**INTRODUCTION TO PROGRAMMING**
**WEEK 4 - DATA STRUCTURES AND FUNCTIONS**

## DATA STRUCTURES

Python data structures are designed for holding and manipulating data with ease. In many ways, these data structures approximate the kinds of mental structures that we make in our minds - lists, queues, dictionaries, etc. Because of this, data structures tend to be very intuitive for most. Also, unlike other languages which are more strongly typed, python data structures are dynamically typed - able to hold many types of data, adjust in size, nest, etc.

### LIST

The list is the most basic python data structure. It is exactly as it sounds, a list of objects that you can manipulate. The syntax for a list is a set of object enclosed in brackets. To create an empty list, just set a variable equal to empty brackets:

```
my_list = []
```

To create a list with some elements in it, just add the elements separated by commas:

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

#### Getting the length of a list

Using the `len` function, you can return the length of a list:

```
print len(my_list) # prints 3
```

#### Accessing lists

Lists can be accessed in the same way that strings are accessed:

| | |
|---|---|
| `my_list[0]` | `'albert'` |
| `my_list[1]` | `'sauwei'` |
| `my_list[0:2]` | `['albert', 'sauwei']` |
| `my_list[1:]` | `['sauwei', 'hwang']` |

Notice here that the objects themselves stay intact even though they are a part of a list. Think of a `list` like a box of chocolates. The chocolates are the elements of the list and they are independent of the box and the other chocolates. Just because a piece of chocolate is in a box does not mean that it's no longer a piece of chocolate. Similarly, if you eat one piece of chocolate, none of the other pieces of affected.

What would this print?

```
my_list[0][3]
```

It prints:

```
'e'
```

Why?

**Modifying a list**

The easiest way to modify a list's content is to just access the list element by its index (numerical place in the list) and use the assignment operator.

```
my_list[0] = 'AL'
print my_list

# ['AL', 'sauwei', 'hwang']
```

Another convenient way to modify a list is the `append()` method. The `append` method allows you to stick an element at the end of a list.

```
my_list.append('smith')
print my_list

# ['albert', 'sauwei', 'hwang', 'smith']
```

Here are some other list methods that will come in handy:

| | |
|---|---|
| `my_list.append('smith')` | `['albert', 'sauwei', 'hwang', 'smith']` |
| `my_list.extend(['granny', 'smith'])` | `['albert', 'sauwei', 'hwang', 'granny', 'smith']` |
| `del my_list[1]` | `['albert', 'hwang']` |
| `my_list.pop(1)` | `['albert', 'hwang']` |
| `my_list.remove('albert')` | `['sauwei', 'hwang']` |
| `my_list.sort()` | `['albert', 'hwang', 'sauwei']` |

***Note on sort()**

The `sort()` method operates on a list **in place**, and has a **`None`** return type. To sort a list using this method, you just have to do this:

```
my_list.sort()
print my_list

# ['albert', 'hwang', 'sauwei']
```

If you want to re-assign a sorted list to a **different** list, use the `sorted()` function instead:

```
new_list = sorted(my_list)
print new_list

# ['albert', 'hwang', 'sauwei']
```

**Introducing....the for loop!**

The `for` loop allows you to iteratively execute a code block over every element of an iterable object. Imagine if you're cleaning out your refrigerator, you might follow a process like this.

- Pick something out of the fridge
- If it's expired, throw it out, otherwise put it back
- Repeat until this is complete for all items in the refrigerator

This would be expressed in python like so:

```
for food_item in refrigerator:
  if food_item.expired == True:
    food_item.ThrowOut()
  else:
    food_item.PutBack()
```

In this example, refrigerator is some sort of entity that holds food objects...sort of like a list?!

**Iterating through a list**

One of the most common things to do on a list object is iterating through the list and doing something to each element - like printing them out or changing them somehow. For this, you should use the `for` loop. The convention for a for loop is:

```
for <loop variable> in <iterable>:
```

A few conventions to note:

- The loop variable is the variable that each element of the iterable will be reassigned to at the beginning of the loop.

- The iterable can be any iterable data structure (list, tuple, dictionary)
- The key parts, “for”, “in” and the colon (“:”) are absolutely key and need to always be included.

Here’s an example:

```
my_list = [‘albert’, ‘sauwei’, ‘hwang’]

for name in my_list:
 print ‘Call me - ’ + name
```

Running the above will print:

```
Call me - albert
Call me - sauwei
Call me - hwang
```

Notice here that the `name` variable is reassigned to each new element going through the loop.

In fact, the variable `name` is not special in anyway, I could have called it anything I wanted. For instance, this would have worked:

```
for thing in my_list:
  print ‘Call me - ’ + thing
```

**Modifying a List by Iterating Through**

Here is another loop, but this time, modifying the list:

```
counter = 0
for name in my_list:
  my_list[counter] = name.upper()
  counter += 1
```

We are looping through the list and capitalizing each element. Now, if we were to print `my_list` it would look like this:

```
print my_list # [‘ALBERT’, ‘SAUWEI’, ‘HWANG’]
```

The `counter` variable is a simple integer that will increment with each iteration. We need this integer to iteratively reference each list element so that we can change it. The `name` variable is just the list element that corresponds to that specific index.

