INTRODUCTION TO PROGRAMMING
WEEK 3 - CONTROL FLOW STATEMENT

WHAT IS FLOW?

In programming, when we talk about "flow" we are speaking primarily about the execution of a program - it's order, speed, beginning, and end. For our basic programs, programs usually execute top-to bottom.

WHY DOES FLOW NEED TO BE CONTROLLED?

Think of your life as a gigantic script (`mylife.py`). When you wake up in the morning until you fall asleep at night, you don't do the SAME things every day (if you do, I feel very sorry for you). Instead, the actions you take are dependent on the circumstances and actions of others. For instance, when you cross the street on your way to work - you only walk when there are no cars coming. If you are thirsty, you will drink something. Just like your life, scripts almost always execute conditionally. If you have a program that does not use control flow, then you have one extremely un-dynamic program which probably is not that useful.

CODE BLOCKS

A code block is a set of statements that will be executed under a certain condition. If I were to write a script for your life, I might have a statement that says "If you cut your finger, you will do the following: wash it under hot water, apply antibiotics, put a bandaid over it, and then return to your desk". The set of actions, "wash, antibiotics, etc." that would be the code block that is to be executed if you cut your finger. In Python, code blocks are designated using indentations. For instance (don't worry if you don't understand the syntax here):

```
if my_finger.iscut == True:
 WashUnderHotWater(my_finger)
 ApplyAntibiotics(my_finger)
 ApplyBandaid(my_finger)
```

Notice here that the **code block** for treating the cut is indented under the conditional statement (`if my_finger.iscut == True`). **This tells the program to execute those 3 statements only if the condition is true.**

While indents can be of any size (as long as they're consistent), the Google Python Style Guide stipulates that you should always use 2 spaces for indents.

Remember, a code block is a block of statements (indented at the same level), that will only execute if the conditional statement above them evaluates to True.

BOOLEANS

Before we go into control flow statements, I should explain the `boolean` data type which is a primitive (remember from last week?) that evaluates out to 2 values - `True` and `False`. While variables of type int can be 1, -1, 2, -2, etc. `booleans` can only be `True` or `False`. `Booleans` are the basis of control flow statements because they indicate whether certain conditions are `True` and subsequently which code blocks should be executed. Examples:

```
FingerIsCut = True

if FingerIsCut == True:
 TreatCut(my_finger)
if FingerIsCut == False:
 KeepTyping()

# (notice the indents!)
```

IF STATEMENTS

An `if` statement is exactly as it sounds, it evaluates a condition and executes a subsequent code block if that statement is `True`. The way an `if` statement evaluates a condition is with a set of inequality statements.

```
int1 = 3
int2 = 5

if int1 == int2:
 # Only indented 2 SPACES!
 print 'int1 is equal to int2'

if int1 != int2:
 print 'int1 is not equal to int2'


if int1 > int2:
 print 'int1 is greater than int2'

if int1 < int2:
 print 'int1 is less than int2'
```

What will print out in this case?

A few notes on convention:

- `if` statements are always written with this syntax:
    - `if <condition>:`
    - there should be a space separating `if` and the condition, you can also bundle

the condition in `()` if you like, but the style guide advises against it. **The colon at the end is KEY.**

- The `==` operator is different from the `=` operator. The first is an equality operator and evaluates to a boolean. `3 == 3` is `True`, while `3 == 2` is `False`. The first is an assignment operator, it assigns a value to a variable. `int1 = 2`.
- See the list below for the major inequality operators we will be using (most of these should be familiar from Algebra).

List of Inequality Operators

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

BOOLEAN OPERATORS (AND/OR)

In addition to making one comparison, you can make and/or comparisons. This is useful for a variety of reasons. For instance, in our example you could say `If hungry == True and time == lunch_time: eatlunch()`. Here are some examples.

```
int1 = 3
int2 = 5

if int1 ==  int2 or int1 > int2:
 print 'int1 is greater than or equal to int2'

if int1 != int2 and int1 == 3:
 print 'int1 is NOT equal to int2 AND is equal to 3'
```

ELSE STATEMENTS

`else` statements are evaluated if the `if` statement above it return false. For instance:

```
int1 = 3
int2 = 5
```

```
if int1 == int2:
 print 'int1 is equal to int2'
else:
 print 'int1 Is NOT equal to int2'
```

Keep in mind that `else` statements execute in relation to the nearest `if` code block above it, so this statement:

```
if int1 == int2:
 print 'int1 is equal to int2'
if int1 < int2:
 print 'int1 is greater than int2'
else:
 print 'int1 is NOT equal to int2'
```

Would be an incorrect way of determining if `int1== int2` since the `else` statement is only executing in relation to the statement `if int1 > int2`.

