STAT/CS 94 Fall 2015 Adhikari HW06, Due: 10/7/15

NAME: SID:

Problem 1 Quirky Quantiles

The median of a set of numbers is the number in the middle when we sort the numbers in increasing order. For example, the median of `[-2, 0, 1, 2, 4, 5, 100]` is 2, and the median of `[6, 4, 3, 4, 5]` is 4. If there is an even number of numbers, there are two candidates for the "middle" number; we'll adopt the convention that the median is the mean of the two middle numbers in that situation. Write Python code (without using `np.median`) that defines a function named `median`. It should take a single argument, an array of numbers, and return a number, their median. Assume the given array isn't empty.

Answer:

```
# The median of numbers, an array.  If len(numbers) is even, the mean of
# the two middle elements is returned.
def median(numbers):
    sorted = np.sort(numbers)
    n = len(sorted)
    # Easy way to check if an integer is even: Check whether the remainder
    # when we divide it by 2 is 0.
    if n % 2 == 1:
        middle_index = (n - 1) / 2
        return sorted[middle_index]
    else:
        left_middle_index = (n / 2) - 1
        right_middle_index = n / 2
        return (sorted[left_middle_index] + sorted[right_middle_index]) / 2
```

(The idea is to sort the array first, then find the middle of the sorted copy. How do we figure out how to calculate the index of the middle entry of an array? Just use small examples and extrapolate. In this case, we handle even- and odd-length arrays differently, so consider an array with 3 elements and an array with 4 elements. For the length-3 array, the middle index is 1, so evidently $\frac{n-1}{2}$ is the formula. For the length-4 array, the left-middle index is 1, so evidently $\frac{n}{2} - 1$ is the formula, and the right-middle index is just one bigger than that.)

Problem 2 Concatenation Confusion

Below are several snippets of Python code, and some contain common bugs. Using your best judgement (and careful reading), determine which ones have bugs – that is, which ones don't do what the author probably intended. For the ones with bugs, write a fixed version of the code. The online documentation for Tables (`data8.org/datascience`) and NumPy might be helpful. A backslash (\) at the end of a line indicates that the line is continued on the next line.

(a)
```
t = Table.read_table("some_data.csv")
sleepiest_person_age = t.sort("Hours Slept", descending=True).select["Age"]
```

(b)
```
t = Table.read_table("some_data.csv")
oldest_person_name = t.sort("Age", descending=True)["Name"][0]
```

(c)
```
t = Table.read_table("other_data.csv")
increasing_width_bins = np.arange(0, 100000, 10000) + \
    np.arange(100000, 500000, 50000) + np.arange(500000, 3000000, 500000)
t.select("Salary").hist(bins=increasing_width_bins, normed=True)
```

Answer: *Note:* You didn't need to explain what was wrong with the code, but we've done that.

(a)
```
t = Table.read_table("some_data.csv")
sleepiest_person_age = t.sort("Hours Slept", descending=True)["Age"][0]
```

There are two issues with the existing code.

First, typing `t.sort(...).select["Age"]` into a Python cell results in an error:

`"'method' object not subscriptable"`

. The value of `t.sort(...)` is a Table. The value of `t.sort(...).select` is a *method*, which is a kind of function. Methods can be *called* using parentheses `()`, but they aren't like lists or tables, so it doesn't make sense to index it using square brackets `[]`, just like it doesn't make sense to index a number (or, for that matter, call a list with `()`). ("Subscripting" is another word for indexing.) We could fix this by writing `t.sort(...).select("Age")` instead.

Second, the name `sleepiest_person_age` implies that the author wanted the age of the sleepiest person, but currently it is actually being assigned to a table with a single column of ages, sorted by hours slept. To get the age of the sleepiest person, we get the array of ages with `t.sort(...)["Age"]`, and then take the first element of that array using `t.sort(...)["Age"][0]`. Note that a *Table* is indexable, and it is indexed by column names, so we can say `t.sort(...)["Age"]`, producing an array of ages. (Before, when we said `t.sort(...).select`, we had a method object, not a Table!) Similarly, an array is indexable, and it is indexed by numbers, so we can say `t.sort(...)["Age"][0]` to get the first element of the array of ages. Sorting a table in some way means that each of its columns is sorted in that way, so the first element of that array corresponds to the sleepiest person.

