

NAME:

SID:

Problem 1 Cryptic Connection

Describe what `func1` and `func2` do in the following Python program, what the value of the last expression in the program is, and why we might be interested in that value. (*Hint:* It may be helpful to copy this code into a notebook cell and run it, modifying it or putting in `print()` statements to see how it works. To do that, you'd need to generate your own test tables `t1` and `t2` with some made-up data.)

```

def func1(t, c, v):
    for i in np.arange(t.num_rows):
        rv = t[c][i]
        if rv == v:
            return t.rows[i]
    return False

def func2(t1, c1, t2, c2):
    cs = t1.column_labels + t2.column_labels
    ics = [[]]*len(cs)
    t = Table(ics, cs)
    for i in np.arange(t1.num_rows):
        v = t1[c1][i]
        r1 = t1.rows[i]
        r2 = func1(t2, c2, v)
        if r2 is not False:
            t.append(r1 + r2)
    return t

# Has 2 columns: 'date' and 'number of ice cream cones sold'
t1 = Table.read_table('ice_cream_sales.csv')
# Has 2 columns: 'the date' and 'peak temperature (celsius)'
t2 = Table.read_table('weather.csv')
func2(t1, 'date', t2, 'the date')

```

Answer: `func1` takes a table, a column name, and a value. It returns the first *row* in the table for which the given column matches the given value. (It could be called “`find_row_with_value`”.) `func2(t1, c1, t2, c2)` is basically the same as `t1.join(t2, [c1, c2])`. The final expression is a table with a row for every date for which there was both ice cream sales and temperature data, containing both the number of ice cream sales and the temperature. We might use this to investigate whether ice cream sales were higher when the weather was hot.

Problem 2 Bernoulli Ballots

A polling organization reports that 42% of voters in a state will vote for Candidate A. Assume that their report is based on a random sample of 1500 voters in the state, and that the method of sampling is essentially equivalent to sampling with replacement. Construct an approximate 90%-confidence interval for the percent of the state's voters who will vote for Candidate A. **Answer:** The sample stan-

dard deviation is $\sqrt{p(1-p)}$, where $p = .42$, the proportion of yes-voters in the sample. Since a proportion is a mean and the sample is large, the central limit theorem applies, and a Normal approximation to the distribution of sample averages is reasonable. Plugging in the sample standard deviation, the standard deviation of the distribution of sample averages is approximately $\sqrt{p(1-p)}/\sqrt{1500}$. So if `se = math.sqrt(.42*(1-.42))/math.sqrt(1500)`, an approximate 90%-confidence interval for the proportion is `[stats.norm.ppf(.05, .42, se), stats.norm.ppf(.95, .42, se)]`, or roughly [0.399, 0.441]. We have to multiply this by 100 to get the percentage, so a better answer is roughly [39.9, 44.1].

Problem 3 Fickle Functions

Imagine that you have two Python functions, `f` and `g`, each taking a single real (floating-point) number as its argument and returning a real number. For example, you could imagine that `f` is the function `math.exp` and `g` is the function `math.sin`. Say that you would like to know how dissimilar `f` and `g` are – the “distance” between them, in some sense.

There are lots of ways to measure distance between functions, but one simple way is to ask: Across a range of possible inputs `x` to `f` and `g`, what is the maximum difference between `f` and `g`? In other words, what is the biggest value of $|f(x) - g(x)|$ across a range of possible values of `x`? This notion of distance between functions is called the *Chebyshev distance* (yes, that Chebyshev!) or more commonly today the max-norm distance. The Kolmogorov-Smirnov distance between two distributions is just the max-norm distance between their CDFs, so you already computed a max-norm distance in lab 7. In this problem and the two after it, we’ll explore that a bit more, starting with something simple. First, *describe the following functions:*

- (a)

```
def what(an_array, a_number):
    return np.count_nonzero(an_array <= a_number) / len(an_array)
```
- (b)

```
def hmm(an_array):
    def result_func(input):
        return what(an_array, input)
    return result_func
```

Answer:

- (a) `what` takes an array and a number and returns the proportion of things in the array less than or equal to the number. In other words, it returns the CDF of the dataset `an_array evaluated at a_number`.
- (b) `hmm` returns *the empirical CDF* of the dataset `an_array`. The CDF is itself a function that takes a number and returns the proportion of things in `an_array` less than or equal to that number.

Problem 4 More Max-Norms

This problem continues the previous problem. Describe the following functions. In each case, assume that an argument with a name like `func<something>` is a function that takes a floating-point number as its argument and returns a floating-point number. Any other argument is a floating-point number. *Hint:* Once you figure out a function’s purpose, it may be useful to give it a more informative name and substitute that name where the function appears later. *Hints II: Afterhints:* If you’re confused about how defining a function inside another function (and returning that function) works, try it out yourself. Start with a very simple function, define another function inside it, and convince yourself that nothing breaks.

