# Announcements For This Lecture

## Readings
- Sections 5.1-5.7 today
- Chapter 4 for Tuesday

## Assignment 1
- Due Wed, Sep. 25th
  - Consultants all weekend!
- Also the lab next week
  - Open office hours for A1
  - Turn in A1 to get credit
- Pair up now in CMS!
  - Too late after you submit

## Assignment 2
- Will post on Wed
  - Written assignment
  - Do while revising A1
A1: The Module urllib2

- Module urllib2 is used to read web pages
  - Function urlopen creates a url object
    - `u = urllib2.urlopen('http://www.cornell.edu')`
  - `url` has a method called `read()`
    - Returns contents of web page
    - **Usage:** `s = u.read() # s is a string`
### Types of Testing

#### Black Box Testing
- Function is “opaque”
  - Test looks at what it does
  - **Fruitful**: what it returns
  - **Procedure**: what changes
- **Example**: Unit tests
- **Problems**:
  - Are the tests everything?
  - What caused the error?

#### White Box Testing
- Function is “transparent”
  - Tests/debugging takes place inside of function
  - Focuses on where error is
- **Example**: Use of print
- **Problems**:
  - Much harder to do
  - Must remove when done
# Black Box Example from Lab 3

<table>
<thead>
<tr>
<th>Fruitful Function</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td># Create the input value</td>
<td># Create the input value</td>
</tr>
<tr>
<td>p = tuple3d.Point(1.0,2.0,3.0)</td>
<td>p = tuple3d.Point(1.0,2.0,3.0)</td>
</tr>
<tr>
<td># Test the input value</td>
<td># Test the input value</td>
</tr>
<tr>
<td>result = has_a_zero(p)</td>
<td>cycle_left(p)</td>
</tr>
<tr>
<td># Compare to expected output</td>
<td># Compare to expected output</td>
</tr>
<tr>
<td>assert_equals(False,result)</td>
<td>assert_floats_equal(2.0,p.x)</td>
</tr>
<tr>
<td></td>
<td>assert_floats_equal(3.0,p.y)</td>
</tr>
<tr>
<td></td>
<td>assert_floats_equal(1.0,p.z)</td>
</tr>
</tbody>
</table>
Structure vs. Flow

Program Structure

• Way statements are presented
  ▪ Order statements are listed
  ▪ Inside/outside of a function
  ▪ Will see other ways…
• Indicate possibilities over multiple executions

Program Flow

• Order statements are executed
  ▪ Not the same as structure
  ▪ Some statements duplicated
  ▪ Some statements are skipped
• Indicates what really happens in a single execution

Have already seen this difference with functions
## Structure vs. Flow: Example

<table>
<thead>
<tr>
<th>Program Structure</th>
<th>Program Flow</th>
</tr>
</thead>
</table>
| ```
def foo():
    print 'Hello'

# Application code
if __name__ == 'main':
    foo()
    foo()
    foo()
``` | ```
>>> python foo.py
'Hello'
'Hello'
'Hello'
``` |

**Statement listed once**

**Statement executed 3x**

Bugs can occur when we get a flow other than one that we were expecting.
## Conditionals: If-Statements

### Format

<table>
<thead>
<tr>
<th>if &lt;boolean-expression&gt;:</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;statement&gt;</td>
<td># Put x in z if it is positive</td>
</tr>
<tr>
<td>...</td>
<td>if x &gt; 0:</td>
</tr>
<tr>
<td>&lt;statement&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### Execution:

if <boolean-expression> is true, then execute all of the statements indented directly underneath (until first non-indentated statement)
Conditionals: If-Else-Statements

**Format**

```
if <boolean-expression>:
    <statement>
  ...
else:
    <statement>
  ...
```

**Example**

```
# Put max of x, y in z
if x > y:
    z = x
else:
    z = y
```

**Execution:**

if `<boolean-expression>` is true, then execute statements indented under if; otherwise execute the statements indented under else.
Conditionals: “Control Flow” Statements

\[\textbf{if} \ b: \]  
\[s1 \ # \ statement\]  
\[s3\]

\[\textbf{if} \ b:\]  
\[s1\]  
\[\textbf{else}:\]  
\[s2\]  
\[s3\]

