Math3315 /CSE3365 Project 1

due March 6, 2016

Prob. 1 Write a function with the given interface listed below. Fill in the gap in the provided script to achieve the required task listed in the triple-quoted str:

```
def removeDups(L1, L2):
    '''The inputs L1, L2 are two lists,
       return a list that contains all the items in L1 that are not in L2.
    '''
```

Prob. 2 Without using any built-in functions such as `sort, min, max,` from Python and its modules, write a function that uses iteration to output the minimum and maximum values, as well as their positions, of a given list of numeric type. The return data should be a tuple containing two items, each item is a tuple, as ((min, id_of_min), (max, id_of_max))). If there is no tie, the position index should be an integer. If there is a tie, the position index should be a list containing all position indexes of the min or max. The interface of your code is listed below, fill in the gap:

```
def  min_and_max(alist):
    '''The input alist is a list of numeric type, find the min and max as well as the
       positions of min and max of the list,  return them as
       ((min, id_of_min), (max, id_of_max)).
       E.g.,  calling    (a, ida), (b, idb) = min_and_max([3, 5, 7, 1, 2, 7, 7, 6])
       should obtain  (a, ida)=(1, 4),  (b, idb)=(7, [3, 6, 7]),
       Note that the position count starts with 1, not 0.
    '''
```

Prob. 3 Develop a function that meets the requirement in the document string of `sumDigits`:

```
def sumDigits(s):
    '''Input s is a string. The returned value should be the sum of all numerical
       digits in s.
       In addition, if the first non-empty char of s is a minus sign, then the
       returned value should add a minus sign to the sum.
       E.g.,  sumDigits('A33$%4C*+D') should return 10, sumDigits('-+z33$%4-D')
              and sumDigits('  -A+3-3-4*D') should both return -10.
       That is, minus sign matters only when it is the first non-empty char.
       You may need to look up s.isdigit(), s.lstrip(' ') from help(str).
     '''
```

Prob. 4 Develop a function that reverses digits of any given integer.

```
def  reverse_digits(num):
    '''The input num should be an integer, if not, raise an exception and return
       with some instructive message to input int type data.
       When coded correctly,
            reverse_digits(12345) and reverse_digits(+12345) should return '54321',
            and reverse_digits(-12345) should return '-54321'.
       That is, besides reversion of digits, the signs need some addition attention.
       To avoid complication caused by int() removing leading 0's, such as int(0001)=1,
       you should return the reversed digits as a string.
    '''
```

Prob. 5 Given two integers $n1 <= n2$, develop a function that finds all the numbers that are power of two in the interval $[n1, n2]$. (A number is power of 2 if it equals to 2^k for some integer k.) Use your function to find the total number of integers that are power of 2 in $[1, 10^{32}]$.

```
def find_power_of_2(n1, n2):
    '''The inputs are two integers n1 <= n2, find all the numbers in [n1, n2] that
       are power of 2, return them in increasing order in a list.
    '''
```

Hint: You *may* write a function such as

```
def is_power_of_2 (n):
    ''' return true if n is a power of 2, and False if not '''
```

Then call this function for the integers in $[n1, n2]$.

Note: There is a better way than looping over from $n1$ to $n2$ with step=1 to decide if each integer is a power of 2 or not. (You may notice step=2 already can reduce workload by about half.) This can be useful if $n2 - n1$ is huge since it is more efficient, for example, to get all power of 2's in $[1, 10^{1000}]$. If you figure out how to do this, that is great, otherwise, the loops mentioned above should be fine, but you likely cannot afford to go up to 10^{32}, you should try $[1, 10^8]$ (or $[1, 10^7]$) instead. (Updated on Sept 14 to reduce difficulty.)

