Administration

- Exercise 2 is due tomorrow.
  - Extended one day due to midterms.
- First assignment is up.
  - Will cover it today.
- Midterm will be Jun 28\textsuperscript{th}, at 6:00.
  - In BA 2185/BA 2195
- Help Centre is still open.
  - BA 2270.
List Review

- Lists are a new type we used to store an array of variables.
  
  - Created with:
    
    ```python
    list_name = [list_elt0, ..., list_eltn]
    ```
  
  - Elements are referenced with
    
    ```python
    list_name[elt_#]
    ```
  
  - Empty lists are allowed.

  - Lists can have changing lengths and are heterogenous.

- Lists and strings can be sliced.
Aliasing/Mutability Review

- Lists are mutable.
  - That is, one can change the value of a list element or append/remove items from a list without needing to create a new list.
  - To capture this, we view a list as a list of memory addresses in our memory model.
  - Changing a list element is modifying the memory address that list element points to.
- This means lists have aliasing problems.
  - Where one has multiple variables referring to the same list, and modifying one of these lists affects all of them.
For Loop Review

- The format of a for loop is:
  ```python
  for list_elt in list_name:
    block
  ```
- The block is executed once for each element in the list.
  - list_elt refers to each list element in turn.
  - So the block code uses a different variable each time.
- Unravelling loops is a useful tool.
Lists and Relational Operators

- `!=` and `==` are defined on lists.
  - Two lists are defined to be equal if each element is equal, and they're in the same places.
  - Not based on memory addresses.
  - So `y == y[:]` evaluates to True.
Nested Lists

- Lists are heterogenous, and often one wants each list element to be another list.
  - Used to represent matrices, tiles, spreadsheet cells, etc.
- To access an element in a nested list, one uses multiple square brackets.
  \[\text{list}_\text{name}[\text{list}1\_]\text{#}][\text{list}2\_]\text{#}\ldots\]
- The closest brackets to the name are evaluated first.
Nested Lists

• Lists are heterogenous, and often one wants each list element to be another list.
  • Used to represent matrices, tiles, spreadsheet cells, etc.
• To access an element in a nested list, one uses multiple square brackets.
  \[\text{list\_name}[\text{list1\_#}][\text{list2\_#}]\ldots\]
• The closest brackets to the name are evaluated first.
Nested Lists

- Lists are heterogenous, and often one wants each list element to be another list.
  - Used to represent matrices, tiles, spreadsheet cells, etc.
- To access an element in a nested list, one uses multiple square brackets.
  list_name[list1_] [list2_] ...
- The closest brackets to the name are evaluated first.
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]
print(eg_list[2][1][0])
```

Global

```
Global
eg_list: 0x1
```

```
<table>
<thead>
<tr>
<th>0x5</th>
<th>int</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x7</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>0x67</td>
<td>str</td>
<td>'a'</td>
</tr>
<tr>
<td>0x1</td>
<td>list</td>
<td>0x5 0x10 0x8</td>
</tr>
<tr>
<td>0x24</td>
<td>list</td>
<td>0x7 0x67</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x13 0x24</td>
</tr>
</tbody>
</table>
```
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]

print neg_list[2][1][0]
```

Global

<table>
<thead>
<tr>
<th>eg_list: 0x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
</tr>
<tr>
<td>0x8 0x13 0x24</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td>0x1 0x5 0x10 0x8</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td>0x24 0x7 0x67</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td>0x10 1</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>0x13 4</td>
</tr>
<tr>
<td>str</td>
</tr>
<tr>
<td>0x67 a</td>
</tr>
</tbody>
</table>

0x5 int 0 0x10 int 1 0x13 int 4 0x7 bool True 0x67 str a
Nested Lists and the Memory Model

\[
\text{eg\_list} = [0, 1, [4, [True, 'a']]]
\]

\[
\text{print }
\]

```
Global

\text{eg\_list: 0x1}
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x24</td>
<td>list</td>
<td>0x7</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x13</td>
</tr>
<tr>
<td>0x1</td>
<td>list</td>
<td>0x5</td>
</tr>
<tr>
<td>0x67</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>0x67</td>
<td>str</td>
<td>'a'</td>
</tr>
</tbody>
</table>
```
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]

print eg_list[2][1][0]
```

<table>
<thead>
<tr>
<th>Global</th>
<th>eg_list: 0x1</th>
</tr>
</thead>
</table>

- `0x1`: Global `eg_list`

