Lecture 21

While Loops

Announcements for This Lecture

Assignments

- A4 grades better than hoped
  - **Mean**: 86.6 **Median**: 91
  - **Std Dev**: 13 points
  - **Passing Grade**: 35
- A5 grades as we wanted
  - **Mean**: 51.2 **Median**: 53
  - **Std Dev**: 5.5
  - **Passing Grade**: 30
- A6 due next Thursday
  - Dataset should be done

Prelim 2

- Thursday, 7:30-9pm
  - A–Sh (Statler Aud)
  - Si–X (Statler 196)
  - Y–Z (Statler 198)
  - SDS received e-mail
- **Make-up** is Friday
  - Only if submitted conflict
  - Also received e-mail
- Graded on Saturday

Recall: For Loops

```
# Print contents of seq
x = seq[0]
print x
x = seq[1]
print x
…
x = seq[len(seq)-1]
print x
```

The for-loop:

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

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

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

1. Process each item in a sequence
   - Compute aggregate statistics fo such as the mean, median, stan
   - Send everyone in a Facebook group an appointment time

```
for x in sequence:
    process x
```

2. Perform *n* trials or get *n* samples.
   - A4: draw a triangle six times to m
   - Run a protein-folding simulati

```
for x in range(n):
    do next thing
```

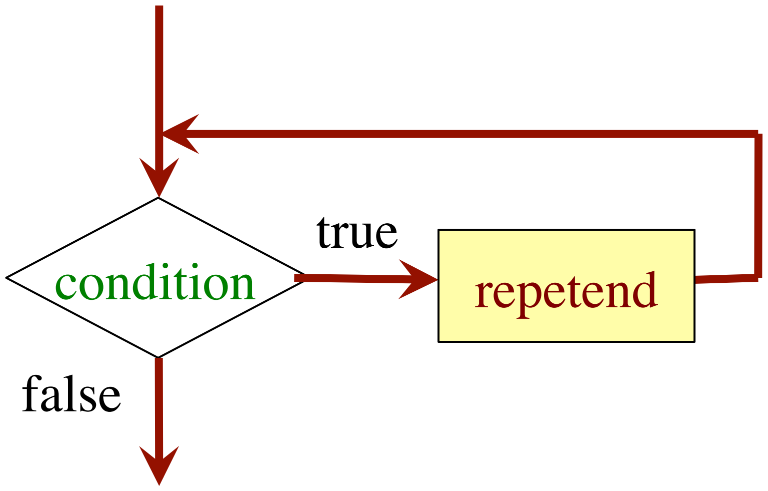
3. Do something an unknown number of times
   - CUAUV team, vehicle keeps moving until reached its goal

????

Beyond Sequences: The **while-loop**

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

repetend 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

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

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

Must remember to increment

```
# 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
```

Range Notation

- 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 ???	

What does 2..1 contain?

A: nothing
B: 2,1
C: 1
D: 2
E: something else

Range Notation

- 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

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

Makes a **second** list.

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

while is more flexible, but requires more code to use

Patterns for Processing Integers

range a..b-1

```
i = a
while i < b:
    process integer I
    i = i + 1
```

```
# 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]
```

range c..d

```
i= c
while i <= d:
    process integer I
    i= i + 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

```
# table of squares to N
seq = []
n = floor(sqrt(N)) + 1
for k in range(n):
    seq.append(k*k)
```

```
# table of squares to N
seq = []
k = 0
while k*k < N:
    seq.append(k*k)
    k = k+1
```

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:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

```
# Table of n Fibonacci nums
fib = [1, 1]
for k in range(2,n):
    fib.append(fib[-1] + fib[-2])
```

Sometimes you do not use the loop variable at all

```
# Table of n Fibonacci nums
fib = [1, 1]
while len(fib) < n:
    fib.append(fib[-1] + fib[-2])
```

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

```
# Remove all 3's from list t
i = 0
while i < len(t):
    # no 3’s in t[0..i–1]
    if t[i] == 3:
        del t[i]
    else:
        i += 1
```

```
# Remove all 3's from list t
while 3 in t:
    t.remove(3)
```

Cases to Use while

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

```
# Remove all 3's from list t
i = 0
while i < len(t):
    # no 3's in t[0..i–1]
    if t[i] == 3:
        del t[i]
    else:
        i += 1
```

Stopping point keeps changing.

