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></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></td>
<td>min()/max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int()/float()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td></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>
So far our programs have executed in a straight line.

Let's change that.
Bools!

To allow our programs to branch we need to be able to have some kinds of tests and depending on the result we go different directions

In order to do *that* we need a new type: the humble bool (short for boolean)

bools can only have two values, either **True** or **False** (both keywords). Notice both **True** and **False** are capitalized.
There are a number of operators which return bools, we call them comparison operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Effective Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>are these equal?</td>
</tr>
<tr>
<td>!=</td>
<td>are these not equal?</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>same as != above</td>
</tr>
<tr>
<td>&gt;</td>
<td>is the first 'greater' than the second?</td>
</tr>
<tr>
<td>&lt;</td>
<td>is the first 'less' than the second?</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is the first 'greater' than or equal to the second?</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is the first 'less' than or equal to the second?</td>
</tr>
</tbody>
</table>
For the moment we'll just stick to a very primitive understanding of bools and return for the nuance/details later.
The **if** statement gives us our first flow control mechanism. Notice **if** is a keyword. How does it work?

```python
if 5 < 10:
    print("Five is less than ten.")
```
If some conditional:
    (code to run if the conditional is True)

This is pseudocode, not real code
Note the code to be run is indented, just like with the function definitions. Also note like a function definition the indent comes after a colon. The indent serves the same purpose- to group some text together.
Example If Code

def less_than_ten(x):
    if x < 10:
        print("Your number is less than ten.")
        return
    print("your number is more than ten")

The second print would only happen if the conditional were False.

P.S. There's a logic error in the program above. See it?
We may want to have a branch that specifically is followed if the if conditional is false. We could have a separate if of the opposite of our first condition but that's clunky. Instead we can use else. 

You can only have an else with an if, it cannot exist by itself. For a given if you can only have one else. Let's rework our code using else.

```python
def less_than_ten(x):
    if x < 10:
        print("Your number is less than ten.")
    else:
        print("your number is more than ten")
```
Elif

Cool. But we had that logic error when \( x = 10 \). We can fix that with the last component of \texttt{if} conditionals, else-if abbreviated as \texttt{elif}.

Like \texttt{else}, you can only have an \texttt{elif} with an \texttt{if}, it cannot exist by itself. Unlike \texttt{else}, you can have any number of \texttt{elif}s for each \texttt{if}. Let's rework our code using \texttt{elif}.

```python
def less_than_ten(x):
    if x < 10:
        print("Your number is less than ten.")
    elif x == 10:
        print("Your number is ten.")
    else:
        print("Your number is more than ten")
```
How's it all work?

Each test is sequential, only one branch will be followed, other two ignored. If else used then we will always follow one branch, otherwise we might follow none of them.

```python
def less_than_ten(x):
    if x < 10:
        print("Your number is less than ten."")
    elif x == 10:
        print("Your number is ten."")
    else:
        print("your number is more than ten")
```
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></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>
After the branch

After all if statements the function will continue if no return reached. Returns end the function completely. Here's a function that takes in a numerical grade and converts it to a letter grade.

```python
def letter_grade(number):
    if number >= 90:
        out = 'A'
    elif number >= 80:
        out = 'B'
    elif number >= 70:
        out = 'C'
    elif number >= 60:
        out = 'D'
    else:
        out = 'F'
    print(out)
    return out
```
Here's another *implementation* of letter_grade, does it work? Hint-no.

```python
def letter_grade(number):
    if number >= 90:
        out = 'A'
    if number >= 80:
        out = 'B'
    if number >= 70:
        out = 'C'
    if number >= 60:
        out = 'D'
    else:
        out = 'F'
    print(out)
    return out
```

why not?
Let's try the visualizer.
There's many ways we could write `letter_grade` but two general big ideas:

```python
def letter_grade(number):
    if number >= 90:
        out = 'A'
    elif number >= 80:
        out = 'B'
    elif number >= 70:
        out = 'C'
    elif number >= 60:
        out = 'D'
    else:
        out = 'F'
    print(out)
    return out
```

```python
def letter_grade(number):
    if number >= 90:
        out = 'A'
    elif 90 > number >= 80:
        out = 'B'
    elif 80 > number >= 70:
        out = 'C'
    elif 70 > number >= 60:
        out = 'D'
    else:
        out = 'F'
    print(out)
    return out
```
The key questions to ask yourself are these:

- should my function only ever take on branch?
  if yes then use if and elifs
  if no then use repeated ifs

  (think of non-exclusive traits for the latter case, like biology classifications)