```
| 0x5 | int  | 0 |
| 0x10| int  | 1 |
| 0x13| int  | 4 |

- `0x7`: `True`

- `0x67`: 'a'

- `0x24`: list
  - `0x7`: `0x13` 0x24
  - `0x67`: list
  - `0x8`: list
    - `0x5`: 0x10 0x8
Nested Lists and the Memory Model

```python
global_list = [0, 1, [4, [True, 'a']]]
```

```python
print global_list[2][1][0]
```

Global

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>list</td>
<td>0x5</td>
</tr>
<tr>
<td>0x7</td>
<td>list</td>
<td>0x13</td>
</tr>
<tr>
<td>0x67</td>
<td>list</td>
<td>0x24</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x67</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>0x13</td>
<td>str</td>
<td>'a'</td>
</tr>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x5</td>
</tr>
</tbody>
</table>

June 14 2012
Nested Lists and the Memory Model

eg_list = [0, 1, [4, [True, 'a']]]

print 0x1[2][1][0]
Nested Lists and the Memory Model

eg_list = [0,1,[4, [True, 'a']]]

print 0x1[2][1][0]
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]
```

```
print 0x1[2][1][0]
```

Global
---

```python
eg_list: 0x1
```

```
0x5 int 0
0x10 int 1
0x13 int 4
0x7 bool True
0x67 str 'a'
0x24 list 0x7 0x67
0x8 list 0x13 0x24
0x1 list 0x5 0x10 0x8
```
Nested Lists and the Memory Model

```
eg_list = [0, 1, [4, [True, 'a']]]

print 0x8[1][0]
```

Global

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>eg_list: 0x1</td>
</tr>
</tbody>
</table>

```
0x5  0
int

0x10 1
int

0x13 4
int

0x24

list 0x7 0x67

0x8
list 0x13 0x24

0x1
list 0x5 0x10 0x8

0x7
bool
True

0x67
str
'a'
```
Nested Lists and the Memory Model

```python
eg_list = [0,1,[4, [True, 'a']]]
```

```python
print 0x8[1][0]
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x4</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>0x67</td>
<td>str</td>
<td>'a'</td>
</tr>
</tbody>
</table>
```

Global

```
eg_list: 0x1
```
Nested Lists and the Memory Model

eg_list = [0,1,[4, [True, 'a']]]

print 0x8[1][0]

Global

eg_list: 0x1

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

<table>
<thead>
<tr>
<th>0x10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x13</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x24</th>
<th>0x7</th>
<th>0x67</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8</th>
<th>0x13</th>
<th>0x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x1</th>
<th>0x5</th>
<th>0x10</th>
<th>0x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]
```

```python
print 0x8[1][0]
```

```
Global

<table>
<thead>
<tr>
<th>Global</th>
<th>eg_list: 0x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>int 0x5</td>
<td>0</td>
</tr>
<tr>
<td>int 0x10</td>
<td>1</td>
</tr>
<tr>
<td>int 0x13</td>
<td>4</td>
</tr>
<tr>
<td>bool 0x7</td>
<td>True</td>
</tr>
<tr>
<td>str 0x67</td>
<td>'a'</td>
</tr>
<tr>
<td>list 0x24</td>
<td>0x7 0x67</td>
</tr>
<tr>
<td>list 0x8</td>
<td>0x13 0x24</td>
</tr>
<tr>
<td>list 0x1</td>
<td>0x5 0x10 0x8</td>
</tr>
</tbody>
</table>
```

June 14 2012
Nested Lists and the Memory Model

eg_list = [0, 1, [4, [True, 'a']]]