Branch Point: Evaluate & Choose

Flow
Program only takes one path each execution
def max(x,y):
    """Returns: max of x, y"""
    # simple implementation
    if x > y:
        return x
    return y

max(0,3):
**Program Flow and Call Frames**

```python
def max(x, y):
    """Returns: max of x, y"""
    # simple implementation
    if x > y:
        return x
    return y
```

`max(0, 3):`

Frame sequence depends on flow

Skips line 2
def max(x,y):
    """Returns: max of x, y"""
    # simple implementation
    if x > y:
        return x
    return y

max(0,3):

Frame sequence depends on flow

Skips line 2
def max(x,y):
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    1 if x > y:
        temp = x
    2 temp = x
    3 x = y
    4 y = temp
    5 return y

• temp is needed for swap
  ▪ x = y loses value of x
  ▪ “Scratch computation”
  ▪ Primary role of local vars

• max(3,0):

\[
\begin{array}{c|c|c}
\text{max} & 1 \\
\hline
x & 3 & y \\
0 & & 0
\end{array}
\]
Program Flow vs. Local Variables

def max(x,y):
    """Returns: max of x, y""
    # swap x, y
    # put the larger in y
    1 if x > y:
    2     temp = x
    3     x = y
    4     y = temp
    5 return y

• temp is needed for swap
  ▪ x = y loses value of x
  ▪ “Scratch computation”
  ▪ Primary role of local vars

• max(3,0):

  max
  x 3 y 0

  2

9/19/13
def max(x, y):
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return y

• temp is needed for swap
  ▪ x = y loses value of x
  ▪ “Scratch computation”
  ▪ Primary role of local vars

• max(3, 0):

```
+---+---+---+
| max| 3 | 3 |
+---+---+---+
| x  | 3 | y  |
+---+---+---+
| y  | 0 | temp| 3 |
+---+---+---+
```
def max(x,y):
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return y

• temp is needed for swap
  ▪ x = y loses value of x
  ▪ “Scratch computation”
  ▪ Primary role of local vars

• max(3,0):

  \[
  \begin{array}{|c|c|}
  \hline
  \text{max} & 4 \\
  \hline
  \text{x} & 0 \\
  \hline
  \text{y} & 0 \\
  \hline
  \text{temp} & 3 \\
  \hline
  \end{array}
  \]
def max(x,y):
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return y

• temp is needed for swap
  ▪ x = y loses value of x
  ▪ “Scratch computation”
  ▪ Primary role of local vars

• max(3,0):

```
<table>
<thead>
<tr>
<th>max</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>temp</td>
<td>3</td>
</tr>
</tbody>
</table>
```

9/19/13 Conditionals & Control Flow
def max(x, y):
    """Returns: max of x, y""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return y

• temp is needed for swap
  ▪ x = y loses value of x
  ▪ “Scratch computation”
  ▪ Primary role of local vars

• max(3, 0):

```
  max
  x  0  y  3
  temp 3
```

def max(x,y):
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return temp

• Value of max(3,0)?
  A: 3
  B: 0
  C: Error!
  D: I do not know
Program Flow vs. Local Variables

def max(x,y):
    
    
    
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return temp

• Value of max(3,0)?
  A: 3 CORRECT
  B: 0
  C: Error!
  D: I do not know

• Local variables last until
  ▪ They are deleted or
  ▪ End of the function

• Even if defined inside if
def max(x,y):
    """Returns: max of x, y""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return temp

• Value of max(0,3)?
  A: 3
  B: 0
  C: Error!
  D: I do not know
Program Flow vs. Local Variables

def max(x, y):
    """Returns: max of x, y"""
    # swap x, y
    # put the larger in y
    if x > y:
        temp = x
        x = y
        y = temp
    return temp