(Arguably, the sleepiest person is the person who slept the *least*. If that's your interpretation of the author's intent, the author should have sorted with `descending=False` instead of `True`; we'll accept that answer, too.)

(b) This code is okay. It's essentially the same as the corrected code for the previous part, but with different columns.

(c)
```
t = Table.read_table("other_data.csv")
increasing_width_bins = list(np.arange(0, 100000, 10000)) + \
    list(np.arange(100000, 500000, 50000)) + list(np.arange(500000, 3000000, 500000))
t.select("Salary").hist(bins=increasing_width_bins, normed=True)
```

The name `increasing_width_bins` implies that the author wanted bins of increasing width (for example, `[0, 1, 2, 3, 5, 7, 10, 13]` would also give bins of increasing width). The author tried to produce a list of smaller bins, a list of wider bins, and a list of yet wider bins, and then concatenate them together. Instead, the original code uses arrays, so the `+` operator tries to *add elements of the arrays together* instead of concatenating them as it would do with lists. So the solution is to convert the arrays to lists before calling `+` on them.

Problem 3 Dubious Dice

Students in a Data Science class are testing whether a die is fair or not. That is, they are testing whether each face of the die appears with chance 1/6 on each roll, regardless of the results of other rolls.

The die is rolled n times. Face 1 appears on a proportion p_1 of the rolls, Face 2 appears on proportion p_2 of the rolls, and so on, so that $p_1 + p_2 + p_3 + p_4 + p_5 + p_6 = 1$. The total variation distance between the empirical distribution of the rolls and the uniform distribution on the numbers 1, 2, 3, 4, 5, and 6 is t.

The students perform a simulation, running numerous replications of n rolls of a fair die and each time computing the total variation distance between the observed distribution and the uniform distribution on $1, 2, \ldots, 6$. You can assume that the number of replications is large enough that the students have a very good approximation to the probability histogram of the total variation distance.

The proportion of replications in which the total variation distance is t or more is 54%.

(a) Write a formula for t in terms of p_1, p_2, p_3, p_4, p_5, and p_6.

(b) If you had to make a conclusion about whether the die was fair, based on the information given, what would you conclude? Why?

(c) The result of the test (*is* / *is not*) statistically significant. Circle one (no reasoning needed).

(d) True or false (and explain): There is about a 54% chance that the die is fair.

Answer:

(a)

$$t = \frac{1}{2} \times (|p_1 - \frac{1}{6}| + |p_2 - \frac{1}{6}| + |p_3 - \frac{1}{6}| \tag{1}$$

$$+ |p_4 - \frac{1}{6}| + |p_5 - \frac{1}{6}| + |p_6 - \frac{1}{6}|) \tag{2}$$

(The total variation distance between two distributions is (half of) the sum of the distances between the probabilities assigned by the distributions to the possible events. The distance between two numbers is just the absolute value of the difference between the numbers. The uniform distribution on 6 die rolls (also known as the distribution of rolls of one fair die) assigns probability $\frac{1}{6}$ to each potential roll.)

(b) Let us formulate this as an hypothesis test and see how the students' procedure fits our framework for hypothesis testing. Our null hypothesis will be that the die is fair, and we'll check whether we would be likely to see something like the data we've seen if the null hypothesis were true.

We first construct a test statistic that is designed to be typically small if the null hypothesis is true. The law of averages says that the empirical distribution of n die rolls will be close to the probability distribution of a single die roll (which, under the null, is the uniform distribution) when n is large. So the difference between the empirical distribution and the uniform probability distribution seems like a reasonable test statistic. The total variation distance is one way we've learned of measuring such a difference.

(It's worth stopping to consider another test statistic that might seem intuitive: The probability of seeing the sequence of die rolls we saw, if the null were true. The problem is that all sequences of n die rolls are equally likely under the null; each has probability $(\frac{1}{6})^n$. So that wouldn't be a useful test statistic, since it would be the same for any observed data!)

