INTRODUCTION TO PROGRAMMING
WEEK 5 - DATA STRUCTURES AND FUNCTIONS (PART II)

DATA STRUCTURES

This week we are going to spend some more time on data structures, because I've found that this is where people get tripped up the most when learning programming. At the end of class, we will do several exercises together to solidify our understanding of these extremely important data structures.

HANDY LIST FUNCTIONS

Does anyone remember, from last week's notes, how to modify every element in a list?

The method we introduced, involved using a for loop to iterate over every element of the loop.

```
my_list = ['albert', 'sauwei', 'hwang']
counter = 0

for name in my_list:
 my_list[counter] = name.upper()
 counter += 1
```

When we do this, we need to create a separate variable, `counter`, which gets incremented each iteration so that we can access the list element through its index.

Getting the index for a list item is a pretty common use of the list data structure, so lists have a built in function called `enumerate()` that allows us to simplify the syntax.

```
for <tuple variable> in enumerate(<list>):
```

The enumerate function returns a list of tuples. In each tuple, the first element is the object index, and the 2nd is the object itself. For instance:

```
for name_tuple in enumerate(my_list):
  print name_tuple
```

This prints:

```
(0, 'albert')
(1, 'sauwei')
(2, 'hwang')
```

This isn't really a significant shortcut unless coupled with another shortcut for referencing the elements of the tuple:

```
for <index variable>, <object variable> in enumerate(<list>):
```

Another, more obvious example may help to clear this up:

```
my_list = ['albert', 'sauwei', 'hwang']

for index, name in enumerate(my_list):
  print name + ' is the number ' + str(index + 1) + ' item'
```

The above program prints out:

```
albert is the number 1 item
sauwei is the number 2 item
hwang is the number 3 item
```

Let's see the capitalization subroutine (from above), but this time using enumerate:

```
for index, name in enumerate(my_list):
 my_list[index] = name.upper()
```

As you can see, the first variable in the comma separated list before the **`in enumerate`** key words is the index variable that each list element's index is re-assigned to in each loop. The second variable is the variable that the actual list element is assigned to in each loop.

THE SPLIT FUNCTION

The split function is actually a string method but without introducing data structures, this function would have been very confusing. Let's take an example

```
my_string = 'albert,sauwei,hwang'
my_list = my_string.split(',')

print my_list[2] # 'hwang'
print my_list # ['albert', 'sauwei', 'hwang']
```

As the name implies (and the example demonstrates), the split function is called from a string and takes a single parameter, a string by which to split the parent string into sub-strings. **It then creates a list with those split out sub-strings**. A few more examples will clear this up:

```
my_string = 'oh when the saints go marching in!'
my_list = my_string.split(' ') # split by a single space
print my_list

# ['oh', 'when', 'the', 'saints', 'go', 'marching', 'in!']

my_string = 'who do you call? Ghost busters!'
my_list = my_string.split('?') # split by question marks
```

```
print my_list

# [‘who do you call’, ‘Ghost busters!’]

my_string = ‘albert,sauwei,hwang’
my_list = my_string.split(‘e’)
print my_list

# ['alb', 'rt,sauw', 'i,hwang']
```

This is a handy way of creating lists when you have a long string and need to programmatically access parts of it.

HANDY DICTIONARY FUNCTIONS

Does anyone remember, from last week’s notes, how to modify every element in a dictionary?

The method we introduced, involved using a `for loop` to iterate over every key of the dictionary and accessing its corresponding values for manipulation.

```
my_dict = {‘bugs’: ‘rabbit’,
           ‘elmer’: ‘human’,
           ‘wiley’: ‘coyote’,
           ‘tom’: ‘cat’,
           ‘jerry’: ‘mouse’}

for key in my_dict:
 my_dict[key] = my_dict[key].upper()
```

Often, you’ll want to access either the key or value in a dictionary element, not just the keys. To do this, use the **`iteritems`** function which returns a list of tuples (key, value) for each dictionary element:

```
for <key variable>, <value variable> in <dictionary>.iteritems()
```

Example:

```
for cartoon, animal in my_dict.iteritems():
 print cartoon + ‘ is a ’ + animal + ‘!’

# Prints the following
bugs is a rabbit
elmer is a human
wiley is a coyote
```

```
tom is a cat
jerry is a mouse
```

Let's do the example above, making all the values upper-cased:

```
for name, cartoon in my_dict.iteritems():
  my_dict[name] = cartoon.upper()
```

This syntax is much cleaner and easier to read.