→ print 0x24[0]
Nested Lists and the Memory Model

```
eg_list = [0, 1, [4, [True, 'a']]]
print 0x24[0]
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x24</td>
<td>list</td>
<td>0x7</td>
</tr>
<tr>
<td>0x67</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x13</td>
</tr>
<tr>
<td>0x24</td>
<td>list</td>
<td>0x5</td>
</tr>
<tr>
<td>0x10</td>
<td>list</td>
<td>0x10</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x8</td>
</tr>
</tbody>
</table>

Global:

```
eg_list: 0x1
```
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]

print 0x24[0]
```

**Global**

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>eg_list: 0x1</td>
</tr>
</tbody>
</table>

**Memory Model Diagram**

```
| 0x5  | 0x10 |
| int  | int  |
| 0    | 1    |

| 0x13 |
| int  |
| 4    |

| 0x24 |
| list |
| 0x7  |
| 0x67 |

| 0x8  |
| list |
| 0x13 |
| 0x24 |

| 0x1  |
| list |
| 0x5  |
| 0x10 |
| 0x8  |

| 0x67 |
| str  |
| 'a'  |
```
Nested Lists and the Memory Model

\[
\text{eg_list} = [0, 1, [4, [True, 'a']]]
\]

\[\text{print } 0x24[0]\]

Global

eg_list: 0x1
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]
```

→ `print 0x7`

<table>
<thead>
<tr>
<th>Global</th>
<th>eg_list: 0x1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
0x5
| int  | 0 |
0x10 |
| int  | 1 |

0x13
| int  | 4 |

0x24
| list | 0x7 | 0x67 |

0x8
| list | 0x13 | 0x24 |

0x1
| list | 0x5  | 0x10 | 0x8 |
```

June 14 2012
Nested Lists and the Memory Model

\[ \text{eg_list} = [0, 1, [4, [\text{True}, 'a']]] \]

\[ \text{print 0x7} \]

**Global**

| 0x1 | 0x5 | int  | 0  |
| 0x10 | 0x10 | int  | 1  |
| 0x13 | 0x13 | int  | 4  |
| 0x7 | 0x7 | bool | True |
| 0x67 | 0x67 | str  | 'a' |
| 0x8 | 0x8 | list | 0x13 0x24 |
| 0x1 | 0x1 | list | 0x5 0x10 0x8 |
| 0x24 | 0x24 | list | 0x7 0x67 |
Nested Lists and the Memory Model

\[
\text{eg\_list} = [0, 1, [4, [\text{True}, 'a']]]
\]

\[\text{print 0x7}\]

### Global Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>list</td>
<td>0x7</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x13</td>
</tr>
<tr>
<td>0x24</td>
<td>list</td>
<td>0x7, 0x67</td>
</tr>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x7</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>0x67</td>
<td>str</td>
<td>'a'</td>
</tr>
<tr>
<td>0x8</td>
<td>list</td>
<td>0x5, 0x10, 0x8</td>
</tr>
</tbody>
</table>
Nested Lists and the Memory Model

```
eg_list = [0, 1, [4, [True, 'a']]]
```

```
print 0x7
```

```
Global

eg_list: 0x1
```

```
<table>
<thead>
<tr>
<th></th>
<th>0x5</th>
<th>0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x13</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>0x24</th>
<th>0x7</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>0x7</td>
<td>0x67</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>0x8</th>
<th>0x13</th>
<th>0x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>0x13</td>
<td>0x24</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>0x1</th>
<th>0x5</th>
<th>0x10</th>
<th>0x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>0x5</td>
<td>0x10</td>
<td>0x8</td>
<td></td>
</tr>
</tbody>
</table>
```

```
0x67 True
```

```
0x7 'a'
```

June 14 2012
Nested Lists and the Memory Model

```python
eg_list = [0, 1, [4, [True, 'a']]]

print(True)
```

Global
---

| $\text{eg}\_\text{list}$: | 0x1 |

---

| 0x5 | int | 0 |
| 0x10 | int | 1 |
| 0x13 | int | 4 |
| 0x7 | bool | True |
| 0x67 | str | 'a' |
| 0x1 | list | 0x5 0x10 0x8 |
| 0x8 | list | 0x13 0x24 |
| 0x24 | list | 0x7 0x67 |
Tuples

