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: If-Elif-Else-Statements

Format

```
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

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

```
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

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

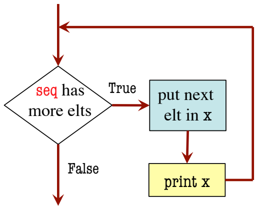
There are two **returns**!
But only one is executed

For Loops: Processing Sequences

The for-loop:

```
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)"""
    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

```
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

For Loops and Conditionals

```
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
```

Body

for-loops: Beyond Sequences

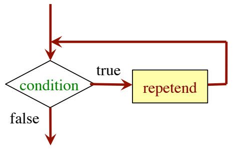
- Work on *iterable* objects
    - Object with an *ordered collection* of data
    - This includes sequences
    - But also much more
- **Examples**:
    - Text Files (built-in)
    - Web pages (urllib2)
- **2110**: learn to design custom iterable objects

```
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
```

Beyond Sequences: The while-loop

```
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

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

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

Must remember to increment

- 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

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 = GUESS (c/2??)
    - $x_{n+1} = x_n - f(x_n)/f'(x_n)$
      $= x_n - (x_n x_n - c)/(2x_n)$
      $= x_n - x_n/2 + c/2x_n$
      $= x_n/2 + c/2x_n$
    - Stop when x_n good enough

```
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
```