Lecture 20: while Loops
(Sections 7.3, 7.4)

CS 1110
Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Announcements
Recall: For Loops

for \( x \) in grades:
    print(x)

• **loop sequence:** grades
• **loop variable:** \( x \)
• **body:** `print(x)`

To execute the for-loop:
1. Check if there is a “next” element of loop sequence
2. If so:
   • assign next sequence element to loop variable
   • Execute all of the body
   • Go back to Step 1
3. If not, terminate execution
Different types of Repetition

1. Process each item in a sequence
   - Compute statistics for a dataset
   - Send all your contacts an email

2. Do something n times
   - Draw a checkers board
   - Run a protein-folding simulation for $10^6$ time steps

3. Do something an unknown number of times
   - Play word guessing game until 6 strikes
   - Go in current direction until edge is detected

https://www.flickr.com/photos/janitors/albums/72157642146435575/with/13058966193/
Beyond Sequences: The while-loop

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

Relationship to for-loop

- Broader notion of “keep working until done”
- Must explicitly ensure condition becomes false
- You explicitly manage what changes per iteration
import random

num = random.randint(0,10)
guessed_it = False
print("I'm thinking of a number.")

while not guessed_it:
    guess = int(input('Guess it: '))
    guessed_it = (num == guess)
print('Well done!')
Q: What gets printed?

```
a = 8
b = 12
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
print(a)
```

A: Infinite loop  
B: 8  
C: 12  
D: 4  
E: I don’t know

This is Euclid’s Algorithm for finding the greatest common factor of two positive integers.  

**Trivia**: It is one of the *oldest* recorded algorithms (~300 B.C.)
for vs. while

- You can almost always use either

- Sometimes **for** is better
  - Do something a **fixed** (pre-determined) number of times

- Sometimes **while** is better
  - Do something an **indefinite** (not infinite) number of times
  - E.g., do something until some event happens, i.e., until a stopping condition is reached
Task #1: do something n times

for k in range(n):    k = 0
    # do something  while k < n:
    # do something               k = k+1

Must remember to increment

My preference? for-loop
for vs. while

Task #2: do something an unknown number of times

for k in range(BIG_NUM):
    # do something
    if time to stop:
        break

while not time to stop:
    # do something

Do NOT use break in any work you submit in CS1110. Practice using while-loop in situations where while-loop is well suited

My preference? while-loop
Task #3: do something to each element of a sequence

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

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

while is more flexible, but sometimes requires more code

My preference? for-loop
Task #4: do something until a limit is reached

e.g., make a table of squares up to N

seq = [ ]
sqn = math.floor(sqrt(N))
for k in range(sqn+1):
    seq.append(k*k)

My preference? while-loop

can use complex expressions to check if a task is done

for-loop requires you to know how many iterations you want ahead of time

seq = [ ]
k = 0
while k*k < N:
    seq.append(k*k)
    k = k+1

for vs. while
Task #5: change a sequence’s length
e.g., remove all 3’s for list nums

for i in range(len(nums)):
    if nums[i] == 3:
        del nums[i]

IndexError: list index out of range

while 3 in nums:
    nums.remove(3)

is this not beautiful?

My preference? while-loop
Task #6: find 1\textsuperscript{st} n Fibonacci numbers

Fibonacci numbers:
\[ F_0 = 1 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2} \]

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

```
fib = [1, 1]
while len(fib) < n:
    fib.append(fib[-1] + fib[-2])
```

My preference?
No strong preference
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 variable (e.g., `keep_going`) to False

### Disadvantages

- **Infinite loops** happen more easily
  - Easy to forget loop vars
  - Or get continuation condition wrong
- **Require** more management
  - Initialize the condition?
  - Update the condition?
Setting up a **while-loop**

0. Situation is to do something until an event happens
1. Write the *continuation condition*
   - Create var names as necessary to express condition
   - May be easier to negate *stop* condition to get *continuation* condition
2. **Initialize loop vars (vars in loop condition)** as necessary
3. In loop body: *update loop vars*
   - to possibly change loop condition from True to False
4. Write the rest of the loop body
Improve number guessing game

```python
import random
min_num= 1
max_num= 10
max_chances= 5
secret_num= random.randint(min_num, max_num)
print("I have a number from "+str(min_num)+" to "+str(max_num))
print("You have "+str(max_chances)+" chances to guess it")
```

# User guesses until all chances used up or guessed correctly

1. Allow fixed number of guesses
   For you to add later:
2. If a guess is wrong, tell player whether it was too high or too low.
Setting up a while-loop

0. Situation is to do something until an event happens
1. Write the continuation condition
   - Create var names as necessary to express condition
   - May be easier to negate stop condition to get continuation condition
2. Initialize loop vars (vars in loop condition) as necessary
3. In loop body: update loop vars
   to possibly change loop condition from True to False
4. Write the rest of the loop body