- Similar to lists, but not mutable.
  - So they cannot be changed once they are initialised.
  - Aliasing is not a problem
  - Faster.
- Syntax for creating tuples is like that of lists, but with parentheses instead of square brackets.
- Syntax for accessing tuple elements is like that of lists.
Tuples

- Syntax for creating a tuple:
  
  \[
  \text{tuple\_name} = (\text{elt0, elt1, \ldots, eltn})
  \]

  - Note that this is ambiguous for a single element.
  - \(a = (10)\) could be an integer or tuple

- Syntax for accessing a tuple element:
  
  \[
  \text{tuple\_name}[\text{elt\#}]
  \]
Tuples

- Syntax for creating a tuple:
  \[
  \text{tuple\_name} = (\text{elt0}, \text{elt1}, \ldots, \text{eltn})
  \]
  - Note that this is ambiguous for a single element.
  - \( a = (10) \) could be an integer or tuple
  - \( a = (10,) \) is unambiguous.

- Syntax for accessing a tuple element:
  \[
  \text{tuple\_name}[\text{elt\#}]
  \]
Assignment Statements

- Evaluate the right side first!
- Variables can be thought of as look up tables.
- The point of an assignment statement is to connect a memory location to a variable name.
- This means that one needs to evaluate the right side first, before one can do anything else.
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)
```

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
</table>

June 14 2012
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: ?</td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

```
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)
```

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: ?</td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4
```

- $x = 0$
- $x = 13 + 4$
- $x = x + f(4)$
- $x = 10 + f(x)$

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$: ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)

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

Global

| x: ? |
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0x5
x = 13 + 4
x = x + f(4)
x = 10 + f(x)
```

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: ?</td>
</tr>
</tbody>
</table>

- `x = 0x5` (0x5)
- `x = 13 + 4`
- `x = x + f(4)`
- `x = 10 + f(x)`
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)
```

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

June 14 2012
def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)
def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x5</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0

x = ?

x = x + f(4)

x = 10 + f(x)
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0x3</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x5</td>
<td>--</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0x3</th>
<th>13</th>
</tr>
</thead>
</table>
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)

Global

<table>
<thead>
<tr>
<th>x: 0x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0x3</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0x3</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x13</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)

Global

<table>
<thead>
<tr>
<th>x: 0x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
</tr>
<tr>
<td>int 17</td>
</tr>
<tr>
<td>0x5</td>
</tr>
<tr>
<td>int 0</td>
</tr>
<tr>
<td>0x3</td>
</tr>
<tr>
<td>int 13</td>
</tr>
<tr>
<td>0x13</td>
</tr>
<tr>
<td>int 4</td>
</tr>
</tbody>
</table>
```

June 14 2012
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0
x = 17
x = x + f(4)
x = 10 + f(x)
```

Global

<table>
<thead>
<tr>
<th>Global</th>
<th>x: 0x5</th>
</tr>
</thead>
</table>

```
int 0x11 17
int 0x5  0
int 0x3 13
int 0x13 4
```
def f(x):
    return x + 4

x = 0
x = 0x11
x = x + f(4)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x5</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>int</td>
</tr>
<tr>
<td>0x5</td>
<td>int</td>
</tr>
<tr>
<td>0x3</td>
<td>int</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 0x11
x = x + f(4)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
<th>x: 0x11</th>
</tr>
</thead>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)

Global
x: 0x11
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)
def f(x):
    return x + 4

x = 0

x = 13 + 4

x = x + f(4)

x = 10 + f(x)

<table>
<thead>
<tr>
<th>0x11</th>
<th>int</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x3</td>
<td>int</td>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = x + f(4)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global x: 0x11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11 17 int</td>
</tr>
<tr>
<td>0x3  13 int</td>
</tr>
<tr>
<td>0x13  4 int</td>
</tr>
<tr>
<td>0x5  0 int</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0

x = 13 + 4

x = 0x11 + f(4)

x = 10 + f(x)

