Lecture 07 topics

• using logical conditions to repeat a task multiple times

• from branching & logical conditions to iterations

• intro to recursion: functions that call themselves?!?

• → using while loops for iterations ←
Review:
Boolean Conditions and Comparison Operators

A **Boolean condition** is anything that is either **True** or **False**.

By using **comparison operators**, we can obtain truth values when comparing one value to another. Here are some comparison operators that will yield either True or False when used:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equals</td>
<td>5 == 6</td>
<td>False</td>
</tr>
<tr>
<td>!=</td>
<td>Not Equal</td>
<td>5 != 6</td>
<td>True</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
<td>5 &gt; 6</td>
<td>False</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal</td>
<td>5 &gt;= 6</td>
<td>False</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
<td>5 &lt; 6</td>
<td>True</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal</td>
<td>5 &lt;= 6</td>
<td>True</td>
</tr>
</tbody>
</table>

Warning! It’s very common to make a mistake here and confuse = (assignment) with == (equality comparison) ....they mean very different things....
Review – Branching example: the Voting Program

# we've seen this simple voting program before,
# it uses the if-elif-else statement with comparison operators

vote = input("Please vote for Candidate A, B, or C (enter A, B, or C): ")

if vote == 'A' :
    print ("Candidate A thanks you for your support."")
elif vote == 'B' :
    print ("Candidate B pledges to make things better!"")
elif vote == 'C' :
    print ("Candidate C promises to stop lying!"")
else : 
    print ("Your write-in vote has been discarded.")

print ("Thanks for voting!")
Review: the **if-elif-else** branching structure

The general structure stops at the first "True" match.
Review: the if-elif-else branching structure

one (and only one) of the four possible choices will be True, and only the corresponding statements will be evaluated.

note: in this sample code, statements are missing in the first three cases:
if you try and run this code unmodified, you'll get an error!
(a comment doesn't count as a statement)
you'll need to write at least something (such as the print statement inside the fourth block) inside each one of the four possible blocks.
If you don't want Python to do anything in one of these cases, you may use the pass statement, which ...doesn't do anything!
and now?
(what if we have to repeat something many times?)
Repetition example: the *Guess My Number* Game

Sample run of the *Guess My Number* game

This program *repeatedly* asks the user to guess a number between 1 and 100.

I.e. this program needs to *repeatedly* perform some action, an *undefined* number of times: how many times will it take for the player to guess the number? You don't know that in advance....

Can we do this with what we know so far in Python, i.e. by using only conditions, *if*-statement branching, and functions?
A way to achieve repetition: Recursive Functions

Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.

- For example, here is the task named "find your way home":

"Find your way home" instructions:

1. If you are at home, stop moving.
2. Take one step toward home.
3. Execute the instructions "Find your way home".

The above solution to finding your way home takes two steps (or three steps). 
First, we don't go home if we are already home. (this is where the action STOPs) 
Secondly, we do a very simple action that makes our situation simpler to solve. 
Finally, we redo the entire procedure. 

(for more, see reference [1] on last slide)
A way to achieve repetition: Recursive Functions

**Recursive functions** are ones that, in their definitions, call themselves.

(that was step 3. in the "Find your way home" instructions on the previous slide)

**Another way to say what recursion is:**
"*define a problem in terms of (a simpler version of) itself*

Typically, recursion is used when a **larger problem** can be broken down into **subproblems**, each of which is smaller than the last one.

(that was step 2. in the "Find your way home" instructions on the previous slide)

You stop when you reach the smallest step called the **basis**.

(that was step 1. in the "Find your way home" instructions on the previous slide)
A Recursive Function Example

Here's an example of a mathematical recursive function:

**Basis:** \( f(0) = 3 \)  
that's for 0

**Recursive step:** \( f(n) = f(n-1) + 3 \)  
that's for \( n > 0 \)

Let's try a few examples of using the above function \( f() \):

- **How much is \( f(1) \)?**
  \[ f(1) = 6, \] because \( f(1) = f(0) + 3 \) and \( f(0) = 3 \)
  (we can use \( f(0) \) directly because it's the basis, or base case for the function \( f() \))

- **How much is \( f(2) \)?**
  \[ f(2) = 9, \] because \( f(2) = f(1) + 3 \) and \( f(1) = 6 \)
  (we can use \( f(1) \) directly because we had just computed its value above)
A Recursive Function in Python

Let's take this example of a *mathematical* recursive function:

**Basis:** \( f(0) = 3 \) that's for 0

**Recursive step:** \( f(n) = f(n-1) + 3 \) that's for \( n > 0 \)

Let's translate this *mathematical* recursive function into an *algorithm* – i.e. the pseudocode for a program (not yet Python code):

**Define a function** \( f() \) **that takes one number:**

Is the number \( 0 \)?
   If so, return 3, since \( f(0) = 3 \)
Otherwise:
   Return 3 + \( f(\text{of the previous number}) \)
A Recursive Function in Python

Define a function f() that takes one number:

- Is the number 0?
  - If so, return 3, since f(0) = 3
- Otherwise:
  - Return 3 + f(of the previous number)

Finally, let's translate the pseudocode into a Python function:

```python
def recursiveFunction(number):
    if (number == 0):  # basis
        return 3
    else:              # recursive step
        return 3 + recursiveFunction(number - 1)

# in the above line, this function calls itself!
# or rather, it calls itself on a
# slightly smaller problem, i.e. number - 1

# let's test this function a few times:
print( recursiveFunction(0) )
print( recursiveFunction(1) )
print( recursiveFunction(2) )
print( recursiveFunction(3) )
print( recursiveFunction(100) )
```
is recursion the only way to repeat?
achieving Repetition with **Iterations**

How can we have our program do the same thing over and over, without using recursion?