**The Range Function**

One extremely convenient feature of python is the range function. This function takes up to 3

integers as its parameters and **generates a list of integers.**

```
range(<start_int>, <end_int>, <interval>)
```

If the `start_int < end_int`, then the range function generates a list of integers that fulfills these requirements:

```
start_int =< integers < end_int # incrementing at a set interval
```

If the `start_int > end_int`, it generates a list of integers that fulfulls these requirements:

```
start_int >= integers > end_int # decrementing at a set interval.
```

Also important is the function signature. When left empty, range assumes the start_int parameter to be 0 and the interval parameter to be 1:

```
range(12) # same as range(0, 12, 1)
range(1, 10) # same as range(1, 10, 1)
range(2, 12, 2)
```

Some examples will help to clarify this extremely useful function::

```
range(0,10,1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

range(2,10)
[2, 3, 4, 5, 6, 7, 8, 9]

range(5)
[0, 1, 2, 3, 4]

range(10,1)
[] # Why is this?

range(10,1,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]

range(10,2,-2)
[10, 8, 6, 4]
```

This is extremely convenient when you want to do something over a set of numbers, for instance:

```
bottle_numbers = range(99,0,-1)

for num in bottle_numbers:
  print str(num) + ' bottles of beers on the wall.'
```

Like almost anything in python, this can be simplified like this:

```
for num in range(99,0,-1):
  print str(num) + ' bottles of beers on the wall.'
```

**Modifying a list by iterating with Range()**

Given what we know about the range function, we can now change all the elements of a list with it (instead of creating a separate counter variable).

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

print my_list

for index in range(0, len(my_list), 1):
  my_list[index] = my_list[index].upper()

print my_list
```

This will print:

```
['albert', 'sauwei', 'hwang']
['ALBERT', 'SAUWEI', 'HWANG']
```

**Searching through a list**

What if you have a list of groceries and you want to check if `'apples'` is in that list? Or a list of names and you want to check to see if someone is present? Use the **`in`** operator.

```
if 'albert' in my_list:
  print 'good, albert is in the list'
else:
  print 'gosh, where is albert in this list?!'
```

Conversely, to check if an element is absent from a list, use the `not in` operator:

```
if 'albert' not in my_list:
  print 'where is albert?!'
```

**TUPLES**

Tuples are another type of data structure that are very similar to lists. In fact, the only difference is that tuples are *immutable* which means that they cannot be changed. Here is an example:

```
my_list = ['albert', 'sauwei', 'hwang']
my_tuple = ('albert', 'sauwei', 'hwang') # Tuples are created using parentheses

print my_list[2] # prints 'hwang'
print my_tuple[2] # prints 'hwang'

my_list[2] = 'Wong'
my_tuple[2] = 'Wong' # Illegal! Will throw an exception

print my_list[2] # prints 'Wong'
print my_tuple[2] # prints 'hwang'
```

As we can see from the above example, attempting to alter the contents of a tuple is illegal. Once the tuple is created, it cannot be changed.

What if we did this?

```
my_tuple = ('a', 'b', 'c')
print my_tuple # prints ['a', 'b', 'c']
```

What?! Doesn't this violate the rules of immutability?

In fact, it doesn't. The variable `my_tuple`, in and of itself, is not actually a tuple object but a pointer to a tuple object. When you invoke `my_tuple = ('a', 'b', 'c')`, you are actually creating a new tuple (with contents 'a', 'b', and 'c') and pointing `my_tuple` to it.

**What's a tuple good for?**

Tuples are good when you are creating a list that should NOT be changed by the program and you want to enforce it. For instance, you might create a tuple of data representing the EEID numbers of a set of Googlers or the birth years of several people.

Tuples are more commonly used to hold sets of static variables (like the boiling points of various liquids or conversion constants).

We won't be using Tuples much in this class.

## DICTIONARIES

When you open up a dictionary, what do you see? You see words and their definitions. There is a single entity that is a word and then a non-unique definition associated with that word. Python dictionaries function in much the same way. They act like lists but with associated `keys` and `values`.

Think of `keys` like the words in a dictionary and the `values` as the definitions. A dictionary holds pairs of `keys` and `values`. A `key` in a dictionary must be unique and associated with a value. However, a `value` can be associated with many keys.

The syntax for declaring a dictionary is with curly brackets. To create an empty dictionary, just set a variable equal to curly braces:

```
my_dict = {}
```

To create a dictionary with values in it, just create the key value pairs separated by colons and commas:

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

In a dictionary:

- `key` - `value` pairs are separated by comma
- In each pair, `keys` are on the left of the colons (`'bugs', 'elmer', 'wiley', 'tomas', 'jerry'`)
- `values` are on the right of the colons (`'rabbit','human','coyote','cat', 'mouse'`)

Just as in a regular dictionary, the `keys` must be unique, otherwise the program will throw an error. This would be illegal and non-sensical:

```
my_dict = {'bugs': 'rabbit',
           'bugs': 'non-rabbit'}
```

**Accessing a dictionary**

In order to access the values of a dictionary, you want to pass the `key` to the dictionary and it will return the `value`.