1.

```
def something(func1, func2):
    def result_func(input):
        return func1(input) - func2(input)
    return result_func
```

```

2. def something_else(func):
    def result_func(input):
        return abs(func(input))
    return result_func

3. def another_thing(func, lowest_input, highest_input):
    NUM_TICKS = 100000
    tick_length = (highest_input - lowest_input) / NUM_TICKS
    values = []
    for input in np.arange(lowest_input, highest_input, tick_length):
        values.append(func(input))
    return max(values)

4. def final_thing(func1, func2, lowest_input, highest_input):
    return another_thing(something_else(something(func1, func2)), \
                        lowest_input, highest_input)

```

5. Write Python code that defines a function named `approximate_ks_distance`. It takes two arguments, each one an array of numbers forming a dataset, and returns the approximate KS distance between the empirical distributions of those two datasets. *Hint:* You may want to use one or several of the functions defined above in the previous problem or this problem. *The Hint: Part II:* Those functions are pretty powerful tools. The staff solutions were only a few lines long.

Answer:

- (a) `something` returns a new function that returns the difference between its first argument *function* and its second argument *function* when applied to an input. That is, `something(f, g)(x)` = $f(x) - g(x)$.
- (b) `something_else` returns a new function that returns the absolute value of the argument function when applied to an input. That is, `something_else(f)(x)` = $\text{abs}(f(x))$.
- (c) `another_thing` returns the (approximate) maximum value of its argument function over the range `[lowest_input, highest_input]`. It is approximate because it only checks 100000 evenly-spaced values, not literally every number between `lowest_input` and `highest_input`, so in theory it could miss the biggest value.
- (d) `final_thing` returns the (approximate) max-norm distance between two functions over a given range of numbers.
- (e) Here is a somewhat verbose way to write the function:

```

def approximate_ks_distance(dataset1, dataset2):
    cdf1 = hmm(dataset1)
    cdf2 = hmm(dataset2)
    smallest_input = min(min(dataset1), min(dataset2))
    largest_input = max(max(dataset1), max(dataset2))
    return final_thing(cdf1, cdf2, smallest_input, largest_input)

```

This is a different way to compute the KS distance between two datasets than the method you saw in lab 7. It's approximate, since it doesn't necessarily check the differences between CDFs at exactly the right points.

Problem 5 Groovy Graphs

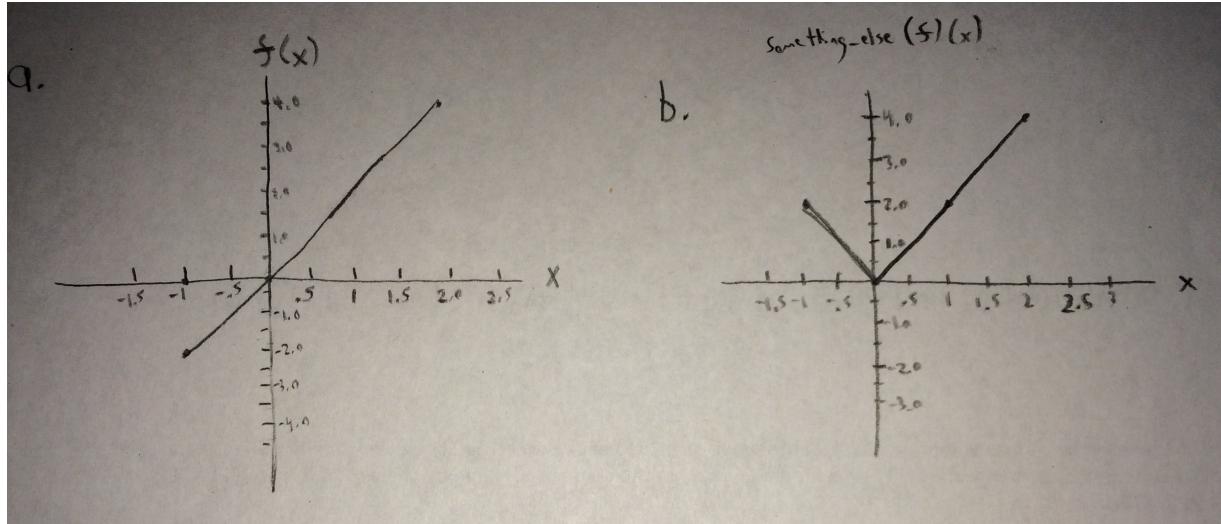
Draw the graphs that appear when the following code is run. (Feel free to investigate plotting in general on your computer, but please don't just copy this exact code into a notebook and draw what you see. That's lame.) Parts of this problem use code from the previous two problems. *Assume that the function definitions in the previous two problems have been executed before this code runs.*

Note: We've seen `plots.xlabel(<some string>)` and `plots.ylabel(<some string>)` before, but for reference: When you call them, they modify the most recent plot you created, putting the *string* you pass as an argument as a label on the x or y axis, respectively.

- ```
(a) def f(x):
 return 2*x
ticks = np.arange(-1.0, 2.0, 0.1)
t = Table([ticks], ['x'])
t['f(x)'] = t.apply(f, 'x')
t.plot('x')
plots.xlabel('x')
plots.ylabel('f(x)')

(b) t['something_else(f)(x)'] = t.apply(something_else(f), 'x')
t.select(['something_else(f)(x)', 'x']).plot('x')
plots.xlabel('x')
plots.ylabel('something_else(f)(x)')
```

Answer:



Note: Details like the number and size of axis ticks were not graded for this problem. You should have axis labels, labeled ticks of some kind, and the right function displayed.

