CSC 108H: Introduction to Computer Programming

Summer 2012

Marek Janicki
Exercise 2 is posted.
  - Due one week from today.

The first assignment will be posted by Monday.
  - Will be due Tuesday after the midterm.
  - Should be started before the midterm.

Help Centre is still open.
  - BA 2270.
String Review

- Strings are a new type we use to represent text.
  - Denoted by ' or “ or ''. 
  - Can use escape characters to put in special characters into strings.
  - Other types can be inserted into a string using string formatting.
- len, ord and char are useful functions.
- .strip, .replace, .lower, .upper, .count are useful methods.
String Review Questions

- Write expression to:
- Determine if 'x' is in string s.
- Remove all instances of 'b' from a string.

- Write a function that takes a lower case letter, and returns the corresponding number assuming a=1, b=2,...

Remove leading and trailing instances of 'b' from a string.
String Review Questions

- Write expression to:
- Determine if 'x' is in string s.
  `'x' in s`
- Remove all instances of 'b' from a string.
  `s.replace('x','')`

Remove leading and trailing instances of 'b' from a string.
  `s.strip('b')`

- Write a function that takes a lower case letter, and returns the corresponding number assuming a=1, b=2, ...
  ```python
def foo(x):
    y = ord(x)
    y -= ord(a)
    y += 1
    return y
  ```
Modules Review

- A module is a single file that contains python code.
- This code can be used in a program that's in the same directory by using `import` or `from module_name import *`
- All of the code in a module is executed the first time it is imported.
- To access imported functions one used `module_name.function_name()`
- Each module has a `__name__`.
  - This is either the filename if the module has been imported or `'__main__'` if the file is being run.
Module Review

- Assume we have a module named foo, that contains a function f, and a variable x.
- How can we get a block to execute only if foo is imported.

- If we import foo without using from, how can we call f or get the value of x?
Module Review

• Assume we have a module named foo, that contains a function f, and a variable x.

• How can we get a block to execute only if foo is imported.

```python
if __name__ == 'foo':
    block
```

• If we import foo without using from, how can we call f or get the value of x?

```python
foo.f, foo.x
```
Lists

- So far, every name we've seen has referred to a single object.
  - Variables names refer to a single int/bool/str/etc.
  - Function names refer to a single function.
- This is not always convenient.
  - Think of keep records for a club.
  - It might be useful to have one way to easily store the names of all the members.
- Can use a list.
Lists

• Lists are assigned with:

\[
\text{list\textunderscore name} = [\text{list\_elt0, list\_elt1, ..., list\_eltn}]
\]

• To retrieve a list element indexed by i one does:

\[
\text{list\textunderscore name}[i]
\]

• So the following are equivalent:

\[
\text{eg\_list} = [15] \quad \text{foo}(15) \\
\text{foo}(\text{eg\_list}[0])
\]
Lists

- Lists are assigned with:
  \[
  \text{list}\_\text{name} = [\text{list}\_\text{elt}0, \text{list}\_\text{elt}1, \ldots, \text{list}\_\text{elt}\text{n}]
  \]

- To retrieve a list element indexed by i one does:
  \[
  \text{list}\_\text{name}[i]
  \]

- So the following are equivalent:
  \[
  \text{eg}\_\text{list} = [15] \quad \text{foo}(15)
  \]
  \[
  \text{foo}(\text{eg}\_\text{list}[0])
  \]
Lists

- Empty lists are allowed: `[]`.
- `list_name[-i]` returns the ith element from the back.
  - Note the difference between `l[0]` and `l[-1]`.
- Lists are heterogeneous:
  - That is, the elements in a list need not be the same type, can have ints and strings.
  - Can even have lists themselves.
Lists

- To get to the i-th element of a list we use:
  \[ \text{list\_name}[i-1] \]
- We use i-1 because lists are indexed from 0.
- This means to refer to the elements of a 4 element list named list\_name we use
  \[ \text{list\_name}[0], \text{list\_name}[1], \text{list\_name}[2], \text{list\_name}[3] \]
List Question.

- What is printed?

```python
eg_list = [0, 'sgeg', True, 12, 'gg']
print neg_list[0]
print neg_list[-0]
print neg_list[-2] + neg_list[-5]
print neg_list[1] + neg_list[-1]
print neg_list[2]
```
List Question.

