WELCOME
BACK!
Questions?
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td></td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min() / max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int() / float()</td>
<td></td>
<td>if / elif / else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True / False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Revisiting Square() 

Remember this:

```python
import turtle

def line_and_turn(length, angle):
    turtle.fd(length)
    turtle.rt(angle)

def square(size):
    line_and_turn(size, 90)
    line_and_turn(size, 90)
    line_and_turn(size, 90)
    line_and_turn(size, 90)

def triangle(size):
    line_and_turn(size, 120)
    line_and_turn(size, 120)
    line_and_turn(size, 120)
```

we can dramatically improve this code with loops.
Why Loops?

Sometimes you need to run some lines of code multiple times. In particular maybe you don't know ahead of time (i.e. when *programming*) how many times the code must run, but you will know during *execution*.

This is exactly what loops are for.

Think about greeting guests for a party. You don't know how many people are coming but you need to greet each of them.

pseudocode:

```plaintext
while there are un-greeted guests:
    greet the next guest
```
Loops

two types of loops, **for** and **while**, we'll start by looking at **while** loops

syntax:

**while conditional:**

  *code to run while condition is true*

  *code to run after the loop is done*

The code will be executed over and over while the condition remains true. Once the condition is false the function will move on.
Example loop

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

what will this print?
how would it change if we swapped the order of the two lines of code?
One downside to while loops is that accidental infinite loops are possible.

```python
x = 0
while x <= 10:
    print(x)
```

we left off the line that changed `x`, consequently the conditional never stops being true and the loop runs forever.

ctrl-c to stop the execution (command-c on mac?)
Remember this:

```python
import turtle

def line_and_turn(length, angle):
    turtle.fd(length)
    turtle.rt(angle)

def square(size):
    sides = 0
    while sides < 4:
        line_and_turn(size, 90)
        sides += 1

def triangle(size):
    sides = 0
    while sides < 3:
        line_and_turn(size, 120)
        sides += 1
```
Remember this:

```python
import turtle

def line_and_turn(length, angle):
    turtle.fd(length)
    turtle.rt(angle)

def poly(size, num_sides):
    sides = 0
    while sides < num_sides:
        line_and_turn(size, 360/num_sides)
        sides += 1
```
def poly(size, num_sides):
    sides = 0
    while sides < num_sides:
        line_and_turn(size, 360/num_sides)
        sides += 1

def lots_of_polys(size, max_sides):
    num_sides = 3
    while num_sides <= max_sides:
        poly(size, num_sides)
        num_sides += 1

poly() draws any regular polygon. lots_of_polys() draws all the polys of sides 3, 4, 5…max_sides.

let's see it in action.
CIS122 Summer

WELCOME BACK!
Questions?
## Where We Are

<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min() / max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int() / float()</td>
<td></td>
<td>if / elif / else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True / False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As with functions and if statements, indents are used to group text together, and once again mistaking what should be indented and what shouldn't is a VERY common error.
def loop_to_ten(x):
    while x < 10:
        x = 0
        print(x)
        x += 1
Spot the error

def loop_to_ten():
    x = 0
    while x < 10:
        print(x)
    x += 1
Examples

Loops lend themselves to example problems, so let's do a few.
Compound interest can be described thusly:
\[ M(t+1) = M(t) + rM(t) \]

where \( M(t) \) is the money at time \( t \)

\( r \) is the interest rate (as a decimal)

and \( M(t+1) \) is the money after one unit of time (where the time unit is decided by how often the interest is compounded (yearly, daily, monthly))

Lets write a function that takes a starting \( M \), an amount of time, and a rate and figures out the final amount of money. We'll assume no withdrawals.
Fibonacci series

let's make a function to print out all the fibonacci numbers up to some arbitrary place.

Fibonacci series:
\[ f(1) = 1 \]
\[ f(2) = 1 \]
\[ f(n) = f(n-2) + f(n-1) \]
Check string for numbers

Lets write a function to check each character of a string and if the character is a digit (i.e. number) it prints out that digit. It also keeps track of the number of digits found and prints that number at the end.

Don't try to build it all at once, do one piece at a time. Let's start by building a function that prints out each letter in a string in turn.

