LOOP BREAK; LISTS

LECTURE 05–1

JIM FIX, REED COLLEGE CSCI 121

COURSE LOGISTICS AND ADMINISTRIVIA

- A checkpoint of Project 1 is due **Thursday**
  - Will post Gradescope sites for submitting `rules.py` and `demo.py`
  - Just want to see that you've completed three rules.
- **Quiz #1** on "hours:minutesXM" *back today*. People generally did well!

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.

A NEED FOR DATA STRUCTURES

- We're limited in our coding if we can store values *only using individual variables*.

- What if we want to process...
...a file full of data? ...a web site full of statistics? ...a collection of items?

- Suppose for example, a user enters in some arbitrary number of values...
  - ➡ With single variables, we can't name all of them.

- Languages provide ***data structures*** to hold collections of values.
  - Python has two built into the language:
  - ➡Python ***lists*** and Python ***dictionaries***.

OUR FIRST DATA STRUCTURE: PYTHON LISTS

- Python lets you represent sequences of data values:

```
>>> xs = [2,3,7,15,100]
>>> xs
[2, 3, 7, 15, 100]
>>> xs[3]
15
>>> xs[0]
2
>>> len(xs)
5
```

- This is a built-in data structure called a Python ***list***.
  - A list is a *sequence* of numbered slots; each slot stores a value.
  - Each slot can be accessed by its ***index***, starting at 0.
  - A list has a ***length***.
- A Python list is also our first explicit example of a Python (data) ***object***

MODIFYING A LIST'S CONTENTS

- A Python list is a ***mutable*** data structure.
  - ➡ This means that its contents can be changed.

```
>>> xs
[2, 3, 7, 15, 100]
>>> xs[3]
15
>>> xs[3] = 200
>>> xs[3]
200
>>> xs
[2, 3, 7, 200, 100]
>>> xs[0] = xs[2] + xs[4]
>>> xs
[107, 3, 7, 200, 100]
>>> xs[4] = 1000
>>> xs
[107, 3, 7, 200, 1000]
```

LIST INDEXING

- You have to be careful when accessing a list; need to be mindful of its length.

```
>>> xs = [2,3,7,15,100]
>>> xs
[2, 3, 7, 15, 100]
>>> xs[5]
error!
```

- Using a negative index allows you to access backward from the end of the list:

```
>>> xs[-1]
100
>>> xs[-2]
15
>>> xs[-5]
2
>>> xs[-6]
error!
```

EXAMPLE LIST FUNCTION

- This checks a list to see if its contents read the same backwards as forwards:

```
def is_palindrome(xs):
    hi = len(xs)-1
    lo = 0
    while hi > lo:
        if xs[lo] != xs[hi]:
            return False
        lo = lo + 1
        hi = hi - 1
    return True
```

EXAMPLE LIST FUNCTION

- This does the same using negative indexing

```
def is_palindrome(xs):
    index = 0
    middle = len(xs) // 2
    while index < middle
        if xs[index] != xs[-(index+1)]:
            return False
        index = index + 1
    return True
```

EXAMPLE LIST FUNCTION

- This checks to see if the contents of two lists are the same:

```
def same_contents(xs,ys):
    if len(xs) != len(ys):
        return False
    i = 0
    while i < len(xs):
        if xs[i] != ys[i]:
            return False
        i = i + 1
    return True
```

EXAMPLE LIST FUNCTION

- This checks to see if the value **`y`** is stored in any of the slots of the list **`xs`**:

```
def contains(y,xs):
    i = 0
    while i < len(xs):
        if xs[i] == y:
            return True
        i = i + 1
    return False
```

LIST CONTENT CHECKS

- Python has `contains` and `same_contents` built into its language:

```
>>> 4 in [1,2,4,8] # Does the list contain an element?
True
>>> 7 in [1,2,4,8]
False
>>> xs = [1,3,4]
>>> ys = [1,3,5]
>>> xs == ys          # Are the lists' contents the same?
False
>>> xs != ys
True
>>> ys[2] = 4
>>> xs == ys
True
>>> xs != ys
False
>>> xs is ys          # Are they the same list object?
False
```

- The operators `in` and `==` check *contents*. The operator `is` checks list ***identity***.

MODIFYING LISTS: ADDING AND INSERTING

- We can add more slots to a list object:

```
>>> xs = [13,5,71]
>>> xs
[13, 5, 71]
>>> xs.append(-57)          # Adds a new slot to the end.
>>> xs
[13, 5, 71, -57]
>>> xs.extend([7,8,9])      # Adds several slots to the end.
>>> xs
[13, 5, 71, -57, 7, 8, 9]
>>> xs.insert(2,100)        # Adds a slot in the middle.
>>> xs
[13, 5, 100, 71, -57, 7, 8, 9]
```

MODIFYING LISTS: REMOVING

- We can remove slots from a list object:

```
>>> xs
[13, 5, 100, 71, -57, 7, 8, 9]
>>> xs.pop()     # Remove the last slot; return its value.
9
>>> xs
[13, 5, 100, 71, -57, 7, 8]
>>> xs[2]
100
>>> del xs[2]    # Remove a slot at a certain index.
>>> xs
[13, 5, 71, -57, 7, 8]
>>> xs[2]        # The other items shift left.
71
```

EXAMPLE LIST FUNCTION

- This function builds a list of integers:

```
def count_up(n):
    i = 1
    counts = []
    while i <= n:
        counts.append(i)
        i = i + 1
    return counts

>>> count_up(7)
[1, 2, 3, 4, 5, 6, 7]
```

EXAMPLE LIST FUNCTION

- This function builds a number's divisor sequence:

```
def divisor_list(number):
    sequence = [1]
    divisor = 2
    while divisor < number:
        if number % divisor == 0:
            sequence.append(divisor)
        sequence.append(number)
        return sequence

>>> divisor_list(35)
[1, 5, 7, 35]
>>> divisor_list(1)
[1]
>>> divisor_list(7)
[1, 7]
>>> divisor_list(36)
[1, 2, 3, 4, 6, 9, 12, 18, 26]
```

EXAMPLE LIST PROCEDURE

- This function modifies a list.

```
def rotate_right(xs):
    if len(xs) > 1:
        last = xs.pop()
        xs.insert(0,last)
```

- Calling `rotate_right` has the ***side effect*** of changing the list you give it:

```
>>> dsForSixteen = divisors_list(16)
>>> dsForSixteen
[1, 2, 4, 8, 16]
>>> rotate_right(csForSix)
>>> csForSix
[16, 1, 2, 4, 8]
>>> rotate_right(csForSix)
>>> csForSix
[8, 16, 1, 2, 4]
```

PYTHON LIST SUMMARY

- List creation via enumeration, concatenation, repetition, slicing:

```
[3,1,7]   []   [1,2]+[3,4,5]
```

- Accessing contents by index; list length:

```
xs[3]   xs[-1]   len(xs)
```

- Updating contents by indexed assignment:

```
xs[3] = 5
```

- Modifying/mutating a list object:

```
xs.append(5)    xs.extend([8,9,10])    xs.insert(2,357)
xs.pop()        del xs[6]
```

- Checking membership, content equality, object identity:

```
3 in xs      xs == [1,2,3]     xs is ys
```

- Scan according to index using a **`while`** loop:

```
i = 0
while i < len(xs):
    print(xs[i])
    i = i + 1
```