Since the data we observed would have been random under the null hypothesis, the empirical distribution is random, and the total variation distance between the empirical distribution and the uniform distribution is random. We could, for example, get really unlucky and roll all 1s with a fair die, in which case the total variation distance would be $\frac{5}{6}$ (plugging in $p_1 = 1$, $p_2 = 0$, $p_3 = 0$, etc in our formula in part (a)). So we can't just check whether the test statistic is 0; we need to see how large it would *typically be* under the null hypothesis. To do this, we just simulate what would happen if we ran our experiment under the null hypothesis: We roll a fair die n times, compute the empirical distribution of the n rolls, and compute the total variation distance of that empirical distribution from the uniform distribution. That gets us one example of a total variation distance. We do that many times, say 10000, and plot those 10000 total variation distances in a histogram. Note that 10000 has nothing to do with n; it is the number of replications of the experiment we perform under the null, which is distinct from the conditions of the actual experiment (which do involve n).

The problem statement says that, in that histogram, 54% of the area lies to the right of the total variation distance we actually saw. That means that, when the null hypothesis is true, 54% of the time we see total variation distances from the uniform distribution even larger than the one we saw, just from random chance. In other words, the test statistic we saw would be totally ordinary if the die were fair.

We have been careful not to say something like "the die is probably fair," because we are technically not entitled to say that. (Part (d) touches on that point.) But certainly the evidence points in that direction, and it's reasonable to conclude that the die is fair, if we're forced to make a choice from only this data. If n is small (say, for example, 1) then we have only weak evidence.

(c) "is not". The result is not statistically significant under any ordinary usage of that term. Typically we say that the P-value of the test statistic is the chance, under the null, of seeing that test statistic or something more extreme (something larger). Under that definition, the P-value we found was .54. We haven't even met the widely-used-but-arbitrary .05 threshold for statistical significance.

(d) False. .54 is just the P-value of this test statistic, and a P-value is *not* the probability that the null hypothesis is false. Rather, it is just the probability of observing the test statistic we observed (or something more extreme) if the null were true. Those two probabilities are just not the same thing at all.

Here is one way to see that P-values cannot be probabilities of hypotheses: In general, we cannot make a statement about the probability of an hypothesis being true without additional background information. For example, say that an extremely honest person has handed you this die and claimed it is fair. Then, before you observe any data, you should assign a high probability to the die being fair, and you should require very convincing evidence of unfairness (that is, observations that are very improbable under the null and very probable under some alternative hypothesis) to lower that assessment to .54. On the other hand, if you personally rigged the die to be unfair, you should start out with a low probability that the die is fair, and require convincing evidence to raise that assessment to .54. Since we have not provided any such background information, there is no way to determine the probability of the hypothesis.

Problem 4 Fancy Functions

The function `map` is used to apply a function to each element of a list, producing a new list containing the results. (It's like Table's `apply` method, but for lists. Note that there is a built-in function in Python 3 called `map` that does something slightly different than what ours will do.) It takes two arguments: first a function `func`, and second a list `the_list`. `func` is itself a function that takes a single argument and returns a value. The `i`th element of the return value of `map` is equal to `func(the_list[i])`. For example, `map(math.sqrt, [1, 16, 4, 9])` has value equal to `[1.0, 4.0, 2.0, 3.0]`. Write Python code that defines `map`. We suggest using a `for` loop.

Answer:

```
def map(func, the_list):
    result = []
    for item in the_list:
        new_item = func(item)
        result = result + [new_item]
    return result
```

Problem 5 Loopy Lookups

When you say something like `my_table["some_column"]`, Python actually calls a function that finds the column labeled `"some_column"` in the table `my_table` and returns it as a NumPy array. (For the curious, the function that gets called is a method of Tables called `__getitem __`. Lists and arrays also have this method, and that's how list and array indexing works.) For this problem you'll implement a similar function, but we'll call it `lookup`. `lookup` takes two arguments: first a Table `the_table`, and second a column name `column_name`, which is a string. It returns the column named `column_name` in `the_table`, which is a NumPy array. If there is no such column, it can do whatever you want. The only restriction is that you cannot use `the_table[column_name]` (or `the_table.__getitem__(column_name)`). Write Python code that defines `lookup` below.

Hint: Read about the `column_labels` and `columns` attributes of Tables.

Answer:

```
def lookup(the_table, column_name):
    for column_index in np.arange(len(the_table.columns)):
        if the_table.column_labels[column_index] == column_name:
            return the_table.columns[column_index]
```

The idea is to search through the list of column labels until we find one that equals `column_name`. When we've found it, we return the corresponding column. Note that we do not iterate over the column labels, but rather over the column indices; when we've found a match, we need to know the index of the matching column so that we can find it in `the_table.columns` and return it.