

### 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
```

### Important Concept in CS: Doing Things Repeatedly

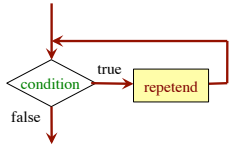
- Process each item in a sequence
  - Compute aggregate statistics for such as the mean, median, standard deviation
  - Send everyone in a Facebook group an appointment time
- Perform *n* trials or get *n* samples.
  - A4: draw a triangle six times to estimate the area
  - Run a protein-folding simulation
- Do something an unknown number of times
  - CUAUV team, vehicle keeps moving until reached its goal



### Beyond Sequences: The while-loop

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

**repeatend or body**



- Relationship to for-loop
  - Broader notion of “still stuff to do”
  - Must explicitly ensure condition becomes false
  - You explicitly manage what changes per iteration

### While-Loops and Flow

```
print 'Before while'
count = 0
i = 0
while i < 3:
    print 'Start loop '+str(i)
    count = count + i
    i = i + 1
    print 'End loop '
print 'After while'
```

Output:

```
Before while
Start loop 0
End loop
Start loop 1
End loop
Start loop 2
End loop
After while
```

### while Versus for

<pre># process range b..c-1 for k in range(b,c)     process k</pre> <p><b>Must remember to increment</b></p> <pre># process range b..c for k in range(b,c+1)     process k</pre>	<pre># process range b..c-1 k = b while k &lt; c:     process k     k = k+1</pre> <pre># process range b..c k = b while k &lt;= c:     process k     k = k+1</pre>
--	--

### Note on Ranges

- m..n* is a range containing *n+1-m* values
  - 2..5 contains 2, 3, 4, 5. Contains 5+1 - 2 = 4 values
  - 2..4 contains 2, 3, 4. Contains 4+1 - 2 = 3 values
  - 2..3 contains 2, 3. Contains 3+1 - 2 = 2 values
  - 2..2 contains 2. Contains 2+1 - 2 = 1 values
  - 2..1 contains ???
- The notation *m..n*, always implies that *m* <= *n*+1
  - So you can assume that even if we do not say it
  - If *m* = *n*+1, the range has 0 values

### while Versus for

<pre># incr seq elements for k in range(len(seq)):     seq[k] = seq[k]+1</pre>	<pre># incr seq elements k = 0 while k &lt; len(seq):     seq[k] = seq[k]+1     k = k+1</pre>
--	---

Makes a **second** list.

while is more flexible, but requires more code to use

### Patterns for Processing Integers

<p style="text-align: center;"><b>range a..b-1</b></p> <pre>i = a while i &lt;= b:     process integer I     i = i + 1</pre>	<p style="text-align: center;"><b>range c..d</b></p> <pre>i = c while i &lt;= d:     process integer I     i = i + 1</pre>
--	--

```
# store in count # of '/'s in String s
count = 0
i = 0
while i < len(s):
    if s[i] == '/':
        count = count + 1
    i = i + 1
# count is # of '/'s in s[0..s.length()-1]
```

```
# Store in double var. v the sum
# 1/1 + 1/2 + ... + 1/n
v = 0; # call this 1/0 for today
i = 0
while i <= n:
    v = v + 1.0 / i
    i = i + 1
# v = 1/1 + 1/2 + ... + 1/n
```

### while Versus for

<pre># table of squares to N seq = [] n = floor(sqrt(N)) + 1 for k in range(n):     seq.append(k*k)</pre>	<pre># table of squares to N seq = [] k = 0 while k*k &lt; N:     seq.append(k*k)     k = k+1</pre>
---	---

A for-loop requires that you know where to stop the loop **ahead of time**

A while loop can use complex expressions to check if the loop is done

### while Versus for

Fibonacci numbers:

$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

<pre># Table of n Fibonacci nums fib = [1, 1] for k in range(2,n):     fib.append(fib[-1] + fib[-2])</pre>	<pre># Table of n Fibonacci nums fib = [1, 1] while len(fib) &lt; n:     fib.append(fib[-1] + fib[-2])</pre>
--	--

Sometimes you do not use the loop variable at all

Do not need to have a loop variable if you don't need one

### Cases to Use while

Great for when you must **modify** the loop variable

<pre># Remove all 3's from list t i = 0 while i &lt; len(t):     # no 3's in t[0..i-1]     if t[i] == 3:         del t[i]     else:         i += 1</pre>	<pre># Remove all 3's from list t while 3 in t:     t.remove(3)</pre>
--	---

Stopping point keeps changing.

The stopping condition is not a numerical counter this time. Simplifies code a lot.

### Cases 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**
  - $x_0 = \text{GUESS } (c/2???)$
  - $x_{n+1} = x_n - f(x_n)/f'(x_n)$   
 $= x_n - (x_n^2 - c)/(2x_n)$   
 $= x_n - x_n/2 + c/2x_n$   
 $= x_n/2 + c/2x_n$
  - 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
```