Handout 2 – Functions, Lists, For Loops and Tuples

[…]

Functions -- parameters/arguments, "calling" functions, return values, etc.

Please make sure you understand this example:

```
def square(x):
    return x * x;
```

The x in parentheses is the input to my function. We call this a **parameter** or **argument**. We say we **pass** x as a parameter/argument to the function.

We use "return" whenever we want the function to **evaluate** to something. When we call something like sqrt(4), we expect it to evaluate to a number, in this case 2. Similarly, when we call encrypt("password") in the login program, we expect it to evaluate to something, in this case a number, so the function has to return something.

Finally, this won't work:

```
x = sqrt + 4
```

Why? You're trying to add a function to something, you can't do that. This is the difference between simply referring to a function, and **calling** the function. To call a function, you always need to include the parentheses and give it input. It is only when you **call** a function that it gets **evaluated**.

A lot of people make repeated mistakes of not calling their functions and getting errors. Make sure you're calling your functions.

Common mistakes with and questions about lists.

Variables are typed, right? A variable can be a string, an int, etc. Similarly, a variable can be a **list**. The point of a list is to store a dynamic/indefinite number of things. A list is always written as stuff inside **[ ] square brackets**.

```
L1 = ['A', 'B', 'C']
L2 = [ ]
```

We can find the length of a list by doing len(L1)or len(L2) -- the first is 3 and the second is 0. But here's the more important point: just like you can do:

```
x = 10
print x <-- prints 10
x = 5
print x <-- prints 5
```

You can also do the same thing with the individual elements in a list. Why? Because the individual

elements are themselves variables. The list is just a way of tying all the variables together.

We access (read or write) these variables by using **indices** (plural of **index**) -- each element is numbered, **starting from 0**. Remember, the first element is at index 0, second is at index 1, etc. This means that **len(L) is never a valid index**. So a list with 5 elements does not have an element at index 5.

When we refer to an element in list L at index i, we say it's **L sub i**, written as **L[i]**. Since this is a variable, we can do the same thing:

```
L[i] = 10 <-- assuming i is a valid index, i.e. from 0 up to (but not including) len(L)
print L[i] <-- prints 10
L[i] = 5
print L[i] <-- prints 5
```

This is the way we **change** elements in a list. We can also add elements to a list by using **L.append(x)**. This adds the element to the end.

```
L = [] <-- empty list
L.append(10) <-- L is now [10]
L.append(20) <-- L is now [10, 20]
```

Finally, we can delete elements in a list by using the **del** keyword:

```
L = ['A', 'B', 'C', 'D']
L[2] <-- evaluates to 'C'
del L[2]
L[2] <-- now evaluates to 'D'
```

Remember that because the length of the list has changed, the indices have also changed -- everything to the right of the deleted element is shifted over to the left. So be careful -- L[3] was valid, but no longer is.

Oh, one last thing -- we can check whether an element is in a list really simply with the **in** keyword. Something like 'A' in ['A', 'B'] returns True, but 'C' in ['A', 'B'] returns False. This is NOT to be confused with the **in** that's part of **for ... in ...** as we'll see below.

If there is anything about lists above you don't understand, please make sure to read the textbook (chapter 9) and look over the labs.

For statements -- when to use range, when not to, etc.

I think everyone understands while loops for the most part:

```
i = 1
while i <= 10:
    print i,
    i = i + 1
```

This will print "1 2 3 4 5 6 7 8 9 10", easy. Now I'm going to make a subtle change:

```
i = 0
while i < 10:
    print i,
```

```
    i = i + 1
```

This will now print "0 1 2 3 4 5 6 7 8 9". Make sure you understand the difference and are careful to look for these things.

Okay, so now for loops are another type of loop, and here's how they work. A lot of people misunderstand this:

for [variable] in [something that evaluates to a LIST!!!]:
**   [statements]**

Please -- whenever you're confused with for statements, refer to this above template!! You must ALWAYS have something that evaluates to a list. This is NOT legal:

```
for i in len(L):
    ...
```

Why? What is len(L)? It's a number. What's it's type? int. What type does it need to be? list.

Okay. Given that syntax, the for loop will execute **[statements]** some number of times, call it N. What is N? N is equal to the **length of the list**. And on each **iteration** ("iteration" means step of the loop), it will set **[variable]** to be an element in the list. The first iteration, **[variable]** will be the first element in the list. The second iteration, the second element. And so on.

So when I say:

```
L = ['W', 'X', 'Y', 'Z']
for holymoly in L:
    print holymoly,
```

What should it print? It should print "W X Y Z". It will NOT print "0 1 2 3". DON'T ASSUME that just because it's a for loop, it must be numbers, or it must be index values if we're doing a list. Stick to the above definition!!

