STAT/CS 94 Fall 2015 Adhikari Final Material

1 Introduction

This document lists the broad *programming* topics you should know about for the final. It also lists, in cheat-sheet form, the Python syntax and functions you might need to understand or use on the final. You will have access to the cheat sheet when you take the exam (but not the list of topics). Outside materials aren't allowed on the exam; we'll print the cheat sheet for you.

2 Programming topics

2.1 General Python stuff

1. Basic expressions: Strings, numbers, booleans (True/False values)
2. Basic arithmetic on numbers; basic logical operators on booleans
3. Assigning names to values with `=`
4. Calling functions with `()`
5. Accessing things: indexing lists and tables with `[]`; accessing attributes and methods with `.`
6. Assigning other things with `=`: assigning slots in lists with `[]`; adding columns to tables with `[]`
7. Running code conditionally with `if` statements
8. Running code iteratively with `for` loops
9. Defining functions with `def` statements
10. Returning values from functions
11. Encapsulating blocks of code (or single "ideas") into functions
12. Thinking about functions as values that happen to be callable with `()`, and passing functions as arguments to other functions ("higher-order" functions)
13. Understanding Python expressions by breaking them down into simple parts

2.2 Lists, arrays, and tables

1. Making lists with `[]`
2. Making arrays from lists with `np.array(...)`; making arrays of consecutive numbers with `np.arange(...)`
3. Making tables by reading data files with `Table.read_table(...)`; directly using the `Table(...)` function
4. Zero-based indexing for lists and arrays, and the left-inclusive / right-exclusive behavior of `np.arange` and slice indexing
5. Producing a concatenated list from two lists with `+`
6. Transforming each element of a list using a list comprehension, forming a new list
7. Differences between arrays and lists
8. Basic functions that do things with arrays, like `np.sum`, `np.mean`, `np.median`, and `np.diff`; operators like `+`, `-`, `*`, `/`, `**`, and `&` acting on two arrays or on an array and a single value

9. Statistical functions, like `stats.norm.cdf`, `stats.binom.cdf`, and `np.percentile`. (This is not a full list. All the statistical functions you might need to use in the exam are in the cheat sheet. The cheat sheet does not necessarily teach you the statistical jargon, though. You should know, for example, what a CDF and a percentile are.)
10. Accessing columns of a table, which are just arrays
11. Making a table with a subset of the columns in an existing table with `.select`
12. Making a table with a subset of the rows in an existing table with `.where`; using logical operations on columns in combination with `.where` to filter rows according to logical conditions
13. Making a bar chart from a categorical-valued table with `.barh`
14. Making a histogram from a table with `.hist`; making a density histogram; controlling the bin widths
15. Applying a function to each element of a column in a table with `.apply` (a higher-order function)
16. Joining two tables with `.join`
17. Grouping rows of a table together with `.group`; aggregating the groups with a function (making `group` a higher-order function)
18. Creating a "contingency table" or "pivot table" on two categorical columns of a table with `.pivot`; aggregating the contents of each list-valued cell in the resulting table with a function (making `pivot` a higher-order function)
19. Sampling rows of a table (producing a new table) with `.sample`
20. Repeatedly sampling from a table, computing a statistic, and displaying the empirical distribution in a histogram (to approximate the probability distribution of the statistic under sampling)

3 Selected lab questions to review

Here is a list of some of the lab content that you might find particularly useful to review. (If you need a new copy of a lab, you can email a GSI; be sure to mention the email address you use to log in to `ds8.berkeley.edu`.)

- Lab 2: questions 11, 13
- Lab 3: questions 2, 8
- Lab 4: question 6
- Lab 5: questions 6, 7, 8, 11
- Lab 6: questions 2, 3, 8, 9, 12
- Lab 7: questions 2, 3, 6
- Lab 8: questions 1.5, 2.1, 2.2, 2.4, 2.5, 2.7 (Note that the bootstrap procedure discussed in this lab is a little different from the Bootstrap Percentile Method described in the text. Either is valid, but if you only remember one, remember the one in the text.)