- What is printed?

```python
eg_list = [0, 'sgeg', True, 12, 'gg']
print eg_list[0]  # 0
print eg_list[-0]  # 0
print eg_list[-2] + eg_list[-5]  # 12
print eg_list[1] + eg_list[-1]  # sgeggg
print eg_list[2]  # True
```
Lists and the memory model.

\[\text{eg\_list} = [0, 1, \text{True}]\]
Lists and the memory model.

\[
egg\_list = [0, 1, \text{True}]\]
Lists and the memory model.

eg_list = [0, 1, True]
Changing a List

- A list is like a whole bunch of variables.
  - We've seen we can change the value of variables with assignment statements.
  - We can change the value of list elements with assignment statements as well.
- We just put the element on the left and the expression on the right:
  \[
  \text{list}\_\text{name}[i] = \text{expression}
  \]
- This assigned the value of the expression to \text{list}\_\text{name}[i].
Immutable objects

- Ints, floats, strings and booleans don't change.

- If we need to change the value of a variable that refers to one of these types, we need to create a new instance of the type in memory.

- That is, instead of making an old int into a new one, we make a new int, and throw the old one away.
Mutability

- If we only want to change one element of a list, then it seems a waste to have to create all of the types that it points to again, even though only one of them has changed.
- So this isn't done. Instead we can change the individual elements of a list.
- Note that since we view these as memory locations, this means that we change the location in memory that the list points to.
Lists and the memory model.

\[
\text{eg\_list} = [0, 1, \text{True}] \\
\text{eg\_list}[0] = 10
\]
Lists and the memory model.

```
eg_list = [0, 1, True]

eg_list[0] = 10
```

```
Global

eg_list: 0x1
```

```

<table>
<thead>
<tr>
<th>int</th>
<th>0x5</th>
<th>int</th>
<th>0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

| bool | 0x8 | True |

<table>
<thead>
<tr>
<th>list</th>
<th>0x1</th>
<th>0x5</th>
<th>0x10</th>
<th>0x8</th>
</tr>
</thead>
</table>
Lists and the memory model.

\[
\text{eg\_list} = [0, 1, \text{True}]
\]

\[
\text{eg\_list}[0] = 10
\]
Lists and the memory model.

\[ \text{eg\_list} = [0, 1, \text{True}] \]
\[ \text{eg\_list}[0] = 10 \]
Lists and the memory model.

\[
\text{eg\_list} = [0,1,\text{True}]
\]

\[
\text{eg\_list}[0] = 10
\]
Aliasings

• Consider:

\[
x = 10 \\
y = x \\
x = 5
\]

\[
\text{print } x, y
\]

• We know this will print 5 10 to the screen, because ints are immutable.
Aliasing

- Let `eg_list` be an already initialised list and consider:

```python
x = eg_list
y = x
x[0] = 15
print y[0]
```

- Lists are mutable, so this will print 15.
What gets printed?

```python
l = [0,1,2]
print l
l[0] = 10
print l[0]
print l
```

```python
l = [0,1,2]
print l
l = 10
print l
print l[0]
```

```python
l = [0,1,2]
print l
l = 10
print l
print l[0]
print l
```
What gets printed?

```python
l = [0,1,2]
print l
l[0] = 10
print l[0]
print l
[l, 1, 2]
10
[10, 1, 2]
```

```python
l = [0,1,2]
print l
l = 10
print l
print l[0]
[l, 1, 2]
10
Crash
```

```python
l = [0,1,2]
print l
l = 10
print l
print l[0]
[l, 1, 2]
10
Crash
```

```python
l = [0,1,2]
print l
l = 10
print l
print l[0]
[l, 1, 2]
Crash
```
Aliasing and functions.

- When one calls a function, one is effectively beginning with a bunch of assignment statements.
  - That is, the parameters are assigned to the local variables.
- But with mutable objects, these assignment statements mean that the local variable refers to a mutable object that it can change.
- This is why functions can change mutable objects, but not immutable ones.
Break, the first.
What gets printed?

```python
def foo(l):
    l[0]=10
x = [15]
print x
foo(x)
print x
```
What gets printed?

def foo(l):
    l[0] = 10
x = [15]
print x
foo(x)
print x

[15]  [15]
[10]  [10]
Why was x not empty?

def foo(l):
    l[0]=10
    l = []
x = [15]

print x
foo(x)
print x

Global
x: 0x1
Why was x not empty?

def foo(l):
    l[0] = 10
    l = []
x = [15]

print x
foo(x)
print x

<table>
<thead>
<tr>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>l: 0x1</td>
</tr>
<tr>
<td>Global</td>
</tr>
<tr>
<td>x: 0x1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
</tr>
<tr>
<td>0x5</td>
</tr>
</tbody>
</table>
def foo(l):
    l[0] = 10
    l = []
x = [15]
print x
foo(x)
print x

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>0x17</td>
</tr>
<tr>
<td>l: 0x1</td>
<td>int</td>
</tr>
<tr>
<td>Global</td>
<td>10</td>
</tr>
<tr>
<td>x: 0x1</td>
<td>list</td>
</tr>
</tbody>
</table>

June 7 2012
Why was x not empty?

def foo(l):
    l[0]=10
    l = []
x = [15]
print x
foo(x)
print x

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l: 0x33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x: 0x1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

June 7 2012
Repetition

- Often times in programs we want to do the same thing over and over again.
- For example, we may want to add every element of a list to some string.
- Or we may want to execute a block of code until some condition is true.
- Or we may want to change every element of a list.
Loops

- Python has two types of loops.
  - The `for` loop.
    - This is a bit simpler.
    - This requires an object to loop over.
    - Some code is executed once for every element in the object.
  - The `while` loop.
    - Some code is executed so long as a certain condition is true.
For Loops with Lists

• syntax:
  
  for item in eg_list:
      block

• This is equivalent to:

  item = eg_list[0]
  block
  item = eg_list[1]
  block
  ...

For Loops with Strings

• `eg_str[i]` evaluates to the i-1\textsuperscript{st} character of `eg_str`.

• **syntax:**