Now how about **range**? This is extremely important -- many of you are completely missing this: **range is a function that returns a list**. What does that mean? That means whenever you put **range(...)** , that expression will **evaluate** to a **list**. Understand?

range(5) evaluates to a **list** that looks like this: [0, 1, 2, 3, 4]
range(0, 5) evaluates to the same list.
range(1, 5) evaluates to this list: [1, 2, 3, 4]
range(1, 10, 3) evaluates to this list: [1, 4, 7]
range(1, 10, 4) evaluates to this list: [1, 5, 9]

If L = ["A", "B", "C", "D", "E"], then:
range(len(L)) evaluates to this list: [0, 1, 2, 3, 4]
range(0, len(L)) evaluates to the same list.
range(1, len(L)) evaluates to this list: [1, 2, 3, 4]
and so on.

Make sure you understand these examples . So what will this do?

```
for i in range(1, 7):
    print i,
```

It will print "1 2 3 4 5 6". Now, a lot of you make this mistake over and over:

```
for i in range(1, 7):
    print i,
    i = i + 1
```

Why is the "i = i + 1" unnecessary? Because every time the **[statements]** (see template above) are finished executing, the computer assigns **[variable]** to the **next element in the LIST**. Let's look at this example:

First, i becomes 1, because it's the **first element** in **range(1, 7)**.
We print i (1), then we set i to i+1 (2).
Next, i becomes 2, because it's the **second element** in **range(1, 7)**.

Do you see why the "i = i + 1" is unnecessary? Don't write it in your programs!! It might not cause a problem in this snippet, but it can easily cause bugs other times. Here's an example:

```
NAMES = ["Alice", "Bob", "Cathy", "Doug"]

for i in range(len(NAMES )):
   print "Hello", NAMES [i]
   i = i + 1

print "We're out of names!"
print "The last name we saw was", NAMES[i]
```

That mistake now crashes our program at the end. **Make sure you see and understand why.**

So, if you want to iterate over **numbers**, then use **range**. If you want to iterate over the elements in a list directly, **don't** use range. Compare:

for i in range(len(L)) <-- i will now be the indexes of L, i.e. from 0 up to (but not including) len(L)
for i in L <-- i will now instead be L[0], then L[1], then... all the way to the last element

Make sure you understand this!!

Why use tuples? How are strings like tuples?

Tuples are **immutable** variations of lists. By **immutable**, we mean that they cannot be changed:

- can't be appended to
- can't change elements, i.e. re-assign their values
- can't delete elements

Once a tuple is created, it's final. So, it almost never makes sense to create an empty tuple. We create empty lists often when we don't know how long the list is going to be, so we create an empty list and keep appending to it for as long as we need to. But a tuple, if you create an empty tuple, it's useless.

Tuples are things in **( ) parentheses**. **However, we still index them with [ ] square brackets.**

a = ('A', 'B', 'C')
a[1] <-- B

a(1) <-- error, Python thinks a is a function, and the function a doesn't exist. You'll get an error that says something like **"a is not callable"**. Why? Because to be **"callable"** means you're a function -- we can call you by putting parentheses.

So why do we use tuples? The answers aren't going to satisfy you guys yet... it's something you'll come to understand as you study more advanced topics, but it has to do with safety and security.

Safety: in essence, everyone makes mistakes when they write programs, because we're all human. These mistakes are bugs. When a program freezes or crashes, that's a bug, caused by some mistake in some code somewhere. Bugs are inevitable, but we can use good programming practice to prevent as many of them as we can.

Now, how do tuples help? Well, let's say we're working with points. We're repeatedly calling functions, sending over points. Our points are 2D, they are (x,y) coordinates. All points should thus always have exactly 2 variables. By using a tuple instead of a list, the computer enforces this. If we used a list, we may have a valid point that becomes invalid when we accidentally add an element, or remove an element. By using a tuple, the computer will give us an error right away, letting us find exactly where we made the mistake, rather than letting it go unnoticed until it crashes the system later, when we have no idea where we made the mistake.

Furthermore, imagine that the points represent something. Let's say they represent the shape of a wing your company has designed that's perfectly more aerodynamic. Now, you write a program that experiments around with these points to try to come up with something better. If the points were mutable, you might accidentally make the mistake of changing the points themselves.... let's say the wing tip moves up by 2 inches and over by 3 inches. etc. If they're tuples, the computer enforces that you don't change the actual points themselves -- you can copy the points and experiment with the copies, but you can't change the original points.

This is even more important when you realize that almost every piece of code interacts with someone who didn't write that code. If I write a function that returns a list of 10 numbers that i use for my encryption algorithm, I don't want the person who calls that function to accidentally delete one of those numbers or change one of them. So I send it as a tuple -- this guarantees that they can't change it. They can again see it, copy them, whatever, but they can't change MY numbers.

Okay, so **strings as tuples**. Careful -- I didn't mean that strings **ARE** actually tuples, I said they are very similar to tuples.

What are tuples? They are an immutable collection of elements. Can't be deleted, added to, or changed. Strings are the same.

```
a = "hello"
len(a) <-- evaluates to 5
a[4] <-- evaluates to "o"
a.append("w") <-- error, no append operation
a[4] = "n" <-- error, assignment not supported
del a[4] <-- error, deletion not supported
```

I'm not sure how else to explain this. Take a look at the pig latin solution to see exactly how to work with strings when you know they're indexable but immutable.

[…]