Lecture 12

`while` Loops; Common `while` Loop Tasks and Exceptions; Newton's Method; Flag-Controlled Loops

1. `while` Loops

The loops we have discussed so far are sometimes called **definite loops** because they run a fixed number of times: once for every element in a list or range, where the list is typically fixed before you first encounter the loop.

Not every loop is so clear cut. For example, consider a program that asks you to put in a password over and over again until you get it right. How many times will the loop run? You don't know until you get to the last pass through the loop.

In [ ]:
```
# EXAMPLE 1a: Password

correct_password = "Hamburger"

user_guess = input("Guess the password: ")

number_of_guesses = 1

while user_guess != correct_password:
    print("Wrong!")
    user_guess = input("Guess again: ")
    number_of_guesses += 1

print("Took you long enough: {0} guesses.".format(number_of_guesses))
```

This type of problem -- when you want code to run over and over again not till you get to the end of some fixed list, but rather until some specific even happens -- is what `while` loops are for.

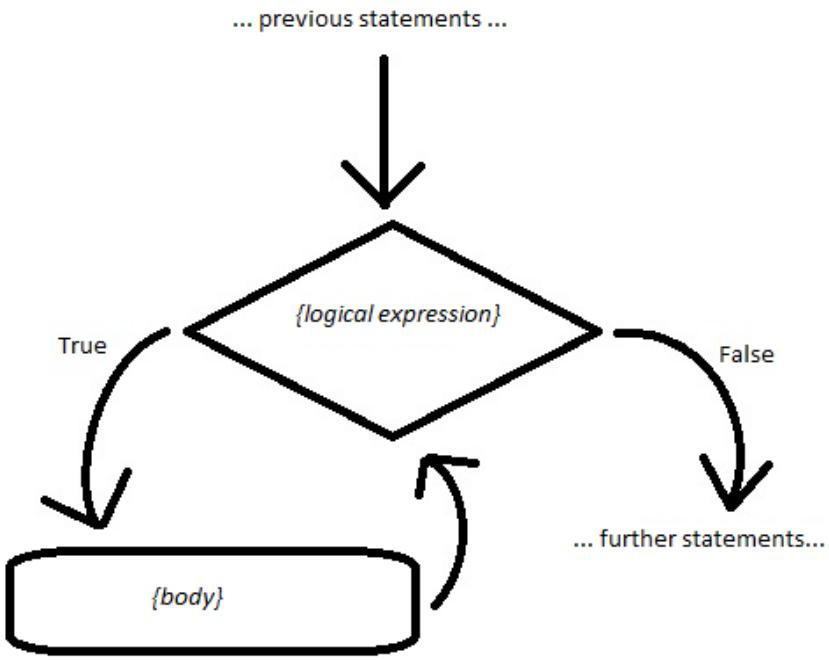
In [ ]:
```
WHILE LOOP SYNTAX:

"... previous statements ..."
while <logical expression>:
    <body, indented>
"... further statements, unindented..."
```

This code will do the following.

- First, `<logical expression>` will evaluate.
- If it evaluates to `True`, <body> will execute.
- Once that is finished, `<logical expression>` will evaluate again...
- ...and if it is `True`, <body> will execute again, with all pertinent variables using their latest values.
- And so on until `<logical expression>` evaluates to `False`. At this point, statements after the <body> are executed.

As always, indenting is important: the part of the code that gets repeated is everything after the `while` line, until you get to a non-indented line.

Here's a flowchart:



In [ ]:
```
# EXAMPLE 1b: Walkthrough
# Can you follow what is happening here?
# Once you know, taking away the quotations at the front.

"""
x = 3
y = 2


while x <  2 * y:
    x += y
    if x < 10:
        y += 1
    print(x,y)

print(y)
```

You might notice that a `while` loop can do anything a `for` loop can, and more! Indeed: **`while` loops are more general than `for` loops.** (If that's not obvious, we'll see some examples in a short while.) And `while` loops are simpler syntactically. Why not teach them first?

- If you can do a straightforward taks with a `while` or a `for`, the `for` loop is usually shorter and easier to read.
- Most problems that truly need a `while` loop are more complex than the one's we've talked about so far.

Here's a problem that doesn't need a `while` loop, but we'll do it with one anyway.

In [ ]:
```
# EXAMPLE 1c: 99 Bottles
# What does this code do?  What should it do?  Can you fix it, still using a while
loop?

var = 1

while var <= 99:
    print("{0} bottles of beer on the wall, {0} bottles of beer".format(var))
    print("If one of those bottles should happen to fall, {0} bottles of beer on th
e wall".format(var-1))
    var = var + 1
```

In [ ]:
```
# EXAMPLE 1c': 99 Bottles
# What does this code do?  What should it do?  Can you fix it, still using a while
loop?

var = 99

while var >= 1:
    print("{0} bottles of beer on the wall, {0} bottles of beer".format(var))
    print("If one of those bottles should happen to fall, {0} bottles of beer on th
e wall".format(var-1))
    var = var - 1
```

As I said, everything that can be done with a `for` can be done with a `while`, as well. The basic trick is:

- Initialize `loop_count` to be `0` or `1`
- Write a line like `while loop_count < len(list_name):`, or `while count <=  <number of times>:`
- At the end of the body of the loop, insert a line incrementing `loop_count`, like `loop_count += 1`

To be clear: ***in Python, this way of going through a list is frowned upon.*** It is not a good idea to write like this in Python -- Python is optimized for `for` loops, and Python coders will expect them in a situation like this. But I'm going to make you learn how to *read* it -- maybe even write it -- anyway, because this type of thing is necessary in other languages, and because it is a bit of a brain exercise.