  ```python
  for item in eg_str:
      block
  ```

• This is equivalent to:

  ```python
  item = eg_str[0]
  block
  item = eg_str[1]
  block
  ...
  ```
A useful Loop Template

- Often times we get something from every element of a list and use this to create a single value.
- Like the number of times some condition is true.
- Or the average of the elements of the list.
A useful Loop Template

- In this case we often use an accumulator_variable that accrues information each time the loop happens.

- This often looks like

  ```python
  accum_var = 0 #maybe [] or '').
  for elt in list_name:
      block #This will modify accum_var
  #accum_var should hold the right value here.
  ```
A useful Loop Template

- The average of the number of elements in the list. (\texttt{len(list\_name)} is length of a list)

\begin{verbatim}
accum\_var = 0 #maybe [] or ''.
for elt in list\_name:
    block #This will modify
accum\_var

#accum\_var should hold the right
#value here.
\end{verbatim}
A useful Loop Template

- The average of the number of elements in the list. \((\text{len}(\text{list\_name}) \text{ is length of a list})\)

```python
accum_var = 0
for elt in list_name:
    block #This will modify accum_var
    accum_var
#accum_var should hold the right #value here.
```
A useful Loop Template

- The average of the number of elements in the list. (len(list_name) is length of a list)

```python
accum_var = 0
for elt in list_name:
    accum_var += elt
#accum_var should hold the right value here.
```
A useful Loop Template

- The average of the number of elements in the list. (\texttt{len(list\_name)} is length of a list)

\begin{verbatim}
accum\_var = 0
for elt in list\_name:
    accum\_var += elt
accum\_var = accum\_var/len(list\_name)
\end{verbatim}
Write loops to

- Return the sum of the squares of all the list elements
- Return the number of elements divisible by 7.
Write loops to

- Return the sum of the squares of all the list elements
  
x = 0
  for i in l:
    x += i * i

- Return the number of elements divisible by 7.
  
x = 0
  for i in l:
    if i % 7 == 0:
      x+=1
For Loops with Lists

```python
item = eg_list[0]
block
item = eg_list[1]
block
...
```

- Note that even if the block changes the value of `item` the value of `eg_list[i]` may not change.
- Depends on whether `eg_list[i]` is mutable.
For Loops with Lists

- To guarantee our ability to change eg_list[i] we need the block to have eg_list[item] instead of item, and item to contain the indices.

```python
item = 0
block
item = 1
block
...
```
Looping over Lists

- To do that, we use the `range()` function.
  - `range(i)` returns an ordered list of ints ranging from 0 to i-1.
  - `range(i, j)` returns an ordered list of ints ranging from i to j-1 inclusive.
  - `range(i, j, k)` returns a list of ints ranging from i to j-1 with a step of at least k between ints.
- So `range(i, k) == range(i, k, 1)`
- To modify a list element by element we use:
  ```python
  for i in range(len(eg_list)):
      block
  ```
Break, the second.
Unravel the Loops