```
# Remove all 3's from list t
while 3 in t:
    t.remove(3)
```

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 = 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
```

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 M**
  - *x*
  - *x*
    $= x_n - (x_n x_n - c)/(2x_n)$
    $= x_n - x_n/2 + c/2x_n$
    $= x_n/2 + c/2x_n$
  - Stop when x_n good enough

```
def sqrt(c):
    """Return: square root of c
    Uses Newton's method
    Pre                          oat)"""

                     onvergence
    while abs(x*x – c) > 1e-6:
        # Get x_{n+1} from x_n
        x = x / 2 + c / (2*x)

    return x
```

Part II of A6 uses a similar idea.

Recall Lab 9

```
Welcome to CS 1110 Blackjack.
Rules: Face cards are 10 points. Aces are 11 points.
      All other cards are at face value.

Your hand:
2 of Spades
10 of Clubs

Dealer's hand:
5 of Clubs

Type h for new card, s to stop:
```

Play until player **stops** or **busts**

Recall Lab 9

```
Welcome to CS 1110 Blackjack.
Rules: Face cards are 10 points. Aces are 11 points.
      All other cards are at face value.

Your hand:
2 of Spades
10 of Clubs

Dealer's hand:
5 of Clubs

Type h for new card, s to stop:
```

How do we design a complex while-loop like this one?

Play until player **stops** or **busts**

Some Important Terminology

- **assertion**: true-false statement placed in a program to *assert* that it is true at that point
  - Can either be a **comment**, or an **assert** command
- **invariant**: assertion supposed to "always" be true
  - If temporarily invalidated, must make it true again
  - **Example**: class invariants and class methods
- **loop invariant**: assertion supposed to be true before and after each iteration of the loop
- **iteration of a loop**: one execution of its body

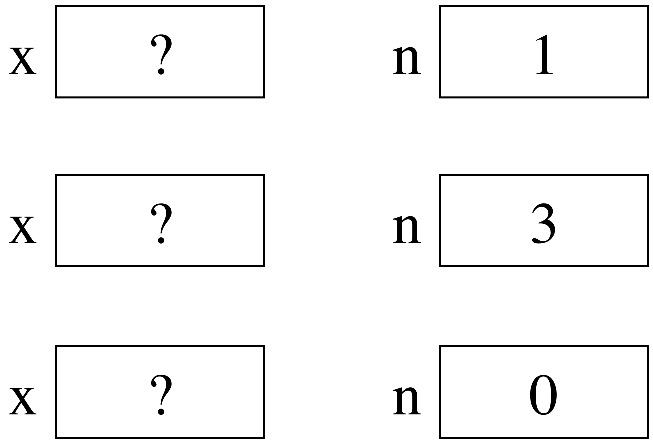
Assertions versus Asserts

- Assertions **prevent bugs**
  - Help you keep track of what you are doing
- Also **track down bugs**
  - Make it easier to check belief/code mismatches
- The assert statement is a (type of) assertion
  - One you are **enforcing**
  - Cannot always convert a comment to an assert

