Lab 3c: Python III - Tuples, Lists, and Measuring Time

0.1 Getting Setup

Download the sample files, `Lab3c.py`, from the web site. Setup your robot, and run the sample file. It should do nothing but print `all done`. You are now ready to begin.

1 Tuples

A *tuple* is a simple way to combine multiple pieces of data into a convenient package. We will only really use them for one purpose: to return multiple values from a function.

To make a tuple, surround any number of variables with parenthesis, like so:

```
a = 4
b = 5
c = 6
q = (a, b, c)
```

When you print a tuple, it prints with the parenthesis, to show you that there this data is stuck together. The variable names do not get stored in the tuple, jus the values. To get the values out of the tuple, use this syntax, with new variable names to assign the values to:

```
(q1, q2, q3) = q
q1
q2
q3
```

Tuples are commonly used to return multiple values from a function, like so:

```
def foo(x, y):
    t1 = x + y
    t2 = x - y
    t = (t1, t2)
    return t

def use_foo(x, y):
    val = foo(x, y)
    print val

    (val1, val2) = val
    print val1, val2

use_foo(2, 1)
```

2 Lists

Tuples are useful, but limited. You can't modify them after they are created. A *List* lets you add and remove elements. Let's make some variables and put them in a list.

```
a = 1
b = 2
c = 3

print a, b, c

l = [a, b, c]

print l
```

We can access elements at specific locations in our list using square braces syntax:

```
print l[0]
print l[1]
```

We use the `append` *method* to add variables to the list. A method is like a function, but it it called by using *dot notation* on a particular piece of data that you want to work with. This is part of *object-oriented programming*, and is beyond the scope of this course, but you can still use the syntax to operate on lists.

```
this = True
that = False

l.append(this)
l.append(that)

print l
```

Note that you can have variables of multiple types in a list. Python doesn't care, but usually lists are of a single type.

The `for` loop is a compact way to iterate over a list:

```
l = [1, 23, 5, 13, 12]
sum = 0
for x in l:
    print x
    sum = sum + x
print sum
```

Note the use of the `in` keyword. This tells Python to pull elements from the list, one at a time, in order, and store them in the temporary variable `x`. Recall for loops from last time? What is actually

happening is that the `range()` function is making a list on the fly for the for loop to iterate over. This temporary list can't be printed, but still works like a list for the for loop.[1]

There are many other list methods, we will introduce them as we go. The full reference is on the web at:

`http://www.clear.rice.edu/engi128/Resources/library_reference.html`

3 Measuring Time

We have showed you the `sleep()` function, which makes python wait for a specified number of milliseconds before continuing. This was useful, but a bit wasteful, because the computer can't do anything else while it is waiting. Another way to measure time is to use the `time()` function. This function returns the current time on the robot, in milliseconds.

```
import sys
sys.time()

#wait for a bit
sys.time()
```

We can use the ability to measure time to rewrite the `move_forward()` function from your homework (answers turned in for your homework should only use the `sleep)(` function:

```
# moves forward for the argument time
# arguments: time
# return: nothing
def move_forward(time):
    time_start = sys.time()
    time_end = time_start + time
    while sys.time() < time_end:
        rone.motor_set_pwm('l', 65)
        rone.motor_set_pwm('r', 65)
        print sys.time()
        ## wait a bit before checking again
        sys.sleep(10)
    rone.motor_set_pwm('l', 0)
    rone.motor_set_pwm('r', 0)

move_forward(1000)
```

[1] Note to advanced Python users, `range` actually returns a `xrange` on the robots.