4 Python cheat sheet

This cheat sheet is organized by topic, though some examples serve double-duty to conserve space. Rather than give exhaustive documentation, we have created examples that demonstrate behavior that might be hard to remember.

4.1 General Python stuff

```
"Hello, world!" # A string-valued expression
1 # An integer-valued expression
1.2 # A float-valued expression
True # A boolean-valued expression
3 ** 4 # An expression whose value is 3 to the 4th power
pow(3, 4) # A function call expression, also 3 to the 4th power
17 % 5 # An expression whose value is 2, the remainder when 17 is divided by 5
2 + 2 == 5 # An expression whose value is False
(17 % 5) == 2 # An expression whose value is True
"3.5" # An expression whose value is a string
float("3.5") # An expression whose value is the number 3.5
str(3.5) # An expression whose value is the string "3.5"
str(3.5) + str(2.1) # An expression whose value is the string "3.52.1"
x = [1,2,3] # An assignment statement; [1,2,3] is a list expression
len(x) # A function call expression whose value is 3, the length of the list x
len([1,2,3]) # Also a function call expression with value 3
x[pow(2,1)] # An indexing expression with value 3
x[0:2] # A slice-indexing expression with value [1,2]
['a','b','c'][1:3] # A slice-indexing expression with value ['b','c']
"abc"[1:3] # Strings are indexable; this expression has value "bc"
"abc"[1] # An indexing expression with value "b" (a length-1 string)
x + [4,5] # An expression with value [1,2,3,4,5]; adding lists concatenates
x[0] = 4 # An index assignment statement
t = Table([[0,1,4,9], [0,1,8,27]], ['squares', 'cubes']) # Making a table
t['squares'] # An indexing expression with value equal to np.array([0,1,4,9])
t['powers of two'] = [1,2,4,8] # An index assignment statement
# Attribute access expression with value ['squares, 'cubes', 'powers of two']:
t.column_labels
t.num_rows # The number of rows in t
len(t.rows) # Also the number of rows in t; rows is a list of Row objects in t

# A function that returns "fizz" if its argument is even, "buzz" otherwise.
# Its name is fizz_if_even.  It takes a single argument which we have given
# the name an_integer; an_integer is defined (as though by an assignment
# statement with =) while fizz_if_even is being called, but not outside.
def fizz_if_even(an_integer):
    remainder_after_division_by_two = an_integer % 2
    if remainder_after_division_by_two == 0:
        return "fizz"
    else:
        return "buzz"
should_be_fizz = fizz_if_even(2)
should_be_buzz = fizz_if_even(3)
an_integer*3 # An error: an_integer is not defined here!

# A function that is erroneously missing a return statement and does nothing
def multiply_by_three(a_number):
    3*a_number
# If we call this function and use the value of the call expression, the value
# is nothing, not three times the argument.
should_have_been_six = multiply_by_three(2) # Doesn't do what we wanted!

# A function that causes a density histogram with bins -2:0,0:1,1:4 to be made.
def make_a_histogram(table):
    table.hist(bins=[-2,0,1,4], normed=True)
# Note: No histogram has been made at this point.  Calling the function
# executes the code inside it and makes a histogram appear.
make_a_histogram(Table([[0,0,2,3]],['nums']))
# A second histogram is made if the function is called again.
make_a_histogram(Table([[1.2,1.3,3.2,-1.2]],['other_nums'])

# A function that returns a list in which func has been applied, and then
# applied again, to each element of the_list.  Uses a for loop.  When the for
# loop is reached, the code inside the for loop is executed once for each
# element of the_list, and the name an_item is set equal to a different element
# of the_list each time the code inside the loop is executed.  Once the for
# loop has been executed len(the_list) times, the next line (return result, in
# this case) is executed.
# So apply_twice(math.sqrt, [16, 81]) equals [2.0, 3.0].
def apply_twice(func, the_list):
    result = []
    for an_item in the_list:
        result = result + [func(func(an_item))]
    return result

# A list comprehension.  Builds a list by evaluating the first expression with
# the variable (in this case x) set to each value in the last expression.
even_positive_integers_less_than_nine = [2*x for x in np.arange(1, 5, 1)]
```

