LOOPS

LECTURE 04-1

JIM FIX, REED COLLEGE CSCI 121

UPCOMING COURSE EVENTS

- This coming Wednesday, 9/21, our first ***QUIZ***:
    - On Python scripting, conditional statements, and integer arithmetic.
    - 20 minutes; in-class; closed-note; written code.

LOOPS

- **Reading**: TP Ch 5, CP Ch 1.5
- A `while` statement can be used to repeat some code.
- The template below gives the syntax of a while loop statement:

```
lines of "set up" code to execute first
while condition-expression:
    lines of "loop body" code to execute if the condition holds
    ...
lines of "follow up" code to execute once the condition no longer holds
```

SIMPLE EXAMPLE

- This example script counts from 101 down to 1:

```
print(“This program will count down by 10.”)
count = 51
while count > 1:
    print(str(count) + ”...")
    count = count - 10
print(“1!!!!”)
```

- Output of the script above:

```
51...
41...
31...
21...
11...
1!!!!
```

- **NOTE:** hit [CTRL-c] to terminate the Python script's execution.

EXECUTION OF A WHILE LOOP

- The template below gives the syntax of a while loop statement:

```
lines of "set up" code to execute first
while condition-expression:
    lines of "loop body" code to execute if the condition holds
    ...
lines of "follow up" code to execute once the condition no longer holds
```

- Here is how Python executes this code:

1. Executes the **set up** code.
2. It evaluates the **condition**. If `False` it *skips* to **Step 5**.
3. Otherwise, if `True`, it evaluates the **loop body**'s code.
4. It goes back to **Step 2**.
5. It executes the **follow up**, and subsequent, code.

SOME LOOP ISSUES TO COVER

- The `while` template and what it means.
- Definite versus indefinite loops.
  - ➡ `countdown.py`, `guess.py`, `guess6.py`
- Infinite loops happen.
  - ➡ Hit [CTRL-c] to terminate a runaway script.
- Using boolean conditions to control loops.
- Using `break` and `continue`.
- Nested loops.

COUNTING DOWN, GENERALIZED, GIVING PAUSE

- This example script counts from 101 down to 1:

```
print("This program will count down to 1 by an amount.")
start = int(input("Enter a value to start near: "))
decrement = int(input("Enter an amount to step down: "))
#
print("Ready? Counting down to 1:")
input("[Hit RETURN]")
#
count = start - ((start - 1) % decrement)
while count > 1:
    #
    print(str(count) + "...", end='')
    sys.stdout.flush()
    time.sleep(1)
    #
    count = count - decrement
#
print("1!!!!!")
```

DEFINITE VS. INDEFINITE LOOPS

- Some terminology:
  - "*Count up to 6.*" and "*Count up to the input value.*" are examples of *definite* loops.
  - "*Get an input until they've entered something valid.*" is an example of an *indefinite* loop. The number of repetitions isn't known.

- An example of the second kind of coding:

```
def get_float(prompt):
    return float(input(prompt))

def get_area():
    a = get_float(“Circle area? “)
    while a < 0.0:
        a = get_float(“Not an area. Try again:”)
    return a
```

DEFINITE VS. INDEFINITE LOOPS

- Some terminology:
  - "*Count up to 6.*" and "*Count up to the input value.*" are examples of *definite* loops.
  - "*Get an input until they've entered something valid.*" is an example of an *indefinite* loop. The number of repetitions isn't known.

- An example of the second kind of coding:

```
def get_float(prompt):
    return float(input(prompt))

def get_area():
    a = get_float(“Circle area? “)
    while a < 0.0:
        a = get_float(“Not an area. Try again:”)
    return a
```

Note that the loop body might not run at all!

