Lecture 6

Control Structures
Conditionals: If-Statements

Format

```
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

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

### Example

```python
# 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: If-Elif-Else-Statements**

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
</table>
| **if** `<boolean-expression>`:  
  `<statement>`  
  ...  
  **elif** `<boolean-expression>`:  
  `<statement>`  
  ...  
  ...  
  **else**:  
  `<statement>`  
  ... | # 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

```
# Put max of x, y in z
z = x \textbf{if} \ x > y \ \textbf{else} \ y
```

expression, not statement
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

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

9/26/14  Control Structures
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

```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 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):
    # 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
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
    if x > y:
        temp = x
        x = y
        y = temp
    return temp
```

9/26/14
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
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
For Loops: Processing Sequences

The for-loop:

```
for x in seq:
    print x
```

• Key Concepts
  - loop sequence: `seq`
  - loop variable: `x`
  - body: `print x`
  - Also called repetend

• Remember:
  - Cannot program …
  - Reason for recursion
For Loops: Processing Sequences

The for-loop:

```python
for x in seq:
    print x
```

- **loop sequence**: `seq`
- **loop variable**: `x`
- **body**: `print x`

To execute the for-loop:

1. Check if there is a “next” element of **loop sequence**
2. If not, terminate execution
3. Otherwise, put the element in the **loop variable**
4. Execute all of **the body**
5. Repeat as long as 1 is true
Example: Summing the Elements of a List

def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers (either floats or ints)"
    # Create a variable to hold result (start at 0)
    # Add each list element to variable
    # Return the variable
Example: Summing the Elements of a List

def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)""
    result = 0
    for x in thelist:
        result = result + x
    return result

- loop sequence: thelist
- loop variable: x
- body: result = result + x
Example: Summing the Elements of a List

```python
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers (either floats or ints)"
    result = 0
    for x in thelist:
        result = result + x
    return result
```

Accumulator variable

- **loop sequence:** thelist
- **loop variable:** x
- **body:** result = result + x
def num_ints(thelist):
    """Returns: the number of ints in thelist
Precondition: thelist is a list of any mix of types""
    # Create a variable to hold result (start at 0)
    # for each element in the list...
    # check if it is an int
    # add 1 if it is
    # Return the variable
**For Loops and Conditionals**

```python
def num ints(thelist):
    """Returns: the number of ints in thelist
    Precondition: thelist is a list of any mix of types"""
    result = 0
    for x in thelist:
        if type(x) == int:
            result = result + 1
    return result
```

9/26/14  Control Structures
def add_one(thelist):
    """(Procedure) Adds 1 to every element in the list
    Precondition: thelist is a list of all numbers
    (either floats or ints)"

    for x in thelist:
        x = x+1

    # procedure; no return

DOS NOT WORK!
More Complex For-Loops

- Combine with a *counter*
  - Variable that increments each time body executed
  - Tracks position in seq

- Example:
  ```python
cnt = 0
for x in seq:
    print `x`+' at '+`cnt`
    cnt = cnt + 1 # incr
  ```

- Nest conditionals inside
  - Body is all indented code
  - Can put other control structures inside the body

- Example:
  ```python
nints = 0 # num of ints
for x in seq:
    if type(x) == int:
        nints = nints + 1
  ```
for-loops: Beyond Sequences

• Work on **iterable** objects
  - Object with an *ordered collection* of data
  - This includes sequences
  - But also much more

• **Examples:**
  - Text Files (built-in)
  - Web pages (urllib2)

• **2110:** learn to design custom iterable objects

```python
def blanklines(fname):
    """Return: # blank lines in file fname
    Precondition: fname is a string""
    # open makes a file object
    file = open('myfile.txt')
    # Accumulator
    count = 0
    for line in file:    # line is a string
        if len(line) == 0:    # line is blank
            count = count + 1
    f.close()    # close file when done
    return count
```

9/26/14   Control Structures   24
Beyond Sequences: The **while**-loop

```python
while <condition>:
    statement 1
    ...
    statement n
```

- **repetend** or **body**

- **Relationship to for-loop**
  - Broader notion of “still stuff to do”
  - Must explicitly ensure condition becomes false
while Versus for

```python
# process range b..c
for k in range(b,c+1):
    process k
```

```python
# process range b..c
k = b
while k <= c:
    process k
    k = k + 1
```

- Makes list $c+1-b$ elements
- List uses up memory
- Impractical for large ranges

- Just needs an int
- Much less memory usage
- Best for large ranges

Must remember to increment
while Versus for

# incr seq elements
for k in range(len(seq)):
    seq[k] = seq[k]+1

# incr seq elements
k = 0
while k < len(seq):
    seq[k] = seq[k]+1
    k = k+1

Makes a second list.

while is more flexible, but is much tricker to use
Case to Use while

- Want square root of $c$
  - Make poly $f(x) = x^2 - c$
  - Want root of the poly
    ($x$ such that $f(x)$ is 0)
- Use Newton’s Method
  - $x_0 = \text{GUESS} \ (c/2??)$
  - $x_{n+1} = x_n - f(x_n)/f'(x_n)$
    $= x_n - (x_n x_n - c)/(2x_n)$
    $= x_n - x_n/2 + c/2x_n$
  - Stop when $x_n$ good enough

```python

def sqrt(c):
    """Return: square root of c
    Uses Newton’s method
    Pre: c >= 0 (int or float)"
    x = c/2
    # Check for convergence
    while abs(x*x - c) > 1e-6:
        # Get $x_{n+1}$ from $x_n$
        x = x / 2 + c / (2*x)
    return x
```

9/26/14 Control Structures 28