- should my function always take one branch?
  if yes use an else
  if no don't

  (if checking for some specific state or case you often don't need an else for example)
iseven() checks if a number is even. Here it's written by a beginner.

```python
def iseven(num):
    if num % 2 == 0:
        return True
    else:
        return False
```
```python
def iseven(num):
    return num % 2 == 0

That popping sound was your mind being blown.
```
Order of Operations

order of operations:
() parens,
** exponents,
+ - unary +/-,
* / multiplications,
% modulus,
+ - addition,
== comparison operations,
!=
>=
<=
>
<
and bool operators
or
not
You can freely nest `if` statements. The code below looks first to establish letter grade then has a nested if to establish any +'s or -'s. This could be done with one big `if-elif-else` statement but this way has some advantages.

```python
def letter_grade(number):
    if number >= 90:
        if number >= 98:
            out = 'A+'
        elif number <= 92:
            out = 'A-
        else:
            out = 'A'
    elif number >= 80:
        ...
```

Nested ifs
Nested ifs

Now we've added a boolean argument to the function: letter_only. If letter_only is **True** then only the letter grade is returned, if **False** then the +'s and -'s are determined as well.

```python
def letter_grade(number, letter_only):
    if number >= 90:
        if letter_only:
            out = 'A'
        else:
            if number >= 98:
                out = 'A+'
            elif number <= 92:
                out = 'A-'
            else:
                out = 'A'
    elif number >= 80:
        ...
```

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></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 bools

bools are a fundamental type in python. They only have two possible values, True or False.

bools have their own operators, similar to +, -, *, etc for ints/floats.

bool operators are and, or, and not
**and**

`and` is an operator on `bool`s, it returns `True` if and only if both `bool`s are `True` otherwise it returns `False`.

<table>
<thead>
<tr>
<th>bool 1</th>
<th>bool 2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
or is an operator on bools, it returns True if and only if at least one of the bools are True otherwise it returns False
logical exclusive or is not an operator in python but you will build a function to do this on the homework so it’s worth a quick look. Exclusive or is **True** if and only if *exactly* one **bool** is **True**.

<table>
<thead>
<tr>
<th>bool 1 or bool 2</th>
<th>bool 1</th>
<th>bool 2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bool 1 xor bool 2</th>
<th>bool 1</th>
<th>bool 2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
not is an operator on a bool, it returns True if and only if the bool is False otherwise it returns False

<table>
<thead>
<tr>
<th>not Bool1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

notice that and and or are binary operators: they need two bools. Meanwhile not is unary: it acts on a single bool.
True and True and True and False

Not really that different than $1 + 1 + 1 + 0$

$1 + (1 + (1 + 0))$
$1 + (1 + (1))$
$1 + (1 + 1)$
$1 + (2)$
$1 + 2$
$3$

True and (True and (True and False))
True and (True and (False))
True and (True and False)
True and (False)
True and False
False

could we have started at the other end?
True and (True and (not True or False) or (not False) and True)

True and (True and (False or False) or (True) and True)

True and (True and (False) or True)

True and (False or True)

True and (True)

True

not comes before and
which comes before or
When things get weird(er)

```
x = 1
y = 2
print(x and y)
print(x or y)
```
x = 1
y = 2
print(x and y)
print(x or y)

Technically python returns one of the operands with **and** and **or**. Why? no idea. It's a design choice of the language. Essentially **or** returns the first "truthy" value it finds while **and** returns the first "falsy" value it finds. In either case if they don't find what they are looking for they return the last value they found.

If using **True/False** this isn't a problem but if you start using logical operators on numbers/strings it will get strange fast.
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>
We want a function that tests if the first argument is larger than both the number 10 and also the second argument.

```python
def whoa(x, y):
    if not x > 10 and y:
        print("Yah!")

whoa(3, 5)
```
We want a function that tests if the first argument is larger than both the number 10 and also the second argument.

```python
def whoa(x, y):
    if not x > 10 and y:
        print("Yah!")
```

should be

```python
def whoa(x, y):
    if not x > 10 and x > y:
        print("Yah!")
```

but this raises the question of why it worked at all
It gets worse

def whoa(x, y):
    if not x > 10 and y:
        print("Yah!")

whoa(3,5)

This should have evaluated the following conditional:
not x > 10 and y
not 3 > 10 and 5
not False and 5
True and 5
try this:
print( True + True )
Your reaction, most likely

Don't worry I felt the same way.
The binary nature of boolean values has led to them being treated as 1s and 0s in a lot of languages, and python is no exception.

So $1 + \text{True}$ is the same as $1 + 1$.
$5/\text{False}$ will give a divide by zero error.

Consequently you may get a lot of strange behavior if you aren't careful with booleans.
Any number (int or float) is treated as True unless it has a value of 0 in which case it is treated as False

Similarly any string is treated as True unless it is an empty string (i.e. "") in which case it is False

so what does this do?

if "False":
    print("Are we doing this thing or what?")
Things that work...

...but probably shouldn't.

"Hi, my name is Jason!" [False]

1 == True

(5 / (True + True + True + True + True)) == 1

if 1:
    print("Why is 1 True?")
print( str(True) )
print( str(False) )

print( int(True) )
print( int(False) )

print( float(True) )
print( float(False) )

print( bool(0) )
print( bool(1) )
print( bool(-1) )
print( bool("a") )
print( bool("""" )

print( bool("") )