Here slicing really shines.
Check list for while (indefinite) loops
1. Set up the loop/end condition
2. Initialize the loop variable (outside of the loop)
3. Write the body of the loop
4. Advance the counter variable
5. What to do when the loop is done?
WELCOME BACK!
Questions?
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>float type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>string help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>bool min()/max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>bool int()/float()</td>
<td></td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>bool round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>bool len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>bool Input()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool &lt;user defined&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool &lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Looping on Sequences

Remember that strings are sequences of characters (strings are not the *only* sequences in python however).

A very common activity is to loop over a sequence and do something with each item in that sequence.

example-count or print the individual characters in a string
def count_letters(aString):
    index = 0
    count = 0

    while index < len(aString):
        char = aString[index]
        count += 1
        index += 1

    return count

But we could do this with len() already, in fact we used the len function in our code above.
def count_lowercase(aString):
    index = 0
    count = 0

    while index < len(aString):
        char = aString[index]
        if char.islower():
            count += 1
        index += 1

    return count
For loops are specifically designed to run over a sequence applying the code within the loop once for each object in the sequence.

This specialization makes them less flexible but also a bit safer and easier to use.
def count_lowercase(aString):
    index = 0
    count = 0

    while index < len(aString):
        char = aString[index]
        if char.islower():
            count += 1
        index += 1
    return count

def count_lowercase2(aString):
    count = 0

    for char in aString:
        if char.islower():
            count += 1
    return count
for <obj> in <seq>:
    <code to execute>

for char in aString:
    print(char)

for i in str1:
    print(i)

Let's try it.
Note both code snippets above work exactly the same. The differences in the names (char/i and aString/str1) are purely cosmetic. They are variables standing in for the actual data we will have later.
for <obj> in <seq>:
    <code to execute>

for char in aString:
    print(char)

for i in str1:
    print(i)

it may help to watch the code work in the visualizer.
Let's write a function that reverses the capitalization of a string argument, i.e. 'Hello' becomes 'hELLO'
we'll use a for loop to make our lives easier.
Advantages/Disadvantages

Advantages-
Because the for loop operates once for each item in the sequence it is impossible to accidentally cause an infinite loop.

The for loop is more compact and sleek when operating over a sequence than the while because it is specially built for that task.

Disadvantages-
The while loop is much more flexible. If you want to get every other item from a sequence you can do it with a for loop but it’s probably just as easy to use a while loop (actually you can do this specific thing with a slice very easily).
Questions?
# Where We Are

<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min()/max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int()/float()</td>
<td></td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If you just want a loop to run a certain number of times there's a way to do that with a for loop.

All of our examples yesterday involved manipulating a sequence as part of the loop, but maybe we just want the sequence to serve as a counter of sorts.
the built in range() function gives us a sequence of numbers from 0 up to the argument value passed to range.

range(5) for instance gives us a sequence that runs from 0 to 4.

range() can also be called with 2 or 3 arguments.
range(1,5) gives us a sequence from 1 to 4.
range(0, 5, 2) gives us a sequence from 0 to 4 by twos (i.e. 0, 2, 4)
let's write a function that counts (prints) the numbers up to some argument we give the function.

Let's write both as a while loop and a for loop.
Nested loops

As I was going to St. Ives,
I met a man with seven wives,
Each wife had seven sacks,
Each sack had seven cats,
Each cat had seven kits:
Kits, cats, sacks, and wives,
How many were there going to St. Ives?

Of course the poem above is a trick question but let's say we wanted to calculate the number of kits, cats, sacks, and wives. We could use algebra, but let's make the computer do the work for us.
def st_ives():
    for each wife
        count the wife
    for each sack
        count the sack
    for each cat
        count the cat
    for each kitten
        count the kitten
    announce the total

Okay let's write the function
We should look at our st_ives code in the visualizer so see how it operates.
while loops are *indefinite*- they can potentially loop infinitely (which may be desired or not)

for loops are *definite*- they loop over some thing which is finite
Recursion is a way of looping without actually using loops. We aren't going to have time to really learn recursion but I want you to get a taste of it.

```python
def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

You will not be tested on recursion.
The essence of recursion is to have a base case, below the base case is when \( n == 0 \), and if you aren't at the base case the function calls itself on an input that is closer to the base case.

```python
def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

You will **not** be tested on recursion