• Value of max(0, 3)?
  A: 3
  B: 0
  C: Error!  CORRECT
  D: I do not know

• Variable existence depends on flow

• Understanding flow is important in testing
Program Flow and Testing

- Must understand which flow caused the error
  - Unit test produces error
  - Visualization tools show the current flow for error

- Visualization tools?
  - print statements
  - Advanced tools in IDEs (Integrated Dev. Environ.)

```python
# Put max of x, y in z
print 'before if'
if x > y:
    print 'if x>y'
    z = x
else:
    print 'else x>y'
    z = y
print 'after if'
```
Program Flow and Testing

- Call these tools **traces**
- No requirements on how to implement your traces
  - Less print statements ok
  - Do not need to word them exactly like we do
  - Do whatever is easiest for you to see the flow
- **Example**: flow.py

```python
# Put max of x, y in z
print 'before if'
if x > y:
    print 'if x>y'
    z = x
else:
    print 'else x<=y'
    z = y
print 'after if'
```

9/19/13
Conditionals & Control Flow

25
# Watches vs. Traces

<table>
<thead>
<tr>
<th><strong>Watch</strong></th>
<th><strong>Trace</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Visualization tool (e.g. <code>print</code> statement)</td>
<td>Visualization tool (e.g. <code>print</code> statement)</td>
</tr>
<tr>
<td>Looks at <strong>variable value</strong></td>
<td>Looks at <strong>program flow</strong></td>
</tr>
<tr>
<td>Often after an assignment</td>
<td>Before/after any point</td>
</tr>
<tr>
<td>What you did in lab</td>
<td>where flow can change</td>
</tr>
</tbody>
</table>

9/19/13 Conditionals & Control Flow 26
def cycle_left(p):
    print 'Start cycle_left()'
    p.x = p.y
    print p.x
    p.y = p.z
    print p.y
    p.z = p.x
    print p.z
    print 'End cycle_left()'

Example: flow.py
Local Variables Revisited

- Never refer to a variable that might not exist
- Variable “scope”
  - Block (indented group) where it was first assigned
  - Way to think of variables; not actually part of Python
- **Rule of Thumb**: Limit variable usage to its scope

```python
def max(x,y):
    """Returns: max of x, y""
    # swap x, y
    # put larger in temp
    if x > y:
        temp = x
        x = y
        y = temp
    return temp
```

```
First assigned
```

```
Outside scope
```
Local Variables Revisited

- Never refer to a variable that might not exist

- Variable “scope”
  - Block (indented group) where it was first assigned
  - Way to think of variables; not actually part of Python

- Rule of Thumb: Limit variable usage to its scope

```python
def max(x,y):
    # swap x, y
    # put larger in temp
    temp = y
    if x > y:
        temp = x
    return temp
```

- First assigned
- Inside scope
def max(x, y):
    """Returns:
    max of x, y"
    if x > y:
        return x
    else:
        return y

Which is better? Matter of preference

There are two returns! But only one is executed
# Conditionals: If-Elif-Else-Statements

## Format

```python
if <boolean-expression>:
    <statement>
...
elif <boolean-expression>:
    <statement>
...
else:
    <statement>
...
```

## Example

```python
# Put max of x, y, z in w
if x > y and x > z:
    w = x
elif y > z:
    w = y
else:
    w = z
```
**Conditionals: If-Elif-Else-Statements**

**Format**

```python
if <boolean-expression>:
    <statement>
    ...
elif <boolean-expression>:
    <statement>
    ...
else:
    <statement>
    ...
```

**Notes on Use**

- No limit on number of `elif`
  - Can have as many as want
  - Must be between `if, else`
- The `else` is always optional
  - `if-elif` by itself is fine
- Booleans checked in order
  - Once it finds a true one, it skips over all the others
  - `else` means all are false
## Conditional Expressions

### Format

\[ e_1 \text{ if } bexp \text{ else } e_2 \]

- \( e_1 \) and \( e_2 \) are any expression
- \( bexp \) is a boolean expression
- This is an expression!

### Example

```plaintext
# Put max of x, y in z
z = x if x > y else y
```

expression, not statement