4.2 Array-specific stuff

```
small_primes_array = np.array([2,3,5,7,11])
odd_positive_integers_less_than_eight = np.arange(1, 9, 2)
# Calling np.array on an array produces a copy of the same array:
np.array(small_primes_array) # The same as small_primes_array
np.array([1,2,3]) + np.array([2,3,4]) # An array equal to np.array([3,5,7])
np.sum(np.array([-2.2,1.0,0.0])) # -1.2
np.mean(np.array([-2.2,1.0,0.0])) # -0.4
np.diff(np.array([-1,3,2,5,5,0])) # An array equal to np.array([4,-1,3,0,-5])
np.array([1,2,3]) - 1 # An array equal to np.array([0,1,2])
np.array([1,2,3]) ** 2 # An array equal to np.array([1,4,9])
2 ** np.array([1,2,3]) # An array equal to np.array([2,4,8])
np.array([1,2,3]) >= 2 # An array equal to np.array([False,True,True]))
# An array of booleans equal to np.array([True,False,False]); element 1 of
# array 1 is logically AND-ed with element 1 of array 2, and so on
np.array([True,False,False]) & np.array([True,True,False])
np.count_nonzero(np.array([True, False, True])) # 2, the number of True values
# np.arange(n) is short for np.arange(0, n, 1).  For example:
counter = 0 # After the for loop, equal to 0 + 1 + 2 + ... + 99, or 4950.
for index in np.arange(100):
    counter = counter + index
```

4.3 Table-specific stuff

```
u = Table.read_table('some_data_file.csv') # A table built from a data file
t['squares'][2] # An expression with value 4 (see definition of t above)
```

```
t['squares'] + t['cubes'] # An expression with value np.array([0,2,12,36])
t['squares'] > 3 # An expression with value np.array([False,False,True,True])
# A table with only the first row of t:
t.where(np.array([True,False,False,False]))
t.where(t['squares'] > 3) # A table with only the last and 2nd-to-last rows of t
t.select(['squares']) # A table with only one column, squares
# A bar chart with a length-4 bar for apples, a length-11 bar for oranges, etc.
v = Table([[4,11,2],['apples','oranges','kiwis']],['count','fruit'])
v.barh('fruit')
t.apply(math.sqrt,'squares') # An array with value np.array([0.0,1.0,2.0,3.0])
# Modifies t to add a row to it with 'squares' .25, 'cubes' .125, and 'powers
# of two' 2**(1/2).  The argument can also be a row of a table or a whole
# table, as long as the number and names of their columns are the same.
t.append([.25, .125, 2**(1/2)])

# Demonstrating join().  In x.join('a',y,'b'), we go through the rows of table
# x one by one, building a resulting joined table.  We look at the value K of
# column 'a' in that row.  Then we look for the first row in table y where the
# value of column 'b' is K.  If there is such a matching row, we add the row
# from x to the joined table and we adjoin the columns in the matching row to
# that row.
w = Table([['Ann','Bob','Cathy','Dan'],['apples','oranges','peaches','apples']],
    ['name','favorite fruit'])
j = w.join('favorite fruit',v,'fruit')

# Demonstrating group().  We choose a column and make a new table with one
# row for each unique value in that column; rows with the same value of that
# column are squashed together.  For each other column, the value in each new
# row is the list of values of that column for the rows that were squashed
# together.
w.group('favorite fruit')
# We can have group() apply a function to each value list.  For example, len
# will tell us the number of things in the list, so we can count the number of
# people with each favorite fruit by:
w.group('favorite fruit', collect=len)

# Demonstrating pivot().  Say that we also know everyone's favorite color, and
# we want to know who has each <favorite color, favorite fruit> pair (for
# example, who likes apples and red).  w.pivot() does this by producing a new
# table summarizing w in that way.  Say we want colors to appear on the
# vertical axis (i.e. each color gets a row in the resulting table) and fruits
# to appear on the horizontal axis (i.e. each fruit gets a column in the
# resulting table).  And in each cell of the table we want a list of the names
# of the people who like that <color, fruit> pair.  Then we would say:
w['favorite color'] = ['red','blue','red','blue'] # Set up the table.
w.pivot('favorite fruit', 'favorite color', 'name')
# Now say we want to know the number of people in each category instead of
# the list of their names.  As with group(), we can pass a function to be
# applied to each list:
w.pivot('favorite fruit', 'favorite color', 'name', collect=len)

# Demonstrating sample().  If t is any table, t.sample(<args>) is a new table
# with the same columns as t but with 0 to several repetitions of each row.
# The total number of rows in the sample is the first argument.  (If no
# argument is given, the total number of rows in the sample is the number of
# rows in the table.)  The second argument is a boolean value; if it is True,
# the sampling is done with replacement, and otherwise it is done without
# replacement.  (By default it is False.)  sample() doesn't care what the
# content of the rows is, and it doesn't shuffle values within columns.  It
# just selects entire rows.  The order of the selected rows is always random.
sampling_table = Table(['Peter', 'Paul', 'Mary'],['name'])
three_random_names = sampling_table.sample(3, with_replacement=True)
two_distinct_random_names = sampling_table.sample(2, with_replacement=False)

# Demonstrating scatter().  If t is a table with a column named 'z', then
# t.scatter('z') makes one scatter plot for each column in t other than z.
# Say there is one other column named 'h'.  Then each row in the table will be
# a point in the scatter plot; the horizontal-axis value of each point is
# the value of z on that row, and the vertical-axis value of each point is the
# value of h on that row.  Including the argument fit_line=True will add
# a least-squares best fit line to the plot.
x_values = np.arange(-2, 7, .1)
negatives_table = Table([x_values, -1*x_values], ['x', '-1*x'])
negatives_table.scatter('x', fit_line=True)
```

