that is, the temperatures in Farenheit, is 0, 10, 20, $\cdots$, 100. The other columns should contain the values to 1 decimal place.

   ```
   >>> print get_conversion_table()
   [[0, -17.8, -15.0], [10, -12.2, -10.0], [20, -6.7, -5.0],
       [30, -1.1, 0.0], [40, 4.4, 5.0], [50, 10.0, 10.0], [60,
       15.6, 15.0], [70, 21.1, 20.0], [80, 26.7, 25.0], [90,
       32.2, 30.0], [100, 37.8, 35.0]]
   ```

   (b) Build the table such that `conversion[0]` holds a column of temperatures in Farenheit (0, 10, 20, $\cdots$, 100), `conversion[1]` holds a column of corresponding temperatures in Celsius, and `conversion[2]` holds a column of corresponding approximate temperatures in Celsius

   ```
   >>> print get_conversion_table()
   [[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100], [-17.8, -12.2,
        -6.7, -1.1, 4.4, 10.0, 15.6, 21.1, 26.7, 32.2, 37.8],
       [-15.0, -10.0, -5.0, 0.0, 5.0, 10.0, 15.0, 20.0, 25.0,
       30.0, 35.0]]
   ```

   (c) Can you use one of the functions you implemented in one of the cohort exercises to easily convert between the two forms of tables?

2. Write a function named `max_list` which takes a two-level nested list `inlist` of integers as an input and outputs a list `outlist` such that `outlist[i]` is the maximum of all numbers in `inlist[i]`. You can assume that `inlist` is never empty.

```
>>> inp = [[1,2,3],[4,5]]
>>> print max_list(inlist)
[3, 5]
>>> inp = [[1,2,3],[4,5],[32,3,4]]
>>> print max_list(inlist)
[3, 5, 32]
>>> inp = [[3,4,5,2],[1,7],[8,0,-1],[2]]
>>> print max_list(inlist)
[5, 7, 8, 2]
>>> inp = [[100],[1,7],[-8,-2,-1],[2]]
>>> print max_list(inlist)
[100, 7, -1, 2]
>>> inp = [[3,4,5,2]]
>>> print max_list(inlist)
[5]
```

3. Write a Python function named `multiplication_table` that takes a value n and returns an n by n multiplication table. For instance, if n is seven, your program will return a table as follows.

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

The first element of the nested list should be a list that represents the first row of the table, the second element represents the second row and so on. For $n < 1$ your function should return the value `None`.

Test Cases:

Test case 1

Input:	N=7
Output:	[[1, 2, 3, 4, 5, 6, 7], [2, 4, 6, 8, 10, 12, 14], [3, 6, 9, 12, 15, 18, 21], [4, 8, 12, 16, 20, 24, 28], [5, 10, 15, 20, 25, 30, 35], [6, 12, 18, 24, 30, 36, 42], [7, 14, 21, 28, 35, 42, 49]]

Test case 2

Input:	N=1
Output:	[1]

Test case 3

Input:	N=0
Output:	None

Test case 4

Input:	N=2
Output:	[[1, 2], [2, 4]]

Test case 5

Input:	N=-1
Output:	None

4. *Dictionary:* Write a function named `most_frequent` that takes in a list of integers and returns a list of the ones that have the most occurrences. If not one but several numbers have the most occurrences, all of them should be reported. For example:

```
input=[2,3,40,3,5,4,-3,3,3,2,0]
most_frequent = [3]

input=[9,30,3,9,3,2,4]
most_frequent = [9, 3]
```

5. *Dictionary:* A polynomial can be represented by a dictionary. Write a function `diff` for differentiating such a polynomial. `diff` takes a polynomial as a dictionary argument and returns the dictionary representation of its the derivative. If `p` denotes the polynomial as a dictionary and `dp` a dictionary representing its derivative, we have `|dp[j-1] = j*p[j]` for `j` running over all keys in `p`, except when `j` equals 0. Here is an example of the use of `diff`

```
>>> p={0:-3, 3:2, 5:-1}
>>> diff(p)
{2:6, 4:-5}
```

In the above example, the dictionary

`p={0:-3,3:2,5:-1}`

means that the 0th coefficient is -3, the 3rd coefficient is 2, and the 5th coefficient is -1. This can be written as

$$p(x) = -x^5 + 2x^3 - 3 \qquad (1)$$

Test Cases:

