1 Administrative Topics

- Return Quiz1.
- Showcase some Proj 2 wiki pages.
- There is no quiz and no homework this week.
- Note: Monday was a snow day, so this material may be covered on Friday.

2 Lists

We have talked about very simple data types (strings, ints, floats, booleans) and very complex ones (modules and functions). Today, we introduce lists, which are somewhere in the middle. Lists in Python are lists of values. For example, here is a list:

\[ [0, 1, 2, 3] \]

It contains the numbers 0, 1, 2, and 3.

In fact this is the return value from a call to the built-in function `range(4)`.

```python
my_list = range(4)
print(my_list)
```
outputs
[0, 1, 2, 3]

We can create our own lists:
```
taylor_ages = [2, 4.5, 36, 36]
```

One of the major uses of lists is to tell for loops what values to give their loop control variables.

## 3 For Loops

The syntax of a for loop is
```
for item in list:
    # block of code “in the loop”
```

- `item` is the loop control variable. It is just a variable. It can have any name.
- `list` is a list of values.

Python executes a for loop by setting the loop control variable to the first item in the list, then executing the block of code. Then, Python sets the loop control variable to the second item in the list, then executes the block of code. This process is repeated until there are no more elements in the list.

How does execution of the for loop affect the memory? First, Python determines what the list is and puts it into memory in a hidden place. We will leave this out of the symbol table. Then python adds the loop control variable to the symbol table (or just updates it if it is already there). It sets its value to the first item in `list`. Then, Python executes the block of code in the loop. This may affect the symbol table. The process is repeated for each item in the loop. Once the loop has finished execution, the loop control variable remains in the table with the last value from the list.

Let’s return to our example of the Taylor family’s ages. We can loop through that list and print each age individually.

Notice that the loop control variable does not need to be named `i`. In fact, there are times when that isn’t an appropriate name. Here, for instance, it
should be age because an item in a list of ages is an age. In general I use i for a counter loop control variable (e.g. one that is 0, 1, 2, 3, etc.). I use x and y for positions. I try to choose variable names that are appropriate to the context.

We can nest for loops. For example

```python
for i in range(3):
    for j in range(2):
        print(i, j)
```

and the output is

```
0 0
0 1
1 0
1 1
2 0
2 1
```

The body of the outer loop executes 3 times because there are 3 items in the list returned by `range(3)`. That means the inner loop is executed 3 times. The body of the inner loop (i.e. the print statement) executes twice each time the inner loop is executed. Therefore, the body of the inner loop (i.e. the print statement) is executed 6 times. Basically, the inner-most code executes once for each combination of inner and outer values. And its does so in the above order.

### 3.1 Drawing blocks with for loops

To draw a row of blocks, we could use a for loop.

```python
def rowOfBlocks(x, y, scale):
    """draw a row of blocks with the bottom lefthand corner at (x,y)
    If scale is 1, then each block will be 200x200.""
    edge = 200
    for i in range(4):
        block(x+i*edge*scale, y, edge*scale)
```

We use a for loop to draw each of the four blocks. There are a few things to notice in the code
• We use scale to scale the edge length. The “natural” size for a block is 200x200.

• The “loop control variable” $i$ has the value 0 the first time through the loop, 1 the second time through, 2 the third time through, and 3 the fourth time through. So we take advantage of that to determine the x-position of the $i$th block.

• We scale the offset in the x position, but not the x-position itself $(x + i\times\text{edge}\times\text{scale})$.

3.2 Looping over lists with indices or values

We can access elements of the list using an index. The index of the first element is 0. Suppose we have

```python
nums = [10, 2, 5]
print nums[0]
```

Then 10 is printed.

