Beyond Sequences: The while-loop

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

- 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 Versus for

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

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

Must remember to increment

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

Patterns for Processing Integers

```
range a..b-1
i = a
while i <= b:
    process integer i
    i = i + 1
```

```
range c..d
i = c
while i <= d:
    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]
```

While-Loops and Flow

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

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

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):
    if t[i] == 3:
        del t[i]
    else:
        i += 1

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

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.

Cases to Use while

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

def sqrt(c):
    return x

Using while-loops Instead of for-loops

Advantages

• Better for modifying data
  * More natural than range
  * Works better with deletion
• Better for convergent tasks
  * Loop until calculation done
  * Exact steps are unknown
• Easier to stop early
  * Just set loop var to False

Disadvantages

• Performance is slower
  * Python optimizes for-loops
  * Cannot optimize while
• Infinite loops more likely
  * Easy to forget loop vars
  * Or get stop condition wrong
• Debugging is harder
  * Will see why in later lectures