4.4 Statistical functions

```
# The probability that something with a Normal distribution with mean -0.5 and
# SD 2 is less than or equal to -1.2.
stats.norm.cdf(-1.2, -0.5, 2)
# The number such that that something with a Normal distribution with mean 2.3
# and SD 0.1 is less than or equal to that number with probability .4.
stats.norm.ppf(.4, 2.3, 0.1)
# A random value (a floating-point number) drawn from a Normal distribution
# with mean 0.5 and SD 1.0.
np.random.normal(0.5, 1.0, size=2)
# An **array** of 2 random values drawn from a Normal distribution with mean
# 0.5 and SD 1.0.
np.random.normal(0.5, 1.0, size=2)

# The probability that a binomial random variable with n=11 and p=.45 is 3.
stats.binom.pmf(3, 11, .45)
# The probability that a binomial random variable with n=11 and p=.45 is 0, 1,
# 2, or 3.
stats.binom.cdf(3, 11, .45)

# The 50th percentile of the numbers [5.0, -1, 2, 3, 0.9], which is 2.
np.percentile([5.0, -1, 2, 3, 0.9], 50)
# The 75th percentile of the numbers [5.0, -1, 2, 3, 0.9], which is 3.
np.percentile([5.0, -1, 2, 3, 0.9], 75)

# multinomial() lets us simulate a sequence of draws (taken at random with
# replacement) from a probability distribution on a categorical random
# variable.  It returns the number of times each category was drawn in an array
# (so the 0th element is the number of times category 0 was drawn, etc).  The
# first argument is the number of draws.  The second argument is a list of
# probabilities, one for each category (so it must sum to 1).
# For example, this simulates a situation in which category 0 has probability
# 9/16, category 1 has probability 3/16, etc, and there are 50 draws.  The
# value of the following is a random 4-element array of integers summing to 50.
# The 0th element of the result is the number of times category 0 appeared in
# this simulation, the 1st is the number of times category 1 appeared, etc.
np.random.multinomial(50, [9/16, 3/16, 3/16, 1/16])
```