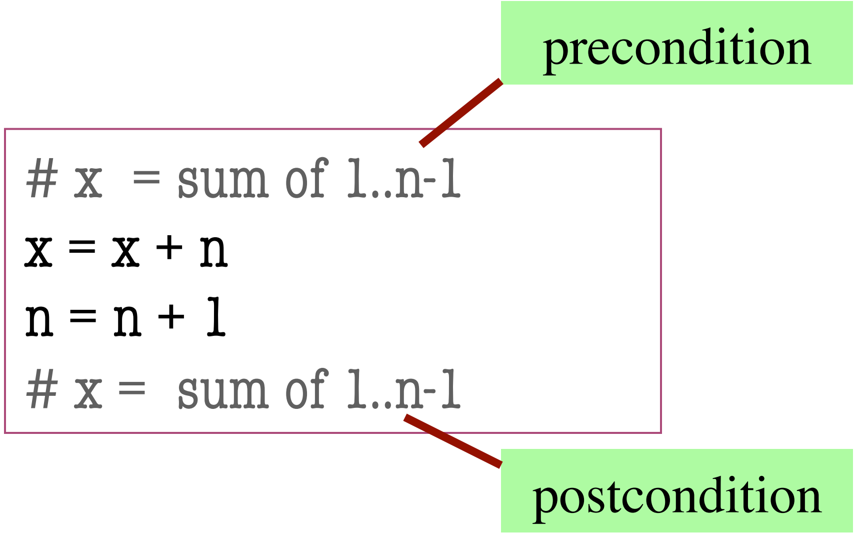
# x is the sum of 1..n

Comment form of the assertion.

The root of all bugs!



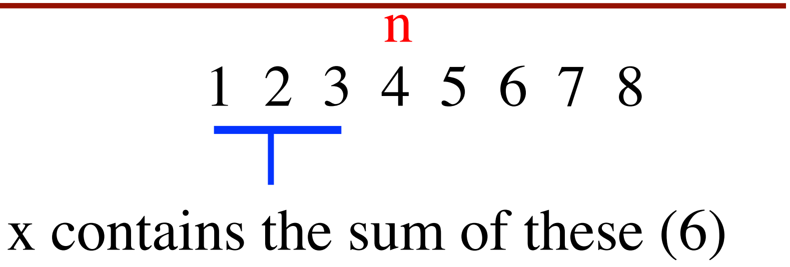
Preconditions & Postconditions



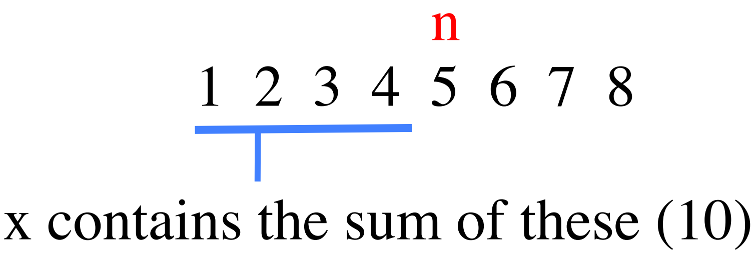
precondition

```
# x  = sum of 1..n-1
x = x + n
n = n + 1
# x =  sum of 1..n-1
```

postcondition



x contains the sum of these (6)



x contains the sum of these (10)

- **Precondition:** assertion placed before a segment
- **Postcondition:** assertion placed after a segment

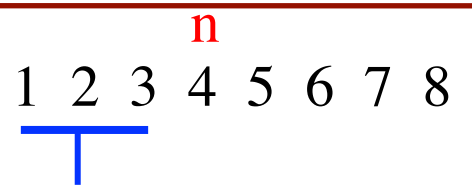
Relationship Between Two

If precondition is true, then postcondition will be true

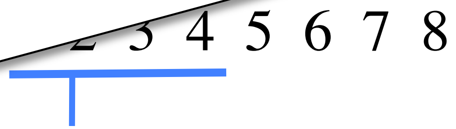
Preconditions & Postconditions

precondition

```
# x  = sum of 1..n-1
x = x + n
n = n + 1
# x =  sum of 1..n-1
```



x contains the sum of these (6)



x contains the sum of these (10)

Next Time: Using these to Design Loop

- **P** rtion pl efore a segment
- **Postcondition:** assertion placed after a segment

Relationship Between Two

If precondition is true, then postcondition will be true