## Problem 6 Buggy Bootstrap?

A wise old man wants to know the average length of the hairs in his long beard. To measure them, he has to pluck them out, so he wants to measure only a few. Being a good statistician, he carefully takes a simple random sample of 100 hairs (you can assume they're taken with replacement), plucks them, measures them, and places the lengths in a *column* named `length` in a *table* called `hair`. However, his Python is a little rusty (in the last few years he has jumped on the Scala bandwagon), and he runs the following code to estimate

a 99.2% confidence interval for the average length of hairs in his beard. He intends for the last line of code to evaluate to a two-element list, where the 0th element is the left side of the 99.2% confidence interval, and the 1st element is the right side of that interval. Find as many bugs as you can in the code, and describe how you would fix each one.

```
REPETITIONS = 10000
actual_mean = np.mean(hair)
bootstrap_means = []
for rep in REPETITIONS:
 resample = hair.sample()
 resample_mean = np.mean(resample)
 bootstrap_means.append(resample_mean)
left_side_index = int((1 - .992)*REPETITIONS)
right_side_index = int(.992*REPETITIONS)
[bootstrap_means[left_side_index], bootstrap_means[right_side_index]]
```

**Answer:**

- (1) `actual_mean` should be `np.mean(hair['length'])` and `resample_mean` should be `np.mean(resample['length'])`.
- (2) The `for` loop should loop over `np.arange(REPETITIONS)`, not `REPETITIONS`.
- (3) `resample` should be `hair.sample(hair.num_rows, with_replacement=True)`; otherwise it's just a shuffled copy of `hair`.
- (4) We are trying to compute an interval that covers 99.2% of the bootstrap resampling distribution of the means. So we want the 0.4th and 99.6th percentiles of the resampling distribution. The current code doesn't sort `bootstrap_means` to compute percentiles, *and* it doesn't use the right percentiles. So we should replace the last 3 lines with:

```
sorted_means = np.sort(bootstrap_means)
left_side_index = int((1 - .996)*REPETITIONS)
right_side_index = int(.996*REPETITIONS)
[sorted_means[left_side_index], sorted_means[right_side_index]]
```

We could also use the function `np.percentile` for this:

```
[np.percentile(bootstrap_means, 1-.996), np.percentile(bootstrap_means, .996)]
```

You could also have just written out the fixed code, since there were so many bugs.

Though the spirit of the problem was to use the bootstrap percentile method, you could also have used a Normal approximation, since we were computing a confidence interval for a mean, and the Central Limit Theorem says that the bootstrap resampling distribution of the mean is close to a Normal distribution. To do that, you would compute the mean and standard deviation of `bootstrap_means` and then find an interval that covers 99.2% of the Normal distribution with that mean and SD using `stats.norm.ppf`. Or you could skip the bootstrap resampling altogether and directly plug in the mean and standard deviation (the latter divided by  $\sqrt{\text{hair.num\_rows}}$ ) of `hair['length']` for the mean and standard deviation of the Normal distribution, and again use a 99.2% interval for your confidence interval.

### Problem 7 Pessimistic Polling

Suppose you are going to conduct an opinion poll based on a sampling scheme that is approximately the same as sampling at random with replacement. What is the minimum sample size that will ensure that 95%-confidence intervals for population proportions will have a total width of 0.04 or less? *Hint:* Yes, you can answer this question. **Answer:** We saw in lecture that the standard deviation of an average of  $n$  things

that are 0 or 1 (a.k.a. the standard error of a sample proportion) is  $\sqrt{\frac{p(1-p)}{n}}$ . This depends on  $p$ , which is what we're calling the probability of seeing a 1. That's a population parameter, which we might know nothing about before we take the sample, so it looks like we're stuck. But it is biggest when  $p = 1/2$ , so we can actually calculate that the *worst-case standard error* is  $\sqrt{\frac{\frac{1}{2}(1-\frac{1}{2})}{n}} = \frac{1}{2\sqrt{n}}$ . If we choose a sample size so that the confidence interval is only 0.04 wide when the population standard deviation is this value, then the confidence interval width could only get smaller if  $p \neq 1/2$  in the population.

We would like the standard error to be at most 0.01, since the width of a 95% confidence interval (using a Normal approximation) is (roughly) 4 standard errors – 2 on either side of the average. (If you want to be precise, solve  $2 \times \text{stats.norm.ppf}(.975) \times SE = .04$ . for  $SE$ .) So  $0.01 \geq \frac{1}{2\sqrt{n}}$ , which is solved by  $n \geq 2500$ . So the minimum required sample size is roughly 2500.