MAP

`map()` is a useful built-in function that takes 2 parameters, a function and and an iterable (lists and dictionaries are iterables). The map function maps each element of the iterable, using the specified function, to a new iterable.

```
def DoubleEven(num):
  if num % 2 == 0:
    num = num * 2
  return num

list1 = [5, 6, 8, 12, 7]
list2 = [1, 8, 23, 90]

new_list1 = map(DoubleEven, list1)
new_list2 = map(DoubleEven, list2)

print new_list1
print new_list2
```

Running the code above will print:

```
[5, 12, 16, 24, 7]
[1, 16, 23, 180]
```

The `map()` function helps to simplify the process of altering the contents of data structures and reduce the number of lines of code written. Without `map()`, the following would have needed to be written like this:

```
for index,num in enumerate(list1):
  if num %2 == 0:
    list1[index] = num * 2

for index,num in enumerate(list2):
  if num %2 == 0:
    list2[index] = num * 2
```

As you can see, the logic for doubling the even values has been repeated twice, once in each for loop. If I wanted to change the logic, I would have to change it in 2 places.

Another key advantage to using `map()` is that you create a function whose name indicates its purpose. In the above example `DoubleEven` is the name of the function and tells the developer and the viewer exactly what is going on.

NESTING

Nesting refers to the process of putting one thing in another. In the the context of data structures, this means putting data instructions in each other. Think of it like putting a bunch of boxes inside a single box. Each sub-box can be a different size and have elements of its own (perhaps even smaller boxes!).

Here is an example of a list of lists:

```
my_list = [['a', 'b', 'c'],
           [1, 2, 3],
           ['d', 'e', 'f']]
```

Each element of `my_list` is a separate list object itself that has full list functionality. For instance:

```
print my_list[1] # [1, 2, 3] - why?
print my_list[1][1] # 2 - why?
```

Now, let's take a look at nesting dictionaries:

```
my_dict = {'alberthwang': {'name': 'Albert Hwang',
                           'eeid': 67739,
                           'title': 'internal tools developer'},
           'satishm': {'name': 'Satish Musurunu',
                       'eeid': 67238,
                       'title': 'tech lead'}}
```

In this example, each element of `my_dict` is a separate dictionary itself, with employee info. Each key is an employee's ldap and each corresponding value is another dictionary that holds the employee's information. You can look up employee information using the ldap from `my_dict` like so:

```
print my_dict['satishm'] # {'name': 'Satish Musurunu', 'eeid':..} - why?
print my_dict['alberthwang']['eeid'] # 67739 - why?
```

IN-CLASS EXERCISES

Please use the remainder of class to practice working with data structures by completing the following exercises.

1) `my_list = range(1,11)`, write a `for` loop to move the entire list up by 2 (i.e. [3,4,5...]). Use `enumerate()`.

2) Using this dictionary:

```
my_dict = {'bugs': 'rabbit',
           'elmer': 'human',
           'wiley': 'coyote',
           'tom': 'cat',
           'jerry': 'mouse'}
```

Write a set of code that will print the following (order of the lines doesn't matter):

```
bugs the rabbit hops
elmer the human walks
wiley the coyote walks
tom the cat schemes
jerry the mouse scampers
```

Use as few lines of code as possible and utilize at least one of the new data structure techniques we learned about this week (i.e. `iteritems()`).

3) Given the string:

```
'1,55,6,89,2|7,29,44,5,8|767,822,999'
```

Use `split()` to create a data structure, `my_data`, such that:

```
print my_data[1][2] # '44'
```

4) Given the string:

```
'alberthwang-90,80,70,50|smadaan-99,80,70,90|satishm-90,90,90,90'
```

Use `split()` to create a data structure, `test_scores`, such that:

```
print test_scores['smadaan'][2] # 70
```

(Hint: Nesting)

5) Given the list:

```
num_list = [5.2, 5.6, 5.1, 5.8, 5.9]
```

Create a rudimentary rounding function to convert `num_list` to this:

```
num_list = [5, 6, 5, 6, 6]
```

Use `map()` to convert the values in `num_list` using your rounding function.

(Hint: This rounding function only has to work with values between 5 and 6)

6) Re-create a solution to HW #3's Part I:

Print the following:

```
the number is 1, that is the best
the number is 2, that's the type of pencil you use in the SAT
the number is 3, that rhymes with 'me'
the number is 4, i want some more
the number is 5, like a high-five!
the number is 6, what a neat trick...
```

This time:

- Do NOT use `while`, `if`, `elif`, or `else`
- Do NOT use a counter variable that gets incremented
- Repeat as little text and code as possible

(Hint: The answer involves data structures!!)

7) **(OPTIONAL - Difficult - Extra Credit)**

Given the string:

```
'ldap:alberthwang,eeid:67739|ldap:meng,eeid:107,building:CL5'
```

Create a data structure from it called `employees`, such that:

```
print employees[1]['building'] # 'CL5'
```

Use as few lines of code as possible. (Hint: Use Split)