Some examples:

- how do we compute something that takes a large number of identical steps that need to be repeated?
- or, how can we continue to accept input, while displaying a message and asking for more user input?
- etc.

To implement such repetitions, we can use a type of loop known as the **while loop**!

**while** loops continue to execute **repeatedly**, until a Boolean condition becomes **False**.
while loops

example:

```python
myNum = float(input("enter a small number "))

# while Condition :
while (myNum >= 100) :
    # whatever is in the "while block"
    # is executed as long as the Condition above is True
    print("sorry, too large")
    myNum = float(input("enter another number "))

# whenever the Condition above becomes False,
# code execution continues here...
```
**while loops**

example:

```python
myCounter = 0
while (myCounter < myNum):
    print(" ", myCounter, end="")
myCounter = myCounter + 1
```
The **while** loop (continued)

*Sentry variable / loop control variable:*  
the **variable** used in loop condition  
`myCounter` in the previous slide

**Loop body:**  
Block associated with loop.  
In the previous slide, it was:

```
print (" ", myCounter, end="")
myCounter = myCounter + 1
```
The **while** loop (continued)

**Infinite loop:**

A loop that will never end: it's considered a logical error.

A type of infinite loop where

the sentry variable is not there (it's a constant)
or it's never updated is easy to track down:

```python
while True:
    print(" an infinite loop ")
```

If you get an infinite loop:

Don’t panic – just restart IDLE (select the menu *Shell ➔ Restart Shell*)

But there are more insidious forms of the never-ending loop…
The **while** loop (in the *Guess My Number* Game)

Now let's look at the construct that we had ignored last time: the **while** statement.

The **while** loop construct:
- allows you to **repeat** a block of code (the block is enclosed in a blue rectangle here)
- this **repetition** is based on a **condition**: the loop will keep repeating the block as long as the condition stays **true**.
using a **while** loop to simulate a conversation...

Sample run of the Candidate Simulator program.

%(the "Dawson" textbook, as from reference [3], also presents a "Three Year Old" simulator. It's up to you to decide which one provides a more accurate simulation of a real-life situation!)
using a **while** loop to simulate a conversation...

Source code of the *Candidate Simulator* program:
Please refer to the "Dawson" book for the source code for the "Three Year Old" simulator. It's up to you to decide which one provides a more accurate simulation of a real-life situation :-) 

```python
# a candidate's simulator
#
# this program demonstrates the while loop

print("\tWelcome to 'Simulating a Candidate'\n")
print("This program simulates a conversation with a candidate.\")
print("Try to stop the madness.\n")

response = ""
while response.lower() !="yes":
    response = input("Will you vote for me?\n")

print("Thank you!")

input("\n\nPress the enter key to exit." )
```

- **the while** statement allows you to repeat section of code as long as some condition is True
- Like *if* statement, in that it tests a condition and executes associate block if condition True
- But, after executing the block, repeats condition test; if condition still True, repeats block
- Continues process until condition tests False
The **while** loop — some definitions:

**Sentry variable:**
Variable used in loop condition `response`

**Loop body:**
Block associated with loop, e.g.:
```python
response = input("Will you vote for me?\n")
```

**Infinite loop:**
A loop that will never end;
should be considered a logical error
Avoiding Infinite Loops

one kind of infinite loop may be a loop where the sentry variable is never updated – that kind of infinite loop is easy to track down...

...but there are more insidious forms of the never-ending loops!

for example, see next page
The Losing Battle Program

Sample run of the Losing Battle program (as from reference [3] on last slide) ...this is an example of an infinite loop!
The Losing Battle Program

# Losing Battle
# Demonstrates the dreaded infinite loop

print("Your lone hero is surrounded by a massive army of trolls.")
print("Their decaying green bodies stretch out, melting into the horizon.")
print("Your hero unsheathes his sword for the last fight of his life.
"")

health = 10
 trolls = 0
 damage = 3

while health != 0:
    trolls += 1
    health -= damage

    print("Your hero swings and defeats an evil troll, " \
    "but takes", damage, "damage points."")

print("Your hero fought valiantly and defeated", trolls, "trolls.")
print("But alas, your hero is no more.")

Source code of the Losing Battle program (example of an infinite loop) as from reference [3]
Fixing an Infinite Loop

```python
while health != 0:
    trolls += 1
    health -= damage
```

The issue here is: the condition is False only when `health` is exactly 0.

To find the issue and eventually fix it, we can use a debugging technique called **Tracing**.

**Tracing** is:
Examining the execution of a program (and its internal values) in single steps.

by Tracing we find out that `health` becomes negative, but never exactly 0.

Now that we know do we fix the issue in?
That's part of your homework this week!
References

1. "Recursion" by H. James de St. Germain
   https://www.cs.utah.edu/~germain/PPS/Topics/recursion.html

2. "Recursion" from "Learning to Program" by Alan Gauld

   http://programgames.com/page4.html