```python
x = [0, 0, 0]  # x = [0, 0, 0]
for i in x:
    i += 1

x = [0, 0, 0]
for i in range(len(x)):
    x[i] += 1
```
Unravel the Loops

\[ x = [0, 0, 0] \]
\[ x = [0, 0, 0] \]
\[ \text{for } i \text{ in } x: \]
\[ i += 1 \]
\[ x[i] += 1 \]

\[ i = x[0] \]
\[ i += 1 \]
\[ i = x[1] \]
\[ i += 1 \]
\[ i = x[2] \]
\[ i += 1 \]
Lists: Functions

- Lists come with lots of useful functions and methods.
- `len(list_name)`, as with strings, returns the length of the list.
- `min(list_name)` and `max(list_name)` return the min and max so long as this is well defined.
- `sum(list_name)` returns the sum of elements so long as they're numbered.
  - *Not* defined for lists of strings.
Lists: Methods

- **sort()** - sorts the list in-place so long as this is well defined. (need consistent notions of > and ==)

- **insert(index, value)** – inserts the element value at the index specified.

- **remove(value)** – removes the first instance of value.

- **count(value)** – counts the number of instances of value in the list.
List Methods

- `append(value)` - adds the value to the end of the list.
- `extend(eg_list)` - glues `eg_list` onto the end of the list.
- `pop()` - returns the last value of the list and removes it from the list.
- `pop(i)` - returns the value of the list in position `i` and removes it from the list.
Pitfalls

- Note that insert, remove, append, extend, and pop all change the length of a list.
- These methods can be called in the body of a for loop over the list that is being looped over.
- This can lead to all sorts of problems.
  - Infinite loops.
  - Skipped elements.
Pitfalls

• Note that append, extend, and pop all change the length of a list.

• These methods can be called in the body of a for loop over the list that is being looped over.

• This can lead to all sorts of problems.
  • Infinite loops.
  • Skipped elements.

• Don't Do This.
How Long are these lists at the end?

```python
x = [] y = [0,1]
for i in y:
    x.append(i)
```

```python
x = [] y = [0,1]
for i in y:
    x.extend(y)
```

```python
x = [] y = [0,1]
for i in range(2):
    x.extend(y)
y.pop()
```
How Long are these lists at the end?

```python
x = []
y = [0, 1]
for i in y:
    x.append(i)
len(x) == 2
len(y) == 2

x = []
y = [0, 1]
for i in y:
    x.extend(y)
len(x) == 4
len(y) == 2

x = []
y = [0, 1]
for i in range(2):
    x.extend(y)
    y.pop()
len(x) == 3
len(y) == 0
```
Copying a List

- We saw that as lists are mutable, we can't copy them by assigning another variable to them.
- Lists are copied in python by using `[:]`
- so the following will cause `x` to refer to a copy of `eg_list`
  ```
  x = eg_list[:]
  ```
- Now we can modify `x` without modifying `eg_list`.
Copying List Question

- x and y are both lists.
- Write code so that one adds the values of y to the end of x, and the values of x to the end of y.
Copying List Question

• x and y are both lists.

• Write code so that one adds the values of y to the end of x, and the values of x to the end of y.

```python
tmp_x = x[:]
x.extend(y)
y.extend(tmp_x)
```
List slicing.

- Sometimes we want to perform operations on a sublist.
- To refer to a sublist we use list slicing.
- \( y = x[i:j] \) gives us a list \( y \) with the elements from \( i \) to \( j-1 \) inclusive.
  - \( x[:] \) makes a list that contains all the elements of the original.
  - \( x[i:] \) makes a list that contains the elements from \( i \) to the end.
  - \( x[:j] \) makes a list that contains the elements from the beginning to \( j-1 \).
- \( y \) is a new list, so that it is not aliased with \( x \).
Strings revisited.

- Strings can be considered tuples of individual characters. (since they are immutable).
- In particular, this means that we can use the list knowledge that we gained, and apply it to strings.
  - Can reference individual characters by string[+/-i].
  - Strings are not heterogenous, they can only contain characters.
  - min() and max() defined on strings, but sum() is not.
  - You can slice strings just as you can lists.
What is the result of the following slices?

```python
s = "I am a string"

s[:10]  # s[0:10]

s[0:13]  # s[0:13]

s[-3:]  # s[-3:]

s[3:]  # s[3:]`
```
What is the result of the following slices?

`s = “I am a string”`

`s[:]
“I am a string”

`s[0:13]
“I am a string”

`s[3:]
“m a string”`

`s[:10]
“I am a str”

`s[-4:]
“ring”

`s[:−4]
“I am a st”`
String methods revisited.

- Now that we know that we can index into strings, we can look at some more string methods.
  - `find(substring)`: give the index of the first character in a matching the substring from the left or -1 if no such character exists.
  - `rfind(substring)`: same as above, but from the right.
  - `find(substring, i, j)`: same as `find()`, but looks only in string[i:j].
Nested Lists

- Because lists are heterogeneous, we can have lists of lists.
- This is useful if we want matrices, or to represent a grid or higher dimensional space.
- We then reference elements by `list_name[i][j]` if we want the jth element of the ith list.
- So then naturally, if we wish to loop over all the elements we need nested loops:

```
for item in list_name:
    for item2 in item:
        block
```
Lab Review

- Next weeks lab covers strings.
- You'll need to be comfortable with:
  - string methods.
  - writing for loops over strings.
  - string indexing.