Global

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int 0x11</td>
<td>17</td>
</tr>
<tr>
<td>int 0x5</td>
<td>0</td>
</tr>
<tr>
<td>int 0x3</td>
<td>13</td>
</tr>
<tr>
<td>int 0x13</td>
<td>4</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0

x = 13 + 4

x = 0x11 + f(4)

x = 10 + f(x)

Global

x: 0x11
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

Global

x: 0x11
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>int</td>
<td>17</td>
</tr>
<tr>
<td>0x3</td>
<td>int</td>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
</tbody>
</table>

Global

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
<td></td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

Global

<table>
<thead>
<tr>
<th>0x11</th>
<th>int</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>0x3</td>
<td>int</td>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
</tbody>
</table>

x: 0x11
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(0x13)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(0x13)
x = 10 + f(x)

Global
x: 0x11
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(0x13)
x = 10 + f(x)

Global
x: 0x11

[Memory Table]

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>int</td>
<td>17</td>
</tr>
<tr>
<td>0x3</td>
<td>int</td>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x5</td>
<td>int</td>
<td>0</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(0x13)
x = 10 + f(x)

Global
x: 0x11
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(0x13)
x = 10 + f(x)

Global
x: 0x11
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(0x13)
x = 10 + f(x)
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

Global
x: 0x11

f
x: 0x13

0x11
17
int

0x5
0
int

0x3
13
int

0x13
4
int
Assignment Statements & Memory Model

def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

| x: 0x11 | 17 |
| int    | int |
| 0x11   | 17 |
| 0x13   | 13 |
| 0x5    | 0  |
| 0x3    | 13 |
| 0x13   | 4  |

Global
x: 0x11
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x13</td>
</tr>
<tr>
<td>Global</td>
</tr>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x13</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

Global x: 0x11

<table>
<thead>
<tr>
<th>f</th>
<th>x: 0x13</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x13</td>
</tr>
<tr>
<td>int</td>
<td>17</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>13</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
</tbody>
</table>
Assignment Statements & Memory Model

```python
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>17</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x3</td>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
<td>4</td>
</tr>
</tbody>
</table>
def f(x):
    return 0x13 + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 0x13 + 4
x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

Global
x: 0x11

f
x: 0x13
def f(x):
    return 0x13 + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 4 + 4
x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 4 + 4
x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 4 + 4

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 8

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 8

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

<table>
<thead>
<tr>
<th>x: 0x13</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x: 0x11</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>
def f(x):
    return 0x18
x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)
def f(x):
    return 0x18

x = 0
x = 13 + 4
x = 17 + f(4)
x = 10 + f(x)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>int</td>
<td>17</td>
</tr>
<tr>
<td>0x3</td>
<td>int</td>
<td>13</td>
</tr>
<tr>
<td>0x13</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x18</td>
<td>int</td>
<td>8</td>
</tr>
</tbody>
</table>

Global

x: 0x13

f

x: 0x11
Assignment Statements & Memory Model

```python
def f(x):
    return 0x18

x = 0
x = 13 + 4
x = 17 + 0x18
x = 10 + f(x)
```

<table>
<thead>
<tr>
<th>Global</th>
<th>x: 0x11</th>
</tr>
</thead>
</table>

- 0x11: int 17
- 0x13: int 4
- 0x18: int 8
- 0x5: int 0
- 0x3: int 13
```
def f(x):
    return 0x18
x = 0
x = 13 + 4
x = 17 + 0x18
x = 10 + f(x)
```

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>

| 0x11 |  
| int  | 17  |
| 0x5  | int  | 0   |
| 0x3  | int  | 13  |
| 0x13 | int  | 4   |
| 0x18 | int  | 8   |
def f(x):
    return 0x18

x = 0

x = 13 + 4

x = 17 + 0x18

x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>
def f(x):
    return 0x18

x = 0
x = 13 + 4
x = 17 + 0x18
x = 10 + f(x)
def f(x):
    return 0x18
x = 0
x = 13 + 4
x = 17 + 8
x = 10 + f(x)

Global
x: 0x11
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 17 + 8
x = 10 + f(x)
Assignment Statements & Memory Model

\begin{align*}
\text{def } f(x) & : \\
\quad \text{return } x + 4 \\
\text{x} & = 0 \\
\text{x} & = 13 + 4 \\
\text{x} & = 25 \\
\text{x} & = 10 + f(x)
\end{align*}

Global
\begin{tabular}{|c|}
\hline
x: 0x11 \\
\hline
\end{tabular}
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 25
x = 10 + f(x)

Global
x: 0x11
def f(x):
    return x + 4
x = 0
x = 13 + 4
x = 0x38
x = 10 + f(x)

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0x11</td>
</tr>
</tbody>
</table>
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 0x38
x = 10 + f(x)

Global
x: 0x11
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 0x38
x = 10 + f(x)

Global
x: 0x38
def f(x):
    return x + 4

x = 0
x = 13 + 4
x = 0x38
x = 10 + f(x)

Global
x: 0x38
Break, the first.
While Loops

- For Loops are great if we know how many times we want to loop over something.
  - In other cases, not so great.
  - If you want to enforce a legal input, for example
  - If you're playing a game and don't know how many turns there will be.
  - If we want to loop indefinitely.

- In these cases we use a while loop.
While loop syntax

