More Syntax, Keywords, Functions, Variable Scope

1. What do the +=, -=, *=, and /= symbols mean?

These are short-hand notation for a common operation in programming

\[
\begin{align*}
    x &+ n  \quad \text{<-- stands for -->} \quad x = x + n \\
    x &- n  \quad \text{<-- stands for -->} \quad x = x - n \\
    x &* n  \quad \text{<-- stands for -->} \quad x = x * n \\
    x &/ n  \quad \text{<-- stands for -->} \quad x = x / n
\end{align*}
\]

Most commonly, this is used to increment or decrement a variable, like so:

\[
\begin{align*}
a &+= 1  \quad \text{<-- increment variable a} \\
b &-= 1  \quad \text{<-- decrement variable b}
\end{align*}
\]

2. What is the % operator?

This is the modulo operator. It is used to get the remainder from a division. Here is an example:

\[
\begin{align*}
    5 \% 3 &= 2  \quad \text{When 5 is divided by 3 (5/3), 3 goes into 5 one time and the remainder is 2.} \\
    6 \% 3 &= 0  \quad \text{When 6 is divided by 3 (6/3), 3 goes into 6 two times and the remainder is 0.}
\end{align*}
\]

3. What does it mean to say `for x in my_string`?

In Python, there is an easy and intuitive way to iterate over all the characters in a string.
We use a for loop.

```
my_string = 'hello world'
for x in my_string:
    print x
```

The for loop has a variable called `x` whose value changes each time through the loop. Initially, `x` is `h`, then `x` is `e`, and so on (don’t forget that the space is a character too!), until the last value that `x` will take on, which is `d`. 
4. **What does break do in a program?**

The break statement in a loop causes the loop to terminate before it has finished running. The break statement causes the program to exit the innermost loop in which it is enclosed.

5. **Variable scope (what parts of my code know about what variables)**

Consider the following code:

```python
1   def f(x):
2       b = x + c
3       return b
4       c = 0
5   print f(1)  # outputs 1
6   print c     # outputs 0
```

This code runs with no errors. Line 4 sets a variable `c` to 0. Line 5 calls the function `f(x)` with 1 substituted for `x`. Inside `f(x)`, create a variable called `b` to be `x+c`, which is `1+0`. Here is the interesting thing. `f(x)` does not have its own variable called `c`, so it looks outside of itself to see if the program has a variable called `c`, which it finds and uses.

Now consider the modified code:

```python
1   def f(x):
2       b = x + c  # ERROR
3       c += 1
4       return b
5       c = 0
6   print f(1)
7   print c
```

This code has an error! `UnboundLocalError: local variable 'c' referenced before assignment`

Again, line 5 sets a variable `c` to 0. Line 6 calls the function `f(x)` with 1 substituted for `x`. Inside `f(x)`, create a variable called `b` to be `x+c`. Here lies the problem. Python sees that in line 3, we are assigning a variable called `c` to some value. Therefore, it assumes that `c` is a variable that exists only in this function. It does not look outside the function for a variable called `c` there! Therefore, since the variable `c` is never given some initial value, line 2 results in an error – the function `f(x)` does not know what the value of `c` is.

So it seems that you can access variables defined outside of a function as long as they are in a read-only way!