Prob. 6 Compute e^x using sum of Taylor series,

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + ...$$

where x is real. Write a function for $x \geq 0$ first, set e^x as the returned value of your function. When $x < 0$, compute e^x using three ways in the same function (that is, you write two functions for this problem):

- Since $e^x = 1/e^{-x}$, you can call your own code that computes e^x for $x \geq 0$ to compute the e^x for $x < 0$, use `exp1` to store the value; (Do **not** call `math.exp()` or other built-in exp function.)
- Sum the Taylor series one by one for $i = 0, 1, 2, 3, ...$, use `exp2` to store the sum;
- Sum the Taylor series for even i and odd i separately, then subtract/add them to get e^x, (the subtract/add depends on if you get the sum of odd-i terms using $|x|$ or x), use `exp3` to store the value.

Then returned a tuple containing the values computed using the above three methods as `(exp1, exp2, exp3)`.

Compare the results you get with the `math.exp()` function. For $x \geq 0$ you should get good approximation; for $x < 0$, which of the three methods is the least accurate? (The comparison between `exp2` and `exp3` is meaningful if you sum up to the same number of terms. Here you likely sum up different number of terms, so the less accurate one will need more terms in the sum. Sometimes even with more terms, the needed accuracy may not be reached, due to poor numerical error accumulation via using less robust formulation. The point is that one of `exp2` and `exp3` is computed using a way that leads to larger numerical error accumulation.)

Stop your summation at some i when $\frac{|x|^i}{i!}$ becomes less than some tolerance, use a keyword argument for this tolerance and default it to 10^{-10}.

```
def myexp_pos(x, tol=10**(-10)):
    ''' Input x is nonnegative real, compute exp(x) using Taylor series.
        Note for x>=0 case you only need to return a single float '''
def myexp_neg(x, tol=10**(-10)):
    ''' Input x is negative real, compute exp(x) using Taylor series '''
    ...
    return (exp1, exp2, exp3)
```

Prob. 7 Write a function for the following tasks:

a) Use numpy to generate a random array $x = (z_1, x_2, \cdots, x_n)$ of length n, where x_i's are drawn from normal distribution with mean μ and standard deviation σ. (This should be easily done using numpy.)

b) Compute the 1-norm and 2-norm of the x generated in 7a) in a vectorized manner, i.e., without using loop. As a reminder, for $x = (z_1, x_2, \cdots, x_n)$, the 1-norm and 2-norm are defined, respectively, as

$$\text{norm1} = \|x\|_1 = \sum_{i=1}^{n} |x_i|, \qquad \text{norm2} = \|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}.$$

c) Using the same x generated in 7a), compute

$$\text{sd1} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2 - \bar{x}^2}, \qquad \text{where } \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

Use both a loop (save the value in sd11) and a vectorized operation (save the value in sd12).

d) Using the same x generated in 7a), compute

$$\text{sd2} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2}.$$

Use both a loop (save the value in sd21) and a vectorized operation (save the value in sd22).

The interface of your code is the following, where the mean and deviation are made keyword arguments with given default values:

```
def stat_comp(n, mean=1000, deviation=0.32):
    ''' fill in the gap to fulfill the above listed tasks, return the answers in
        a tuple (x, norm1, norm2, sd11, sd12, sd21, sd22)
    '''
```

General comment:

Partial scripts as well as test codes are provided for each problem. You should not change the file names of these python scripts.

You need to submit your code and the test run log on or before the due day. An easy way to produce a run log is to open a terminal and use the command line: `python inputfile > outputfile` , this will save the screen output for the script `inputfile` in a file named `outputfile`.

You need to change the `inputfile` to a real filename (such as `p1_rmDups.py` etc), for the name of the `outputfile`, use the same header as the inputfile, but changes its file extension from `.py` to `.log`. E.g., for problem 2, you issue the following command-line command in a terminal:
`python p2_minmax.py > p2_minmax.log` .

For your submission, please zip together all the python scripts as well as their related run-log files, and submit only this single zip file to Canvas. Your zip file should contain fourteen files (for each problem you need two files: the completed python script and its run-log). For example, to get full credits for the first two problems, you need these four files: `p1_rmDups.py, p1_rmDups.log; p2_minmax.py, p2_minmax.log`. (Remember to read the `README.txt` file in the provided zip file.)

The instruction for uploading files to Canvas can be found here:
https://guides.instructure.com/m/4212/l/54353-how-do-i-upload-a-file-as-an-assignment-submission-in-canvas.