Writing code that makes decisions: 
**for** and **while** loops

Lesson 3 – 9/13/16

With notes!
Today's topics

1. Intro to loops
2. for loops
3. while loops
4. Application of loops: file reading
1. Intro to loops
What is a loop?

- Loops simply let you execute a single piece of code multiple times
- For example, if you wanted to generate 10 random numbers: instead of copying and pasting `random.randint(0, 1)` ten times, you can simply put it in a loop that is set to loop ten times.

Sorry for that annoying problem set problem last time, but I wanted you to really feel the need for loops! 😊
Example

Instead of:

```
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
```

You can write:

```
for i in range(10):
    print random.randint(0,1)
```

Or:

```
count = 0
while count < 10:
    print random.randint(0,1)
    count = count + 1
```
First we'll talk about for loops.

Example

Instead of:

```python
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
```

You can write:

```python
for i in range(10):
    print random.randint(0,1)
```

Or:

```python
count = 0
while count < 10:
    print random.randint(0,1)
    count = count + 1
```
2. for loops
The **for** loop

**Purpose:** execute a block of code a specific number of times.

**Syntax:**

```
for var in iterable:
    do this
```

**Examples:**

```
for i in range(5):
    print i

for letter in "ATGCG":
    print letter
```
The **for** loop

**Purpose:** execute a block of code a specific number of times.

**Syntax:**
```
for var in iterable:
    do this
```

**Examples:**
```
for i in range(5):
    print i

for letter in "ATGCG":
    print letter
```

Note the indentation, much like with if/else statements, is very important and defines what belongs to the for loop.

Note that "var" can be any variable name you want, and you do not need to define/declare it before using it here.
Ways of using the \texttt{for} loop

The simplest way to create a loop that loops a certain number of times is to use \texttt{range()}: 

Example:

\begin{verbatim}
for i in range(5):
    print "hi"
\end{verbatim}

Result:

\begin{verbatim}
hi
hi
hi
hi
hi
\end{verbatim}

\texttt{range(5)} will loop 5 times
\texttt{range(6)} will loop 6 times
...and so on.
Ways of using the `for` loop

What `range(x)` actually does is create a list of numbers from 0 to `x-1`. A list is an iterable, so we can use it in the loop. The variable after `for` (here, `i`) will be assigned to each value in the iterable, one at a time.

Example:

```python
for i in range(5):
    print(i)
```

Result:

0
1
2
3
4

More about how range() works in a minute, and more on lists next time.
Ways of using the **for** loop

A string is also an iterable, and so we can use a **for** loop to iterate over each individual character in the string, one at a time:

Example:

```python
for letter in "Hello!":
    print letter
```

Result:

```
H
e
l
l
o
!
```

*Important to note:*

You can name the variable after for anything you want, and you do NOT need to define it before using it in the for loop.
Practice with `for`

What will the following code print?

```python
for i in range(4):
    print(i)
```
Practice with `for`

What will the following code print?

```python
for i in range(4):
    print i
```

Result:

0
1
2
3
Practice with `for`

What will the following code print?
```
for i in range(4):
    print i * 2
```
Practice with `for`

What will the following code print?

```python
for i in range(4):
    print i * 2
```

Result:

0
2
4
6
Practice with for

What will the following code print?

count = 0
for i in range(4):
    count = count + 1
print count
This is a basic counter. Kind of pointless in this particular example, but overall a very very useful thing. Remember it!

