Homework 2: Iteration, Strings and Lists

Due January 29, 11:59 pm

Worth 10 points

Read this first. A few things to bring to your attention:

1. Start early! If you run into trouble installing things or importing packages, it's best to find those problems well in advance, not the night before your assignment is due when we cannot help you!
2. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.

Instructions on writing and submitting your homework.

Failure to follow these instructions will result in lost points. Your homework should be written in a jupyter notebook file. I have made a template available on Canvas, and on the course website at `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/hw_template.ipynb`. You will submit, via Canvas, a `.zip` file called `yourUniqueName_hwX.zip`, where `X` is the homework number. So, if I were to hand in a file for this, homework 1, it would be called `klevin_hw1.zip`. Contact the instructor or your GSI if you have trouble creating such a file.

When I extract your compressed file, the result should be a directory, also called `yourUniqueName_hwX`. In that directory, at a minimum, should be a jupyter notebook file, called `yourUniqueName.hwX.ipynb`, where again `X` is the number of the current homework. You should feel free to define supplementary functions in other Python scripts, which you should include in your compressed directory. So, for example, if the code in your notebook file imports a function from a Python file called `supplementary.py`, then the file `supplementary.py` should be included in your submission. In short, I should be able to extract your archived file and run your notebook file on my own machine. Please include all of your code for all problems in the homework in a single Python notebook unless instructed otherwise, and please include in your notebook file a list of any and all people with whom you discussed this homework assignment. Please also include an estimate of how many hours you spent on each of the three sections of this homework assignment.

These instructions can also be found on the course webpage at `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/hw_instructions.html`. Please direct any questions to either the instructor or your GSI.

1 Fun with Strings (2 points)

In this problem, you'll implement a few simple functions for dealing with strings. You need not perform any error checking in any of the functions for this problem.

1. A palindrome is a word or phrase that reads the same backwards and forwards (`https://en.wikipedia.org/wiki/Palindrome`). So, for example, the words "level", "kayak" and "pop" are all palindromes, as are the phrases "rats live on no evil star" and "Was it a car or a cat I saw?", provided we ignore the spaces and punctuation. Write a function called `is_palindrome`, which takes a string as its only argument, and returns a Boolean. Your function should return `True` if the argument is a palindrome, and `False` otherwise. For the purposes of this problem, you may assume that the input string will consist only of alphanumeric characters (i.e., the letters, either upper or lower case, and the digits 0 through 9) and spaces. Your function should ignore spaces and capitalization in assessing whether or not a string is a plindrome, so that `tacocat` and `TACO cat` are both considered palindromes.

2. Let us say that a word is "abecedarian" if its letters appear in alphabetical order (repeated letters are okay). So, for example, "adder" and "beet" are abecedarian, whereas "dog" and "cat" are not. Write a function `is_abecedarian`, which takes a single argument in the form of a string and returns `True` if the argument is abecedarian and `False` otherwise. Here you may assume that the input consists only of alphabetic characters and spaces. You function should ignore spaces, so that the string `abcd efgh xyz` is considered abecedarian.

3. Write a function called `count_vowels` that takes a single string as an argument and returns a non-negative integer, the number of vowels that appeared in the string. For the purposes of this question, the vowels are the letters "a e i o u".

2 Fun with Lists (2 points)

In this problem, you'll implement a few very simple list operations.

1. Write a function `list_reverse` that takes a list as an argument and returns that list, reversed. That is, given the list `[1,2,3]`, your function should return the reversed list, `[3,2,1]`. Your function should raise an appropriate error in the event that the input is not a list.

2. This one is a common interview question. Write a function called `binary_search` that takes two arguments, a list of integers `t` (which is guaranteed to be sorted in ascending order) and an integer `elmt`, and returns `True` if `elmt` appears in list `t` and `False` otherwise. Of course, you could do this with the `in` operator, but that will be slow when the list is long, for reasons that we discussed in class. Instead, you should use *binary search*: To look for `elmt`, first look at the "middle" element of the list `t` If it's a match, return `True` . If it isn't a match, compare `elmt` against the "middle" element, and recurse, searching the first or second half of the list depending on whether `elmt` is bigger or smaller than the middle element. **Hint:** be careful of the *base cases*: What should you do when `t` is empty, length 1, length 2, etc.? **Note:** your solution must actually make use of binary search to receive credit, and your solution must not use any built-in sorting or searching functions.

3 More Fun with Strings (2 points)

In this problem, you'll implement some very simple counting operations that are common in fields like biostatistics and natural language processing. You need not perform any

error checking in the functions for this problem.

1. Write a function called `char_hist` that takes a string as its argument and returns a dictionary whose keys are characters and values are the number of times each character appeared in the input. So, for example, given the string "gattaca", your function should return a dictionary with key-value pairs $(g, 1), (a, 3), (t, 2), (c, 1)$. Your function should count *all* characters in the input (including spaces, tabs, numbers, etc). The dictionary returned by your function should have as its keys all and only the characters that appeared in the input (i.e., you don't need to have a bunch of keys with value 0!). Your function should count capital and lower-case letters as the same, and key on the lower-case version of the character, so that G and g are both counted as the same character, and the corresponding key in the dictionary is g.