```python
while condition:
    block
```

- The **condition** evaluates to a boolean variable.
- The **block** is executed so long as the condition is true.
- If the **condition** is **False** the first time the while loop is seen, the **block** is never executed.
Unravelling While Loops

• We saw that for loops can be unravelled to make the program simpler to analyse, albeit longer.

• While loops are more complicated and are not always possible to be unravelled.
  • For eg. if the number of times the block is executed is dependent on user input.

• So to analyse them we need to use other tools.
  • Debugger, visualiser, hand simulation, etc.
While vs. For

- Every for loop can be written as a while loop.
- Not ever while loop can be written as a for loop:
  ```python
  while True:
      block
  ```
- How do we choose between while and for?
While vs. For

• Every for loop can be written as a while loop.
• Not ever while loop can be written as a for loop:
  ```python
  while True:
      block
  ```
• How do we choose between while and for?
  • for is simpler.
  • In general we prefer simpler loops, as they are easier to read.
While vs. For

- While loops are used when:
  - We want infinite loops.
  - We want to loop some number of times that we can't predict.
  - That is, we want to loop until some condition is met.
Docstrings

- Recall that the first line of a docstring contains type information.
  - Specifically it tells us the parameter types and the expected output type.
  - '''(parameter types) -> output type'''
Docstrings

- Recall that the first line of a docstring contains type information.
  - Specifically it tells us the parameter types and the expected output type.
  - `' '(parameter types) -> output type''

- If we want to return multiple things, we wrap them with a tuple and use the following format
  - `' '(parameter types) -> (output types)'''
Docstrings

- Recall that the first line of a docstring contains type information.
  - Specifically it tells us the parameter types and the expected output type.
  - '''(parameter types) -> output type'''

- If we want to return multiple things, we wrap them with a tuple and use the following format
  - '''(parameter types) -> (output types)'''
  - '''(NoneType) -> (int, str, list)'''
Docstrings

- Recall that the first line of a docstring contains type information.
  - Specifically it tells us the parameter types and the expected output types.
  - `''(parameter types) -> (output types)''`
Docstrings

- Recall that the first line of a docstring contains type information.
  - Specifically it tells us the parameter types and the expected output types.
  - '''(parameter types) -> (output types)'''
- This is only for the benefit of the humans writing and reading the program.
- Python does not check or enforce this convention in any way.
- Changing your docstring does not change your function in any way.
Docstrings

- Recall that the first line of a docstring contains type information.
  - Specifically it tells us the parameter types and the expected output types.
  - ''(parameter types) -> (output types)''
- This is only for the benefit of the humans writing and reading the program.
- Python does not check or enforce this convention in any way.
- **Changing your docstring does not change your function in anyway.**
Indentation

• I have been using indented blocks a lot when giving python syntax.

    for item in list:
        block
Indentation

• I have been using indented blocks a lot when giving python syntax.
  
  ```python
  while condition:
    block
  ```
Indentation

- I have been using indented blocks a lot when giving python syntax.