Practice with `for`

What will the following code print?
```python
count = 0
for i in range(4):
    count = count + 1
print count
```

Result:
4
Practice with **for**

What will the following code print?

```python
count = 0
for i in range(4):
    count = count + i
print count
```
Practice with `for`

What will the following code print?
```python
count = 0
for i in range(4):
    count = count + i
print count
```

Result:
6

---

*Important to note:*
This is similar to a counter, but instead of adding 1 each time, we're adding up various numbers.
This is sometimes called an *accumulator*, and it’s useful in many situations, so remember it!
Practice with `for`

What will the following code print?

```python
for nt in "ATGAT":
    print nt
```
Practice with for

What will the following code print?

```python
for nt in "ATGAT":
    print nt
```

Result:

A
T
G
A
T
Practice with for

What will the following code print?

```python
count = 0
for nt in "ATGAT":
    if nt == "A"
        count = count + 1
print count
```
Practice with `for`

What will the following code print?

```python
count = 0
for nt in "ATGAT":
    if nt == "A":
        count = count + 1
print count
```

Result:

2
Practice with `for`

What will the following code print?

```python
newSeq = ""
for nt in "ATG":
    newSeq = newSeq + nt + "*"
print newSeq
```
Practice with `for`

What will the following code print?

```python
newSeq = ""
for nt in "ATG":
    newSeq = newSeq + nt + "*"
print newSeq
```

Result:

A*T*G*

---

Important to note:

This is sort of like an accumulator for strings. We can build up a string in a loop by repeatedly concatenating characters to an existing string.

Don’t concatenate onto the original string as you iterate over it. This is bad form and could cause weird results. Just create a new string.

---

Kind of a dumb example, but a very important concept.
More about `range()`

**Purpose:** Creates a list with the indicated range. If only one parameter \( n \) is given, will automatically create a list from 0 to \( n-1 \).

**Syntax:**

\[
\text{range} (\text{start, stop, interval})
\]

**Examples (in interpreter):**

```python
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1, 6)
[1, 2, 3, 4, 5]
>>> range(0, 11, 2)
[0, 2, 4, 6, 8, 10]
```

It may seem a little odd that it goes from 0 to \( n-1 \), but since most things in programming start counting at 0, it ends up being more convenient this way.

We'll learn more about lists next time!
Practice with `range()`

What will the following code print?

```python
print range(4)
```
Practice with \texttt{range()}\texttt{()}

What will the following code print?
\begin{verbatim}
    print range(4)
\end{verbatim}

\textbf{Result:}
\begin{verbatim}
    [0, 1, 2, 3]
\end{verbatim}
Practice with `range()`

What will the following code print?

```python
print range(4, 8)
```
Practice with `range()`

What will the following code print?
```python
print range(4, 8)
```

Result:
```
[4, 5, 6, 7]
```
Practice with `range()`

What will the following code print?

```python
print range(0, 50, 10)
```
Practice with `range()`

What will the following code print?
```python
print range(0, 50, 10)
```

Result:
```python
[0, 10, 20, 30, 40]
```
3. while loops
Example

Instead of:

```python
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
print random.randint(0,1)
```

You can write:

```python
for i in range(10):
    print random.randint(0,1)
```

Or:

```python
count = 0
while count < 10:
    print random.randint(0,1)
    count = count + 1
```
The while loop

**Purpose:** execute code until the conditional statement becomes False.

**Syntax:**

```python
while conditional:
    indented code will execute until the conditional becomes false
```

**Example:**

```python
x = 0
while x < 4:
    x = x + 1
```

How is this different than a for loop?
Unlike the for loop, you don't need to specify exactly how many times the loop will run.

Again, the indentation is what defines what is part of the loop
Practice with `while`

What will the following code print?

```python
x = 0
while x < 4:
    print "hi"
    x = x + 1
```
Practice with `while`

What will the following code print?

```python
x = 0
while x < 4:
    print "hi"
    x = x + 1
```

Result:

```
hi
hi
hi
hi
hi
```
Practice with **while**

What will the following code print?

```python
x = 0
while x < 4:
    print x
    x = x + 1
```
Practice with while

What will the following code print?

```python
x = 0
while x < 4:
    print x
    x = x + 1
```

Result:

0
1
2
3
Practice with *while*

What will the following code print?

```python
x = 0
while x < 4:
    x = x + 1
print x
```
Practice with while

What will the following code print?

```python
x = 0
while x < 4:
    x = x + 1
    print(x)
```