GUESSING GAME

- This example script engages the user in a guessing game:

```
number = random.randint(1,100)
print("I have chosen a random number from 1 to 100.")
print("Try and guess what it is.")

guess = int(input("Your guess? "))
while guess != number:
    if guess > number:
        print("That guess was too high!")
    else:
        print("That guess was too low!")
    guess = int(input("What's your next guess? "))

print("You got it right! Great job.")
```

NESTING CONTROL STATEMENTS WITHIN A LOOP

- Of course you can put a conditional statement within a loop's body.

```
count = 0
while count < 6:
    if count % 2 == 0:
        print(str(count) + ” is even.”)
    else:
        print(str(count) + ” is odd.”)
    count = count + 1
print(“Done.”)
```

- Output of the script above:

```
0 is even.
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
Done.
```

GUESSING GAME WITH 6 GUESSES

- This example script engages the user in a *more challenging* guessing game:

```
number = random.randint(1,100)
print("I have chosen a random number from 1 to 100.")
print("Try and guess what it is.")

guess = int(input("Your guess? "))
guesses = 1
while guesses < 6 and guess != number:
    if guess > number:
        print("That guess was too high!")
    else:
        print("That guess was too low!")
    guess = int(input("What's your next guess? "))
    guesses = guesses + 1

if guess == number:
    print("You got it right! Great job.")
else:
    print("Oh, so sorry. You ran out of guesses.")
    print("The number was "+str(number)+".")
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

- ***What does this do???***

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

Inner loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

Inner loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

Outer loop, along with set-up/follow-up

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

Executed once for each value of a.

Inner loop, along with set-up/follow-up

Outer loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

BREAKING OUT OF A LOOP

- Here is another way of writing the counting loop.

```
print(“Counting from 0 to 5:”)
count = 0
while True:
    if count >= 6:
        break
    print(count)
    count = count + 1
print(“Done.”)
```

- The code uses a `break` statement to jump down to the follow-up code.
- If within several loops, it jumps to just after the innermost one.
- This is an artificial example
- Using `break` statements can sometimes make code more readable than code that expresses all the "break out" or stopping conditions.

USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **break** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

- I worry that **break** can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **break** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

PLEASE use* break *sparingly, and with taste.

- I worry that **break** can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

USING RETURN WITHIN A LOOP

- This procedure uses **`return`** to exit its loop and the procedure:

```
def countUpTo(n)
    count = 1
    while True:
        if count > n:
            return
        print(count)
        count = count + 1
```

- The **`return`** statement breaks out of the loop and returns back to the place where **`countUpTo`** was called.

SUMMARY

- The while loop statement expresses **iterative** code.
  - Allows you to perform a series of actions *until* a condition holds.
  - The negation of this *terminating condition* is the loop's condition.
- It's possible for the code to loop forever. This is an ***infinite*** loop.
- Counting loops are common examples of ***definite*** loops.
- Loops that iterate an undetermined number of times are ***indefinite***.

SUMMARY (CONT'D)

- Loop bodies can contain other control statements:
  - For example, you can have `if` statements or other `while` statements.
  - If another loop statement is inside, then it is a ***nested loop***.
  - If a `break` statement, we can jump out of the loop mid-body.
  - If a `return` statement, we exit the loop *and* the function/procedure.

PROJECT 1: GAME OF LIFE AND IMAGE PROCESSING

- Posted on the web at jimfix.github.io/csci121/assign/project1.html
- It is a grid simulation.
- It is also an image processing platform.
- You'll write functions that compute a grid cell's value.
  - Based on its current value, from 0 to 100.
  - Based on its neighboring cell's values, also from 0 to 100.
- Applied successively over the entire grid, you obtain interesting behavior.

  (DEMO)
- Start looking at it!!! Play with the existing rule code.
- It's due ***Monday, October 3rd at 1pm***.

PROJECT 1 NEEDS TKINTER

- On some systems running Project 1 causes an error at the code line:

```
from tkinter import *
```

- This is the Python graphics library we use, and apparently isn't installed.

- For a Mac or a Windows machine :
  - Enter the Terminal command:

```
pip3 install tk
```

ADVANCED STUFF

- For those few using WSL on Windows:
  - Enter the terminal command:

```
sudo apt install python3-tk
```

  - Install a (free) tool called **MobaXterm**.
  - Run MobaXterm and create a *"New session..."* of type WSL.
  - Run the Grid program inside that terminal session