STAT/CS 94 Fall 2015 Adhikari Midterm Material

1 Introduction

This document lists the broad *programming* topics you should know about for the midterm. It also lists, in cheat-sheet form, the Python syntax and functions you might need to understand or use on the midterm.

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)

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. Differences between arrays and lists
7. Basic functions that do things with arrays, like `np.sum`, `np.mean`, and `np.diff`; operators like `+`, `-`, `*`, `/`, `**`, and `&` acting on two arrays or on an array and a single value
8. Accessing columns of a table, which are just arrays
9. Making a table with a subset of the columns in an existing table with `.select`
10. 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
11. Making a bar chart from a categorical-valued table with `.barh`
12. Making a histogram from a table with `.hist`; making a density histogram; controlling the bin widths
13. Applying a function to each element of a column in a table with `.apply` (a higher-order function)

14. Joining two tables with `.join`
15. Grouping rows of a table together with `.group`; aggregating the groups with a function (making `group` a higher-order function)
16. 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)
17. Sampling rows of a table (producing a new table) with `.sample`
18. 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 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.

3.1 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
(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
```

3.2 Array-specific stuff

```
small_primes_array = np.array([2,3,5,7,11])
odd_positive_integers_less_than_nine = np.arange(1, 9, 2)
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
counter = 0 # After the for loop, equal to 0 + 1 + 2 + ... + 99, or 4950.
for index in np.arange(100):
    counter = counter + index
```

3.3 Table-specific stuff

```
u = Table.read_table('some_data_file.csv') # A table built from a data file
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])

# 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.  So table j below has 3 rows, one for Ann, Bob, and Dan; each row
# has 'name' and 'favorite fruit' from table w and the count of that person's
# favorite fruit from table v.  Cathy is missing from j because her favorite
# fruit didn't appear in v.  There is no row in j with favorite fruit kiwi,
# since no one in w had that favorite fruit.
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.  So the following expression's value is a table with 3 rows, for
# apples, oranges and peaches; the row with favorite fruit 'apples' has 'name'
# equal to ['Ann','Dan'], the row with favorite fruit 'oranges' has 'name'
# equal to ['Bob'], and the row with favorite fruit 'peaches' has 'name' equal
# to ['Cathy'].
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 the following expression's
# value is equal to:
# Table([['apples','oranges','peaches'],[2,1,1]],['favorite fruit','name len'])
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')
# The result looks like this:
# favorite color | apples name | oranges name | peaches name
# blue           | ['Dan']     | ['Bob']      | None
# red            | ['Ann']     | None         | ['Cathy']
# 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)
# The result looks like this:
# favorite color | apples name | oranges name | peaches name
# blue           | 1           | 1            | 0
# red            | 1           | 0            | 1

# Demonstrating sample().  Say that we want to sample numbers from 1 to N,
# inclusive on both sides.  We make a table with one column containing those
# values.  sample() returns a new table with the same columns as the original
# but with 0 to several repetitions of each row.  The total number of rows is
# the first argument.  Whether the sampling is done without replacement (a
# row can be selected only once) or with replacement (each time a row is
# is selected we choose among all the rows) is controlled by the second
# argument.  sample() doesn't care what the content of the rows is, but we'll
# use a simple table with 1 column in these examples.  The sampling table could
# also be a table of people or zip codes with several columns, if that's what
# we wanted to sample.
N = 6 # Maybe we're simulating a die roll.
potential_numbers = np.arange(1,N+1,1)
sampling_table = Table([potential_numbers],['nums'])
three_random_rolls = sampling_table.sample(3, with_replacement=True)
three_distinct_random_rolls = sampling_table.sample(3, with_replacement=False)
# We could get an array of numbers like this:
two_random_rolls_array = sampling_table.sample(2, with_replacement=True)['nums']
# We could get a single number like this:
one_random_number = sampling_table.sample(1, with_replacement=True)['nums'][0]
```