ELSE IF STATEMENTS

Writing multiple `if` statements, while seemingly logical can actually become computationally expensive and illogical. For instance, consider the program below. It makes no sense to evaluate whether or not it is tuesday if we have established that it is monday. Yet, the program as it stands below will continuously evaluate all 7 days.

```
if day == 'mon':
 print 'it is monday'
if day == 'tues':
 print 'it is tuesday'
if day == 'wed':
 print 'it is wednesday'
```

The solution to this problem is, of course, another type of control flow statement . The **else-if** statement. You want to use this if you want to evaluate the veracity of one statement and then based on that evaluate other truth statements. For instance:

```
if today == 'mon':
 print 'today is monday'
elif today == 'tue':
 print 'today is tuesday'
elif today == 'wednesday'
 print 'today is wednesday'
else:
```

```
 print 'today is thursday, friday, saturday, or sunday'
```

Two things to note here:

- syntax for else-if is **`elif`**
- execution will NOT continue to **fall through** the block but will stop once it hits a `True` statement
- the **`else`** statement at the end of the block acts as a catch-all (executes if none of the above conditions are `True`)

In Python, the best way to use `elif` is to check for conditions that might be mutually exclusive. For instance, if it's monday, then it's not tuesday or wednesday, etc. It can also be used to build numerical ranges like so:

```
temp = 72

if temp > 100:
 print 'it is very hot'
elif temp > 80:
 print 'it is hot'
elif temp > 65:
 print 'it is warm, but not too hot'
elif temp > 45:
 print 'it is chilly'
else:
 print 'You clearly don\'t live in the bay area do you?'
```

What would print in this example?

```
it is warm, but not too hot
```

Notice the lack of continual fall through?

This prevents execution from falling through and creates ranges for evaluation (100 - 80, 80 - 65, 65 - 45, all else).

WHILE STATEMENT

A `while` statement executes a code block continuously while a condition is `True`. Each time the code block underneath the `while` statement executes, it checks the `while` condition again before executing. If the condition fails, then execution exits from the loop.

```
int1 = 0

while int1 < 6:
```

```
 print int1
 int1 = int1 + 1 #OR int1 += 1
```

What will execute here?

```
0
1
2
3
4
5
```

One danger of the `while` loop is an infinite loop. For instance:

```
int1 = 0

while int1 < 10:
 print int1
 int1 = int1 -1
```

This will never stop executing and will eat up the server's memory until it forces the program to shut down.

SIDE NOTE - INCREMENTOR/DECREMENTOR SHORTHAND

In programming, we often have to do a lot of static re-assigning of a variable's value. This happens most often in loops where a single variable will need to be incremented by 1 or decremented by 1 each time the loop executes.

We could write it like this:

```
int1 = int1 - 1
int2 = int2 + 1
```

Or we could use the incrementor/decrementor shorthand:

```
int1 += 1 # shorthand for int1 = int1 + 1
int2 -= 1 # shorthand for int2 = int2 - 1
```

These also work on strings:

```
str1 += 'hello' # shorthand for str1 = str1 + 'hello'
```

Here are some examples:

```
int1 = 5
int1 += 1
print int1 # prints 6
```

```
int2 = 6
int2 -= 1
print int2 # prints 5


str1 = 'hello'
str1 += ' world!'
print str1 # prints 'hello world!'
```

THE BREAK STATEMENT

In a `while` loop, you often want to stop execution at a certain point, to do that use the `break` statement:

```
int1 = 0

while int1 < 10:
 print int1
 if int1 == 5:
   print 'Only counting to 5!'
   break
 int1 += 1
```

This will effectively prevent execution from proceeding to the next iteration (6), and will print:

```
0
1
2
3
4
5
Only counting to 5!
```

Note here:

- **To nest conditions, simply indent again** - in this case we see that the code block that will be executed with the `if` statement is indented under it which is indented in the `while loop`.

A common use-case for the `while` loop is getting user information:

```
while True: # a seemingly infinite loop
  order_product = raw_input('What product do you want? ')
  order_num = raw_input('How many of that product do you want? ')
  .. # do something with the order

  order_another = raw_input('Would you like to order another product? ')
```

```
if order_another == 'no':
  break
```

END OF CLASS EXERCISE

Print the following using a `while` loop and `if-elif-else` statements:

```
the number is 1, that is the best
the number is 2, that's the type of pencil you use in the SAT
the number is 3, that rhymes with 'me'
the number is 4, i want some more
the number is 5, like a high-five!
the number is 6, what a neat trick...
```

Make it as elegant as possible, no repeated code please.