```
print my_dict['elmer']

# 'human'
```

**Updating a dictionary**

To modify the values inside a dictionary, the same thing applies - you want to use the assignment operator and send it the `key` value.

```
my_dict['bugs'] = 'bunny'
print my_dict['bugs'] # prints 'bunny'
```

You can also use this technique to add key-value pairs to a dictionary that don't already exist.

For instance, with `my_dict`, I could do the following:

```
my_dict['tweety'] = 'bird'
print my_dict

#

{'tomas': 'cat', 'wiley': 'coyote', 'elmer': 'human', 'bugs': 'rabbit'
, 'jerry': 'mouse', 'tweety': 'bird'}
```

The {tweety: bird} key value pair has now been added to `my_dict`.

**Common Operations**

| | |
|---|---|
| `len(my_dict)` | `5` |
| `my_dict.clear()` | `{}` |
| `del my_dict['bugs']` | `{'elmer': 'human',`<br>`'wiley': 'coyote',`<br>`'tomas': 'cat',`<br>`'jerry': 'mouse'}` |

**Searching through a Dictionary for a Key**

What if you wanted to check if the key `'jerry'` was in `my_dict`? Use the **`in`** operator.

```
if 'jerry' in my_dict:
  print 'Hi Jerry!'
```

This only works for `keys` in a dictionary, not the `values`.

**Iterating through a dictionary**

Again, using the `for-in` loop we do:

```
for cartoon in my_dict:
  print cartoon
```

What do you notice? What is it printing? It's printing the `keys`!

```
bugs
elmer
wiley
tomas
jerry
```

How would you print the values? Combine the technique of accessing the dictionary through the key:

```
for cartoon in my_dict:
  print my_dict[cartoon]
```

This prints:

```
rabbit
human
coyote
cat
mouse
```

**FUNCTIONS**

Functions are defined subroutines that you can custom create in python to perform a set of actions. As you may have noticed, Python has several built-in functions that we have used already in the class (raw_input, range, len, etc.). Here is how a function works. Imagine a Python script for your life. During the day, this set of code may execute:

```
def Yawn(time_of_day):
  yawn_text = 'AHHH...' + time_of_day + '.'
  return yawn_text

if just_woke == True:
  print Yawn('Mornings') # prints 'AHHH...Mornings.'
if is_bored == True:
  print Yawn('Afternoons') # prints 'AHHH...Afternoons.'
```

In this example, `Yawn` is a function and invoking `Yawn()` calls, or executes, that function.

What do you think are the main advantages to writing functions?

**Defining a function**

To define a function, you want to use the `def` statement:

```
def AddFive():
  return 5
```

The key elements here are the `def` statement, the parentheses and the colon at the end.

You can also define a function with **parameters**, or values that the function can read in and do

work on:

```
def AddFive(num):
 num += 5
 return num
```

In the above function, the only **parameter** is `num` which is passed to the function at run time. You can also write functions that don't take parameters. Like so:

```
def SayHi():
  return 'hello!'
```

Here we see the use of the **`return`** statement. This statement tells us the function to stop executing and return the specified value to the function caller. Take this example, an inches converter.

```
def ConvertInches(cm):
 inches = cm/2.54
 return str(inches)

num_inches = ConvertInches(254)
print 'Here is 254cm in inches: ' + num_inches
```

Running the above will print:

```
Here is 254cm in in inches: 100
```

As you see here, just by specifying the function name in the script, I can call it to action on any element, return a value, and execute further on that returned value.

### Encapsulation

Functions are a great example of what computer scientists call "encapsulation". Encapsulation is a concept and a methodology that states that all independently running code should be defined in separate, independent units. When programs execute, they should just put these units together and run them. Let's look at the fake example again at the beginning of the section:

```
def Yawn(time_of_day):
  yawn_text = 'AHHH...' + time_of_day + '.'
  return yawn_text

if just_woke == True:
  print Yawn('Mornings') # prints 'AHHH...Mornings.'
if is_bored == True:
  print Yawn('Afternoons') # prints 'AHHH...Afternoons.'
```

What if I wanted to change the yawn text to say ‘WOOYAAHHH...’ instead of ‘AHHH...’? With the subroutine encapsulated in a function, I only have to change it once (in the definition of the function). If I hadn’t used a function, I would have had to change it under both code blocks (if just_woke and if is_bored).

**Default parameters**

In python, we also have the luxury of defining default parameters. Simply use the assignment operator in the function signature and we have a default value that will be used in the parameter’s place if nothing is specified. For instance:

```
def ConvertInches(cm=10.0):
 inches = cm/2.54
 return str(inches)

num_inches = ConvertInches(20.0)
print ‘here is 20 cm in inches ’ + num_inches

num_inches2 = ConvertInches()
print ‘here is the default conversion ’ + num_inches2
```