So, for example, how would you compute 100 terms of

$$e^3 + e^5 + e^7 + e^9 + e^{11} + \dots$$

with a `for` loop? With a `while` loop?

In [ ]:
```
# EXAMPLE 1d: 50 terms of e^3 + e^5 + e^7 + e^9 + e^11 + ...
import math

# FOR loop:
rs = 0
for term_n in range(1,51):
    the_exponent = 2*term_n + 1
    rs += math.exp(the_exponent)
print(rs)

# WHILE loop:
rs = 0
term_n = 1     # FIRST difference
while term_n <= 50:   # SECOND difference
    the_exponent = 2*term_n + 1
    rs += math.exp(the_exponent)
    term_n += 1       # THIRD difference

print(rs)
```

2. Common `while` Loop Tasks, and a very brief Intro to Exception Handling

Input validation

We kind of saw this before, but one use for `while` loops is input validation -- if a program requires user input, you can make sure that it is valid.

In [ ]:
```
# EXAMPLE 2a: Input Validation

user_entry = float(input("Enter a score between 0 and 100: "))

# Get the first attempt OUTSIDE of the loop, so that you have something to check th
e first time you approach
# the while loop.
while 0 > user_entry or user_entry > 100:
    user_entry = float(input("Enter a score between 0 and 100: "))

print(user_entry)
```

What about checking whether the input is even a valid number? I'm only going to answer this because it's such an obvious question, but I almost would rather not -- it's a bit harder than it seems, and kind of a distraction.

The problem is that the line

```
user_entry = float(input("Enter a score between 0 and 100: "))
```

will cause the program to stop with a run-time error if you enter something that is not convertible to a `float`.

And if, instead, you tried something like

```
user_entry = input("Enter a score between 0 and 100: ")
if type(user_entry) == float:
```

...well, even if I input `1.23`, the value of `user_entry` will still be a `str` at the time you hit that line. The **real** question you want to ask is not "is `user_entry` a `float`?" (the answer to that is "No"), but rather, "are the contents of the string `user_entry` *convertible* to a `float` -- will the `float()` function even work??"

One way to answer that question is a careful analysis of the characters present in `user_entry`. A second way uses ***exception handling***, which are a mechanism for dealing with potential run-time errors. Let's look at the second way.

In [ ]:
```
# EXAMPLE 2b: Input Validation, Part 2

valid = False

while not valid:
    user_entry = input("Enter a score between 0 and 100: ")
    try:
        user_entry = float(user_entry)
        if 0 <= user_entry and user_entry <= 100:
            valid = True
    except ValueError:
        print("Number please!")


print(user_entry)
```

`try` and `except` go together, kind of like `if` and `else` (but `except` isn't optional). The code inside a `try` block executes -- and if executes successfully, that's that, and the `except` block is ignored. But if there is a run-time error in the `try` block, instead of Python just stopping, it moves on to the `except` block, and if the error is of the appropriate type, it performs that code (so that block is sorrrrrtttt of like an `elif` block). There's a lot to say here, but I'd rather move on.

Saving indefinite streams of entry (Sentinel-controlled loops)

What if you want the user to enter a bunch of data whose length isn't known ahead of time? One strategy is to use a ***sentinel***: a special value that can be entered that is interpreted not as data, but as a stop signal. For example, in the following example, the `str` value `"DONE"` is the sentinel.

In [ ]:
```
# EXAMPLE 2c: Sentinel

print("Input numbers until you are bored.  When you are finished, enter DONE.")

number_list = []

# First entry outside the loop, so that first loop test has something to look at.
entry = input("Gimme a number: ")

while entry != "DONE":
    entry = int(entry)
    number_list.append(entry)

    # Get next entry
    entry = input("Gimme a number: ")

print(number_list)
```

Create a list that adds up number supplied by the user, until the user inputs the number `-9`. The number `-9` itself should not be added, it should simply be the "stop!" signal.

In [ ]:
```
# EXAMPLE 2d: Another Sentinel

print("Input numbers until you are bored.  When you are finished, enter -9.")

# First entry outside the loop, so that first loop test has something to look at.
entry = int(input("Gimme a number: "))

thesum = 0

while entry != -9:
    thesum += entry
    # Get next entry
    entry = int(input("Gimme a number: "))


print(thesum)
```

3. Newton's Method

Problem: solve the equation $f(x) = 0$ for x.