```python
if condition:
    block1
else:
    block2
```
Indentation

- I have been using indented blocks a lot when giving python syntax.

```python
def foo(parameters):
    block
```
Indentation

- I have been using indented blocks a lot when giving python syntax.
  ```python
def foo(parameters):
    block
  ```
- I want to make it explicit that these blocks last as long as the indentation is at least one tab.
  - It can be more, because blocks can contain sub blocks.
Sub-blocks

def foo(parameters):
    block
        sub-block
    block
Sub-blocks

def foo(x):
    if (x % 2 == 0):
        sub-block
        block

• Recall:
    if condition:
        block1
def foo(x):
    if (x % 2 == 0):
        sub-block
    block

• Recall:
  if condition:
    block1
def foo(x):
    if (x % 2 == 0):
        sub-block
    block

- Recall:
  if condition:
      block1
Sub-blocks

def foo(x):
    if (x % 2 == 0):
        sub-block
    block

• Recall:
  if condition:
      block1
Sub-blocks

def foo(x):
    if (x % 2 == 0):
        print 'even'
    print 'odd'
Indentation

- I have been using indented blocks a lot when giving python syntax.
  ```python
def foo(parameters):
    block
  ```
- I want to make it explicit that these blocks last as long as the indentation is at least one tab.
  - It can be more, because blocks can contain sub blocks.
- When you stop indenting the block ends.
Indentation

• When you stop indenting the block ends.
  def foo(parameters):
    block1
  block2
    block3

• Blocks 1, 2 and 3 are all different, and only block 1 is inside the function definition.

• If the last line of block2 is not something that expects a block to follow it, block 3 is illegal.
Indentation

- When you stop indenting the block ends. White space does not count as ending a block.

```python
def foo(parameters):
    block1

    block3
```

- Here block 1 and block 3 are considered to be part of the same block, regardless of whether or not the empty line contains spaces/tabs/etc.
Indentation

- When you stop indenting the block ends.
  White space does not count as ending a block.
  ```python
def foo(parameters):
    block1

    block3
  ```

- Here block 1 and block 3 are considered to be part of the same block, regardless of whether or not the empty line contains spaces/tabs/etc.

  Note that this may vary depending on the IDE.
Break, the second
Files.

- So far we've seen some basic file stuff.
- Media opens files
- The testing script for Assignment 1 opens a file.
Files as types.

- Python has a type used to deal with files.
- There are four main things we want to do with files:
  - Figure out how to open them.
  - Figure out how to read them.
  - Figure out how to write to them.
  - Figure out how to close them.
Opening files.

- Can hardcoded the filename in the code.
  - Like done in the script for assignment 1.
- Can ask the user for a file name using `raw_input()`
- Some modules have their own built-in functions for opening files.
  - `media` has `choose_file()` which opens a dialog window.
Opening files.

- Once we have a filename we can call open:
  
  ```python
  open(filename, 'r')  # for reading (this is the default mode).
  open(filename, 'w')  # for writing (erases the contents of a file).
  open(filename, 'a')  # for appending (keeps the contents of the file).
  ```

- This function returns a new object, a file object.
Reading Files.

• The most basic way is the read the whole file into a string:

```python
filename.read() - returns a string that is the contents of the entire file.
```

• Not recommended for big files.

• Can read a single line of the file.

```python
filename.readline() - reads a line of the filename.
```

• A subsequent call the readline() will read the next line of the file, the first line is lost.
Reading Files.

- Can read a fixed number of characters.
  
  ```python
  filename.read(10) – will read 10 characters.
  ```

- If you call it again, it will start reading from the place after the characters that it has read.

- Can read the file a line at a time.
  
  ```python
  for line in filename: 
    print line 
  ```

- Note that the string `split` method is often very useful.
Writing to Files.

- Write to files using:

  `filename.write("This is a string")`

- Multiple writes are concatenated.

- Need to open a file in append or write mode to write to it.

- Append mode will add the strings to the end of the file.
Closing Files.

- Close a file with:
  
  `filename.close()`

- Generally a good idea.

- Frees up system resources.