Result:

```
1
2
3
4
```
Practice with `while`

What will the following code print?
```
x = 0
while x < 4:
    x = x + 1
print x
```
Practice with **while**

What will the following code print?

```python
x = 0
while x < 4:
    x = x + 1
print x
```

Result:

4
A more useful example: Number guessing game

```python
secretNumber = 56
notGuessed = True

while (notGuessed):
    guess = int(raw_input("What number am I thinking of? "))
    if (guess == secretNumber):
        print "Wow, you got it!"
        notGuessed = False
    else:
        print "Wrong, guess again."
```

A more useful example: 
Number guessing game

```
secretNumber = 56
notGuessed = True

while (notGuessed):
    guess = int(input("What number am I thinking of? "))
    if (guess == secretNumber):
        print "Wow, you got it!"
        notGuessed = False
    else:
        print "Wrong, guess again."
```

By using a `while` loop, we give the user unlimited chances to guess.
With while loops, you must be a little more careful than with for loops because of the possibility of creating an endless loop. An endless loop occurs when your conditional never becomes False, thus causing you to never exit the loop.

**Beware: endless loops**

**Code:**

```python
count = 1
while (count <= 10):
    print count
```

**Output:**

```
1
1
1
1
1

... (never ending)
```

Since we never increment count within the loop, it always remains 1, and therefore the while condition is always True.
Endless loops

Always watch out for possible endless loops! If you’re not sure, temporarily add a print statement somewhere in the loop so you can monitor how many times the loop runs.

If you find your code is taking an unexpectedly long time to run, check for an endless loop.

Stopping a program that is stuck in an endless loop: Ctrl + c
More practice with \texttt{while} loops

Endless loop or not?

\begin{verbatim}
count = 0
while (count < 10):
    print count
    count = count + 1
\end{verbatim}
More practice with while loops

Endless loop or not?

```python
count = 0
while (count < 10):
    print count
    count = count + 1
```

Answer: no
More practice with *while* loops

Endless loop or not?

```python
count = 0
while (count > 5):
    print count
    count = count + 1
```
More practice with while loops

Endless loop or not?

```python
count = 0
while (count > 5):
    print count
    count = count + 1
```

Answer: no

(this won't print anything, actually, since the condition count > 5 is never True)
More practice with \texttt{while} loops

Endless loop or not?

\begin{verbatim}
count = 0
while (count != 5):
  print count
  count = count + 1
\end{verbatim}
More practice with while loops

Endless loop or not?

```python
count = 0
while (count != 5):
    print count
    count = count + 1
```

Answer: no
More practice with \texttt{while} loops

Endless loop or not?

\begin{verbatim}
count = 0
while (count != 5):
    print count
    count = count + 1
\end{verbatim}
More practice with \texttt{while} loops

Endless loop or not?

\begin{verbatim}
count = 0
while (count != 5):
    print count
    count = count + 1
\end{verbatim}

Answer: \textbf{yes}

Why? \textit{We never increment} \texttt{count within the loop, so it never becomes equal to 5.}
More practice with `while` loops

Endless loop or not?

```python
count = 0
while (count != 5):
    print count
    count = count + 2
```
More practice with \texttt{while} loops

Endless loop or not?

\begin{verbatim}
    count = 0
    while (count != 5):
        print count
        count = count + 2
\end{verbatim}

Answer: yes

Why? Since we're incrementing count by 2 each time, count takes the values 0, 2, 4, 6, 8, etc. count never equals 5, so the condition \texttt{count \!=\! 5} never becomes False, and we keep looping forever.
Which kind of loop should I use?