Newton's method attacks this problem using tangent lines.

To explain the idea, here's a warmup:

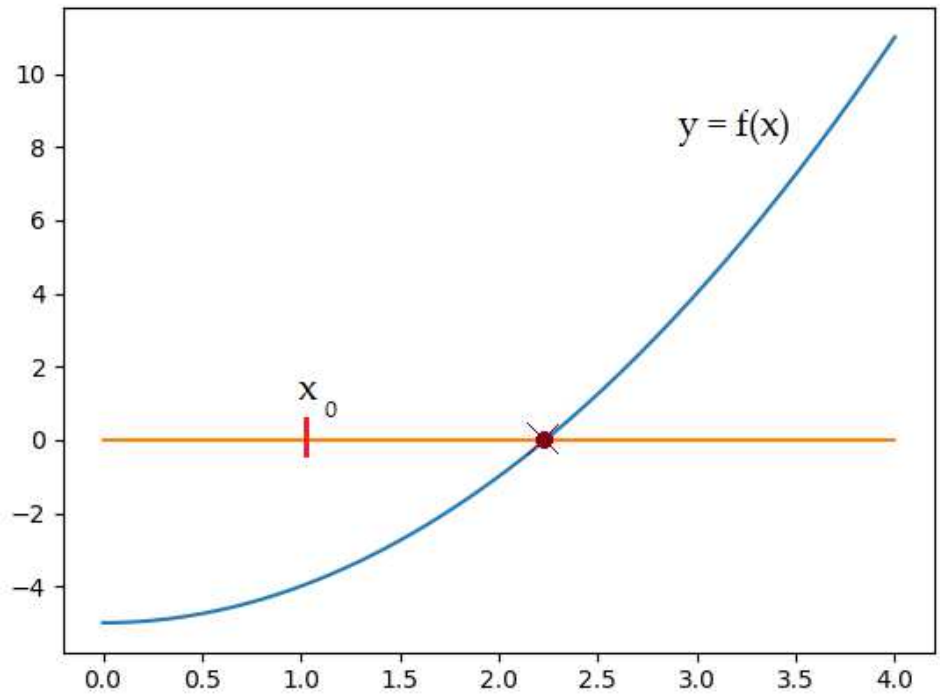
$g(x) = x^2$

$h(x) = 6x - 9$

When $x = 3$, what is $g(x)$? $h(x)$? How about when $x = 3.1$? Or when $x = 3.01$?

That each pair is close isn't an accident. $y = 6x - 9$ is the equation of the tangent line to $y = x^2$ at the point $(3, 9)$; or, otherwise put, $h(x)$ is the *linearization* of $g(x)$ at $x = 3$. This means that, at least when $x \approx 3$, $h(x)$ is a good replacement for $g(x)$. Furthermore, the linear function $h(x)$ is easier to work with, and solve equations with. This is the idea behind Newton's method.

Let's work with a concrete example: $f(x) = x^2 - 5$. Here's a graph of this function:



We want to solve $x^2 - 5 = 0$, which is the same as finding the x-intercept of the graph. We of course have simple ways to do this example, but it is one of the easiest examples to illustrate Newton's method with. The method works almost identically with harder examples.

The principle is: pick a random point x_0, linearize $f(x)$ at $x = x_0$, solve the easier linearized equation to get x_1, which is hopefully closer to a real solution. Then, repeat!

So, to start, let $x_0 = 1$. This is our more-or-less random first guess; I chose it mostly because I knew it was an easy number to start with, and because it's slightly close to the true solution.

Now, let's find the tangent line to $y = x^2 - 5$ at the point $x = 1$; that would be $y = 2x - 6$.

4. Flag-Controlled Loops

Recall that a *flag* is a fancy name for a `True`-`False` (or "yes-no") variable. Sometimes, the easiest way to control a loop is using a flag, something like:

```
while keep_going == True:
```

where `keep_going` is a `bool` variable. This variable will start out as `True`, but perhaps somewhere along the way something will happen that signals that the program should stop repeating -- at that moment, `keep_going` will be reset to be `False`.

(BTW: instead of writing `while keep_going == True:`, you could just write

```
while keep_going:
```

because `keep_going == True` is just going to be evaluated, producing the same value as `keep_going` itself.)

Here's an example: a famous guessing game.

In [ ]:
```
# EXAMPLE 4a: Guess the number

import random

# Choose the random number
secret_number = random.randrange(1,101)

# This holds the answer to the question "Should we keep guessing?"
# In the beginning, we certainly should
keep_guessing = True

print("I'm thinking of a number between 1 and 100! Can you guess it?")

while keep_guessing:
    guess = int(input("Enter a guess: "))
    if guess > secret_number:
        print("Too high!")
    elif guess < secret_number:
        print("Too low!")
    else:
        # Must be equal! Now is the time for the program to stop.
        print("You got it!")
        keep_guessing = False
```