Test case 1

Input:	p={0:-3, 3:2, 5:-1}
Output:	{2: 6, 4: -5}

Test case 2

Input:	p={1:-3, 3:2, 5:-1, 6:2}
Output:	{0:-3, 2:6, 4:-5, 5:12}

Test case 3

Input:	p={0:-3, 3:2, 8:2}
Output:	{2:6, 7:16}

Test case 4

Input:	p={0:-4, 2:12, 3:-2, 4:3, 8:2}
Output:	{1:24, 2:-6, 3:12, 7:16}

Test case 5

Input:	p={0:-3, 1:12, 2:-2, 3:2, 10:2}
Output:	{0:12, 1:-4, 2:6, 9:20}

Problems: Exercises

1. *bool, tuple, None*: The following problems test your knowledge of `bool`, `tuple` and `NoneType` types; no need to write a program for these but you can verify your answer by writing programs.

   (a) What are the types of a and b?

   ```
   a=(1)
   b=(1,)
   ```

   (b) List from the variables below those that will evaluate to `False` when converted to `bool`.

   ```
   a='abc'
   b=0+0j
   c=(1,)
   d=''
   e='None'
   f=None
   ```

   (c) What will be the values of expressions at steps (a), (b) and (c) below?

   ```
   t=(1,2,3)
   t+t  # (a)
   t*2  # (b)
   t[1:-1]  # (c)
   ```

2. *Loops*: Two words of equivalent length "interlock" if taking alternating letters from each forms a new word. For example, "shoe" and "cold" interlock to form "schooled." Write

a function named `interlock` which takes `(word1, word2, word3)` as input and return true if and only if `word1` and `word2` interlock and generates `word3`.

Test Cases:

Test case 1

Input: word1='shoe', word2='cold', word3='schooled'
Output: True

Test case 2

Input: word1='shoes', word2='cold', word3='schooled'
Output: False

Test case 3

Input: word1='', word2='cold', word3='schooled'
Output: False

Test case 4

Input: word1='shoes', word2='cold', word3=''
Output: False

Test case 5

Input: word1='', word2='', word3=''
Output: False

Test case 6

Input: word1='can', word2='his', word3='chains'
Output: True

3. Write a function named `throw_dice` for estimating the probability of getting at least one 6 when throwing n dice. Read n and the number of experiments as inputs. Round the return value to 2 decimal places.

4. *Loops, random numbers*: Write a function named `pi_approx_by_monte_carlo` that approximates the value of π using Monte Carlo simulation. The function should take an integer argument as the number of random throws in approximating the wanted value. Round the return value to two decimal place. For a 5-minute video explanation, see `http://youtu.be/VJTFfIqO4TU`. Note that your result may be different from the test cases, i.e. depending on the random seed you use. Furthermore, you should include the boundary of the circle in the calculation of the pi estimation.

   Test Cases:

```
>>> pi_approx_by_monte_carlo(100)
pi = 3.36
>>> pi_approx_by_monte_carlo(100000)
pi = 3.15
>>> pi_approx_by_monte_carlo(10000000) (takes approx. 7 seconds)
pi = 3.14
```

5. Somebody suggests the following game. You pay 1 dollar and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you win r dollars, other you lose your investment. Should you play this game when r = 10? Answer the question by making a

function named `game` that simulates this game. Read r and the number of experiments N as inputs. The function should return `True` if the the answer is 'Yes' and return `False` if the answer is 'No'.

6. Previously, we use the Euler's method to obtain a first order approximation of ODE. However, Euler's method is not very accurate. In practice, an improved Euler's method known as the second-order Runge-Kutta method has been found to work very well in many applications. The second-order Runge-Kutta method from t_n to t_{n+1} is given by

   $$y(t_{n+1}) = y(t_n) + h\left(\frac{1}{2}f(t_n, y(t_n)) + \frac{1}{2}f(t_n + h, y(t_n)) + hf(t_n, y(t_n))\right)$$

   where h is the step size and $\frac{dy}{dt} = f(t, y)$. Now, write a function `approx_ode2` by implementing the Runge-Kutta method with step size, $h = 0.1$, to find the approximate values of $y(t)$ for the following initial value problem (IVP):

   $$\frac{dy}{dt} = 4 - t + 2y, \quad y(0) = 1$$

   from $t = 0$ to $t = 5$ at a time interval of 0.5. Since the above IVP can be solved exactly by the integrating factor method to obtain $y(t) = \frac{1}{2}t - \frac{7}{4} + \frac{11}{4}e^{2t}$, compare your solutions obtained using `approx_ode2` (Runge-Kutta method) with that obtained using `approx_ode2` (Euler's method) by finding the approximation error values up to 3 decimal places.

End of Problem Set 4.