In general:

• **Use a `for` loop when:**
  - You know exactly how many times you need to loop
  - You want to process each line of a file (as we'll see soon) or item in a list (as we'll see next time)

• **Use a `while` loop when:**
  - You need to loop until some condition is fulfilled, but you don't know when that will happen

You can usually use either type of loop, but one will feel a lot more natural and be easier to code. That's the one you should go with.
4. Application of loops:
   file reading
File reading

• File reading (and writing) is something you'll probably be doing a lot in your work
• Luckily, Python makes it super easy!
• Today we'll cover file reading
File reading

The 3 basic steps of file reading:

1. Open the input file
2. Read in data line by line, do some processing
3. Close the input file

File writing is very similar, but we'll save it for the next lesson.
Example of simple file reading

# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
    print "Line:", line
inFile.close()
What `open()` technically does is create something called a File object, and this is what we store in the variable "`inFile`". This file object is an iterable object, so we can use a loop on it. The individual iterable units are lines in the file, so during each iteration a single line is assigned to the loop variable. The lines will always be read from first to last.
Example of simple file reading

```python
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
    print "Line:", line
inFile.close()

If this is genes.txt, what will this script output?
```

*genes.txt:*
uc007afd.1
uc007ain.1
uc007afr.1
uc007atn.1
uc007bcd.1
uc007bmh.1
uc007byr.1
Example of simple file reading

```python
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
    print "Line:", line
inFile.close()
```

Output:
Line: uc007af.d.1
Line: uc007a1n.1
Line: uc007afr.1
Line: uc007atn.1
Line: uc007bcd.1
Line: uc007bnh.1
Line: uc007byr.1
Example of simple file reading

```python
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
    print "Line:\", line

inFile.close()
```

**Output:**
```
Line: uc007afd.1
Line: uc007aln.1
Line: uc007afr.1
Line: uc007atn.1
Line: uc007bcd.1
Line: uc007bmb.1
Line: uc007byr.1
```

**Why are there extra spaces?**
Because of invisible \n characters!
When we read each line of the file, there is actually a \n on the end of each line. This gets read in as part of the string. Then print adds another \n on the end when it prints the string (as it always does). This is what causes the double spacing – we technically have \n\n on the end of each string.
As you may have noticed, the print command automatically inserts a \n after it prints everything (if it didn't, you'd have everything running together on the same line when you use multiple print statements).

## Side note: Newline (\n)

- Whenever you hit "enter" or "return", you're actually inserting a newline character, which is invisible when you view the file in a text editor
- This "character" is \n, and you can manually insert it into your strings when you're printing to create newlines wherever you want.

For example:

```
print "Hello\nWorld"
```

Output:

```
Hello
World
```
Simple file reading, with \n removal

```python
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
    line = line.rstrip('\n')
    print "Line:", line
inFile.close()
```

You can pretty much use this as a template for most file reading situations. The main part you'll be changing is to replace

    print "Line:", line

with something more interesting/useful
Simple file reading, with `\n` removal

```python
# Read and print genes.txt
fileName = "genes.txt"

inFile = open(fileName, 'r')
for line in inFile:
    line = line.rstrip('\n')  # `rstrip()` removes the indicated character from the end of the string, if it is there. If the indicated character is not there, does nothing.
    print "Line:" , line
inFile.close()
```

There are many cases when the `\n` will interfere with what you want to do, so it's good to get in the habit of including this line of code.
File reading functions

- When you open a file, you're actually creating what's called a "File object" – this is what gets assigned to the variable.
- You can think of the File object as simply an interface to the file you're working with.
- File objects come with a set of special methods related to reading and writing files:
  - `.read()` - reads in the entire file at once
  - `.readline()` - reads one line at a time
  - `.readlines()` - reads all lines in file into a list
  - `.write()` - write a string to a file
  - `.close()` - close the file

These methods can only be used on file objects. The syntax for using them is similar to what we've seen with modules, e.g.:

```python
inFile.read()
inFile.readline()
```

Where "inFile" should be replaced with whatever variable name you assigned to your file when you opened it.
File reading functions

Examples:

```python
inFile = open("genes.txt", 'r')  #create file object

header = inFile.readline()      #read first line of file
line = inFile.readline()        #read second line of file
restOfLines = inFile.readlines() #read rest into list

inFile.close()                 #clean up after ourselves
```