We can loop through list of numbers directly using this syntax

```python
nums = [10, 2, 5]
for num in nums:
    print num
```

It prints

10
2
5

We can also loop through the list using indices. To do so, we use the `range` function to generate a list of indices, then loop over the values of that list. We use those values to index into the nums list

```python
nums = [10, 2, 5]
for i in range(len(nums)):
    print nums[i]
```

Notice that the `range` function takes the length of the `nums` list as input. It outputs `[0,1,2]`. Then the value of $i$ is 0 the first time through the loop,
so \text{nums}[0], which is 10 is printed. Then the value of \text{i} is 1 the second time through the loop, so \text{nums}[1], which is 2 is printed. Then the value of \text{i} is 2 the first time through the loop, so \text{nums}[2], which is 5 is printed.

These two loops produce identical outputs. We will talk more about which one to choose later one. What I want to point out now is that we use the name \text{i} for the loop control variable if it is used as an index or a simple counter (as above with the row of blocks). We use a name that reflects the content of the list. So when we are looping through a list of numbers, it makes sense to call it \text{num}. In class, we talked about a list of colors and called the loop control variable \text{color}.

4 Writing a Function with a Conditional

Let’s put it altogether in a function. Here it is:

```python
# weatherF : prints a string describing the weather
# based on the temperature given as input
# input:
# f : the temperature in degrees fahrenheit
#    (<int> or <float>)
# output:
# a description of the weather (<str>)
def weatherF(temp):
    if temp > 70:
        print 'hot'
    else:
        print 'not hot'

#main
weatherF(81)
```

This introduces a new type of statement – a conditional, or “if” statement. We use the keyword if followed by a Boolean or an expression that evaluates to a Boolean. If that expression is true, we evaluated the statements “in” the first nested block (e.g. \text{weather = } 'hot'). In other words, we take the first “branch”. If that expression is not true, we evaluate the code in the “else” block (taking the second “branch”).

The if statement can have additional conditions and blocks. We can use the
"if ... else if ... else if ... else" construction. We do this by using the “elif” keyword (a contraction for “else if”). For example, we can add additional conditions to test more values of the temperature (see code example below). Only one block is executed and that is the first block associated with a true condition. For example, if the temperature is 65 degrees F, then the weather will be warm.

```python
# weatherF : prints a string describing the weather 
# based on the temperature given as input
# input
# ftemp : the temperature in degrees fahrenheit
#       (<int> or <float>)
# output:
#   a description of the weather (<str>)
def weatherF( ftemp ):
    if ftemp >= 70:
        print 'hot'
    elif ftemp >= 55:
        print 'warm'
    else:
        print 'cold'

#main
weatherF(81)
```

Note that the elif expression is not `elif temp > 55 and temp <= 70`. It would be entirely correct if it were. But it isn’t necessary to do the second comparison. Why not? Because if-statement expression statements are evaluated in order. If the temp were greater than 70, then the first branch would have been taken. And the elif expression wouldn’t be evaluated. So, if we get to the elif then we already know that the temperature is less than or equal to 70.

### 4.1 Stepping through the code

We didn’t step through the symbol tables in class, but here are notes about it if you are interested.

Let’s draw the symbol table for weatherF when it is called with a temperature of 81 degrees F.

I run it, and at the first line of top-level (main) code, the symbol table looks
like this:

```
main

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>weatherF</td>
<td>→ &lt;function&gt;</td>
</tr>
</tbody>
</table>
```

When the line `weatherF(81)` is executed, a new symbol table appears:

```
main

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>weatherF</td>
<td>→ &lt;function&gt;</td>
</tr>
</tbody>
</table>

weatherF(81)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>81&lt;int&gt;</td>
</tr>
</tbody>
</table>
```

The first line executed in `weatherF` is `if temp > 75:`. The condition evaluates to `True`, so the first branch is taken. This means the next line executed is `print 'hot'`. The string 'hot' is printed to the Terminal. Then the function is finished executing and the table goes away.

Finally, the `weatherF` symbol table is erased, and the assignment statement (in `main`) is completed:

```
main

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>weatherF</td>
<td>→ &lt;function&gt;</td>
</tr>
</tbody>
</table>

weatherF(81)
```

Then the main code is done and the table goes away.