Lecture 8

Conditionals & Control Flow
Announcements For This Lecture

Readings

• Sections 5.1-5.7 today
• Chapter 4 for Tuesday

Assignment 2

• Posted Today
  ▪ Written assignment
  ▪ Do while revising A1

Assignment 1

• Due TOMORROW
  ▪ Due before midnight
  ▪ Submit something…
  ▪ Can resubmit to Sep. 28

• Grades posted Saturday
• Complete the Survey
  ▪ Must answer individually

9/18/14 Conditionals & Control Flow 2
# Types of Testing

<table>
<thead>
<tr>
<th>Black Box Testing</th>
<th>White Box Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function is “opaque”</strong></td>
<td><strong>Function is “transparent”</strong></td>
</tr>
<tr>
<td>- Test looks at what it does</td>
<td>- Tests/debugging takes place inside of function</td>
</tr>
<tr>
<td>- <strong>Fruitful</strong>: what it returns</td>
<td>- Focuses on where error is</td>
</tr>
<tr>
<td>- <strong>Procedure</strong>: what changes</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong>: Unit tests</td>
<td><strong>Example</strong>: Use of print</td>
</tr>
<tr>
<td><strong>Problems:</strong></td>
<td><strong>Problems:</strong></td>
</tr>
<tr>
<td>- Are the tests everything?</td>
<td>- Much harder to do</td>
</tr>
<tr>
<td>- What caused the error?</td>
<td>- Must remove when done</td>
</tr>
</tbody>
</table>
Fruitful Function

# Create the input value
p = tuple3d.Point(1.0, 2.0, 3.0)

# Test the input value
result = has_a_zero(p)

# Compare to expected output
assert_equals(False, result)

Procedure

# Create the input value
p = tuple3d.Point(1.0, 2.0, 3.0)

# Test the input value
cycle_left(p)

# Compare to expected output
assert_floats_equal(2.0, p.x)
assert_floats_equal(3.0, p.y)
assert_floats_equal(1.0, p.z)
## 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

#### Program Structure
```
def foo():
    print 'Hello'

# Application code
if __name__ == 'main':
    foo()
    foo()
    foo()
```

#### Program Flow
```
>>> python foo.py
'Hello'
'Hello'
'Hello'
```

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

#### Format

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

#### Example

```
# Put x in z if it is positive
if x > 0:
    z = x
```

### Execution:

if `<boolean-expression>` is true, then execute all of the statements indented directly underneath (until first non-indented 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 elsec.
Conditionals: “Control Flow” Statements

if \( b \):
    \( s1 \) # statement

\( s3 \)

if \( b \):
    \( s1 \)
else:
    \( s2 \)

\( s3 \)

Branch Point: Evaluate & Choose

Statement: Execute

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):

Frame sequence depends on flow
def max(x,y):
    """Returns: max of x, y"""
    # simple implementation
    1 if x > y:
    2     return x
    3 return y

Frame sequence depends on flow

max(0,3):

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

Skips line 2
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""
    # 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  1
  x  3  y  0
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  3
y  0
```

9/18/14
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
  x   3  y  0
  temp 3
**Program Flow vs. Local Variables**

```python
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 | 0 |
+---+---+---+---+
| temp | 3 |
```

9/18/14  Conditionals & Control Flow  16
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></th>
<th>max</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>temp</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
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 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
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'
```

Traces
# Watches vs. Traces

<table>
<thead>
<tr>
<th>Watch</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visualization tool</td>
<td>Visualization tool</td>
</tr>
<tr>
<td>(e.g. print statement)</td>
<td>(e.g. print 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 where flow can change</td>
</tr>
<tr>
<td>What you did in lab</td>
<td></td>
</tr>
</tbody>
</table>
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()'
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):
    """Returns: max of x, y""
    # swap x, y
    # put larger in temp
    temp = y
    if x > y:
        temp = x
    return temp
```

First assigned

Inside scope
Variation on max

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

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

#### Example

```
# 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

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

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

## Example

\[ \# \text{ Put max of } x, y \text{ in } z \]

\[ z = x \textbf{ if } x > y \textbf{ else } y \]

expression, not statement