1 Administrative Topics

- Stephanie showcases projects from Proj 4.

2 Lists

Let’s compare lists to strings:

A string is an immutable sequence of characters.

A list is a mutable sequence of values of any type (including lists!).

2.1 Creating Lists

We have seen that the range function returns a list. We can also create a list explicitly, like this:

\[
\text{myList} = [0,1,2,3]
\]

2.2 List Operations

The same operators that applied to strings apply to lists, with additional operators which mutate the list (basically, we can have a slice operation on the left hand side of an assignment statement).
There are several operators that shown in Table 1.

Table 1: List Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in s</code></td>
<td>True if an item of s is equal to x, else False</td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td>False if an item of s is equal to x, else True</td>
</tr>
<tr>
<td><code>s + t</code></td>
<td>the concatenation of s and t</td>
</tr>
<tr>
<td><code>s * n</code></td>
<td>n shallow copies of s concatenated</td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td>i'th item of s, origin 0</td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>slice of s from i to j (but not including the jth item)</td>
</tr>
<tr>
<td><code>s[i:j:k]</code></td>
<td>slice of s from i to j with step k (but not including the jth item)</td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>length of s</td>
</tr>
<tr>
<td><code>s[i] = x</code></td>
<td>item i of s is replaced by x</td>
</tr>
<tr>
<td><code>s[i:j] = t</code></td>
<td>slice of s from i to j is replaced by the contents of the iterable t</td>
</tr>
<tr>
<td><code>s[i:j:k] = t</code></td>
<td>the elements of s[i:j:k] are replaced by those of t</td>
</tr>
</tbody>
</table>

When indexing into a list, it is acceptable to use negative integers. -1 indicates the last item in the last. -2 indicates the penultimate item in the list, etc.

When using non-negative integers to index, remember that the first item is at index 0, the second item is at index 1, etc.

Here are some examples illustrating the operations (with the command typed at the Python prompt and the result that is displayed) assuming the following line has been executed:

```python
georgeList = ['George', 7, 47, 
               ['Where the Mountain Meets the Moon',
                'The Bluejay Pirate',
                'The Magic Treehouse books']]```

- `georgeList[0] ⇒ 'George'`
- `georgeList[3][-1] ⇒ 'The Magic Treehouse books'`
Note the “double indexing” in the last example above. First we access the book list using `georgeList[3]`, then we index within the resultant sublist with the `[−1]` statement.

Note that we could repeat everything with value rather than the symbol and we would get the same results. This is true because the operations are performed on the value itself – symbols just let us get to the values.

### 2.3 Be Careful with Sublists

Remember that lists are objects and that “copying” a list merely results in the copy of the reference (arrow) to the list object.

If we were to execute the line

```python
booklist = georgeList[3]
```

Then, `booklist` would point to the exact same list as `georgeList[3]`. In other words, if I were to change an item of `booklist`, `georgeList` would be able to “see” that change.

```python
booklist[0] = "Charlotte’s Web"
print georgeList
```

results in the output:

```python
```

Likewise, using an operator like this:

```python
g12 = list*2
```

(which doubles the list and puts it into a variable named `g12`) performs what is called a shallow copy, meaning sublists themselves aren’t copied – just their references (arrows) are copied. In other words, both book sublists in `g12` will refer to the same list. Beyond that, they will both refer to the list referenced by `booklist` and `georgeList[3]`.

The moral of the story is that when you are writing code with lists of lists, make sure....
2.4 Lists Methods

One of the most useful list methods is append, which appends an item to the end of a list. Its syntax, along with that of several other list methods is in Table 2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.append(x)</td>
<td>append the item x to the end of list s</td>
</tr>
<tr>
<td>s.extend(t)</td>
<td>append each item of list t to the end of list s</td>
</tr>
</tbody>
</table>

For a complete list of list methods, see http://docs.python.org/library/stdtypes.html#mutable-sequence-types

2.5 Range function

The built-in range function produces lists of integers.

You can call it with one argument and it will return a list of integers, starting at zero, ending at 1 before that number, e.g

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

You can call it with two arguments. The first is the “start” argument. The second is the “stop before” argument.

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

You can call it with three arguments. The first is the “start” argument. The second is the “stop before” argument. The third is the “step” argument. It
controls the step size between ints.

```python
>>> range(1, 4, 2)
[1, 3]
>>> range(1, 5, 2)
[1, 3]
>>> range(1, 6, 2)
[1, 3, 5]
```

## 2.6 Looping through lists

You can loop through lists directly, or using an index.

To loop through a list directly (to loop through its elements, you just put the list after the for loops in):

```python
nums = [4, 1, 5]
for num in nums:
    print num
```

To loop through a list using indices, use the `len` function to find out how long the list is, then the `range` function to create a list of indices, starting at 0, ending at one less than the length of the list.

```python
nums = [4, 1, 5]
for i in range(len(nums)):
    print nums[i]
```

A note about names. Use a name like `i` for the loop condition variable if it is an index. Use a name that reflects the content if you are looping through it directly (e.g. `num` for a list of numbers).