2. In natural language processing and bioinformatics, we often want to count how often characters or groups of characters appear. Pairs of words or characters are called "bigrams". For our purposes in this problem, a bigram is a pair of characters. As an example, the string `mississippi` contains the following bigrams, in order:

   `'mi', 'is', 'ss', 'si', 'is', 'ss', 'si', 'ip', 'pp', 'pi'`

   Write a function called `bigram_hist` that takes a string as its argument and returns a dictionary whose keys are 2-tuples of characters and values are the number of times that pair of characters appeared in the string. So, for example, when called on the string `mississippi`, your function should return a dictionary with items

   `'mi','is','ss','si','ip','pp','pi'`

   and respective count values

   $$1, 2, 2, 2, 1, 1, 1.$$

   As another example, if the two-character string `ab` occurred four times in the input, then your function should return a dictionary that includes the key-value pair $(a, b), 4$. Your function should handle all characters (alphanumerics, spaces, puncuation, etc). So, for example, the string `cat, dog` includes the bigrams `'t,'`, `', '` and `' d'`. As in the previous subproblem, the dictionary produced by your function should only include pairs that actually appeared in the input, so that the absence of a given key implies that the corresponding two-character string did not appear in the input. Also as in the previous subproblem, you should count upper- and lower-case letters as the same, so that GA and ga both count for the same tuple, (g, a).

4 Tuples as Vectors (4 points)

In this problem, we'll see how we can use tuples to represent vectors. Later in the semester, we'll see the Python `numpy` and `scipy` packages, which provide objects specifically meant to enable matrix and vector operations, but for now tuples are all we have. So, for this problem we will represent a d-dimensional vector by a length-d tuple of floats.

1. Implement a function called `vec_scalar_mult`, which takes two arguments: a tuple of numbers (floats and/or integers) `t` and a number (float or integer) `s` and returns a tuple of the same length as `t`, with its entries equal to the entries of `t` multiplied by `s`. That is, `vec_scalar_mult` implements multiplication of a vector by a scalar.

Your function should check to make sure that the types of the input are appropriate (e.g., that `s` is a float or integer), and raise a `TypeError` with a suitable error message if the types are incorrect. However, your function should gracefully handle the case where the input `s` is an integer rather than a float, or the case where some or all of the entries of the input tuple are integers rather than floats. **Hint:** you may find it useful for this subproblem and the next few that follow it to implement a function that checks whether or not a given tuple is a "valid" vector (i.e., checks if a variable is a tuple and checks that its entries are all floats and/or integers).

2. Implement a function called `vec_inner_product` which takes two "vectors" (i.e., tuples of floats) as its inputs and outputs a float corresponding to the inner product of these two vectors. Recall that the inner product of vectors $x, y \in \mathbb{R}^d$ is given by $\sum_{j=1}^{d} x_j y_j$. Your function should check whether or not the two inputs are of the correct type (i.e., both tuples), and raise a `TypeError` if not. Your function should also check whether or not the two inputs agree in their dimension (i.e., length, so that the inner product is well-defined), and raise a `ValueError` if not.

3. It is natural, following the above, to extend our scheme to the case of matrices. Recall that a matrix is simply a box of numbers. If you are not already familiar with matrices, feel free to look them up on wikipedia or in any linear algebra textbook. We will represent a matrix as a *tuple of tuples*, i.e., a tuple whose entries are themselves tuples. We will represent an m-by-n matrix as an m-tuple of n-tuples. To be more concrete, suppose that we are representing an m-by-n matrix M as a variable `my_mx`. Then `my_mx` will be a length-m tuple of n-tuples, so that the i-th row of the matrix is given (as a vector) by the i-th entry of tuple `my_mx`.

   Write a function `check_valid_mx` that takes a single argument and returns a Boolean, which is `True` if the given argument is a tuple that validly represents a matrix as described above, and returns `False` otherwise. A valid matrix will be a tuple of tuples such that

   - Every element of the tuple is itself a tuple,
   - each of these tuples is the same length, and
   - every element of each of these tuples is a number (i.e., a float or integer).

4. Write a function `mx_vec_mult` that takes a matrix (i.e., tuple of tuples) and a vector (i.e., a tuple) as its arguments, and returns a vector (i.e., a tuple of numbers) that is the result of multiplying the given vector by the given matrix. Again, if you are not familiar with matrix-vector multiplication, refer to wikipedia or any linear algebra textbook. Your function should check that all the supplied arguments are reasonable (e.g., using your function `check_valid_mx`), and raise an appropriate error if not. **Hint:** you may find it useful to make use of the inner-product function that you defined previously.