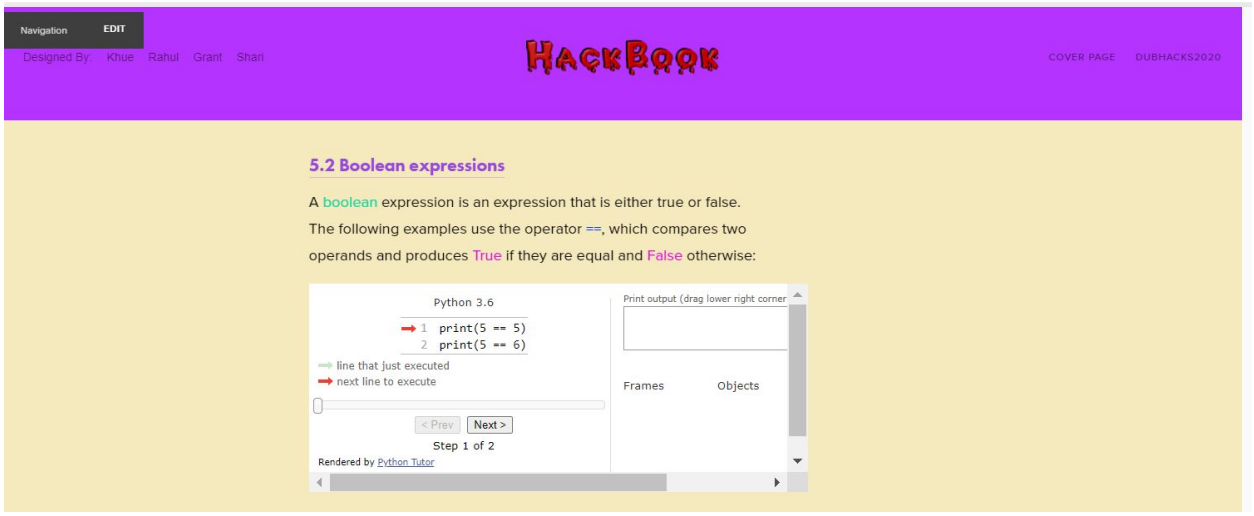
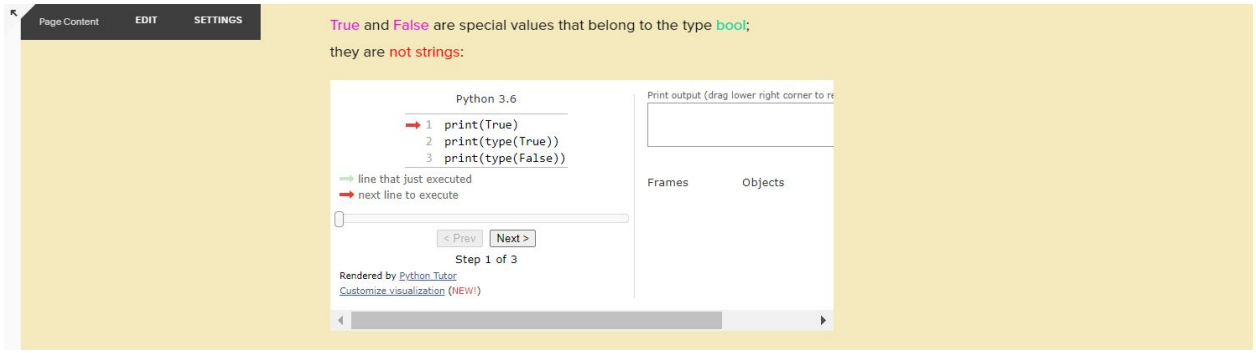
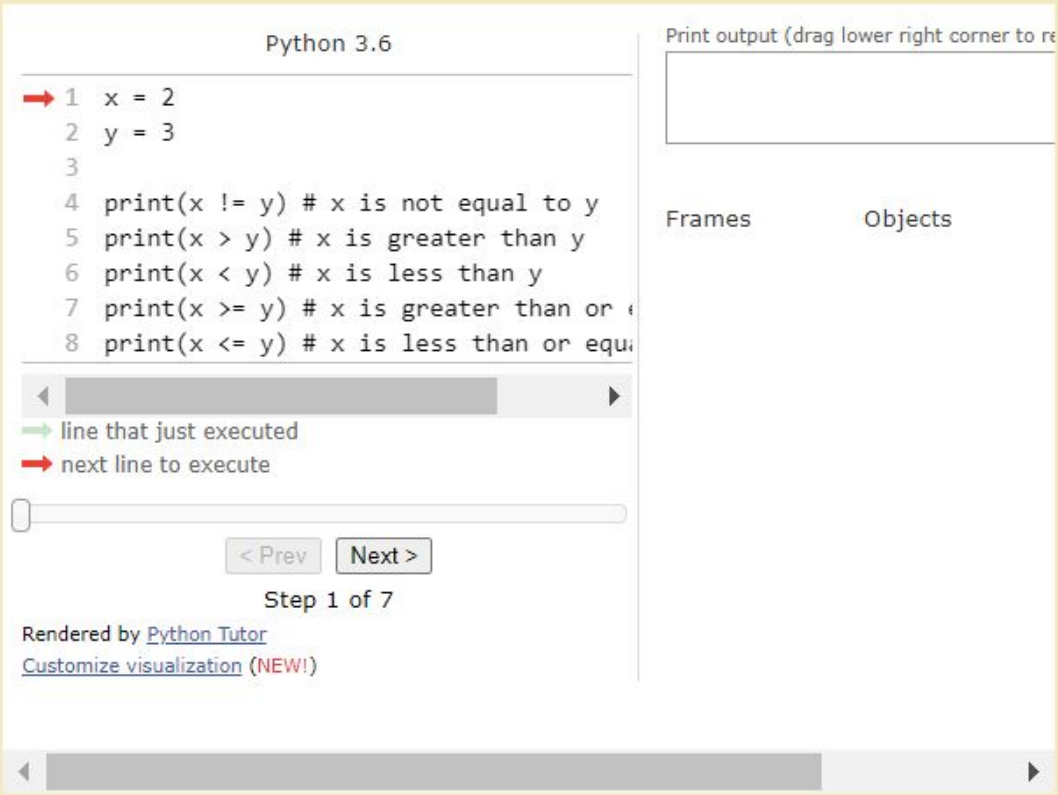
Expected Output:





The == operator is one of the relational operators; the others are: !=, >, <, >=, <=



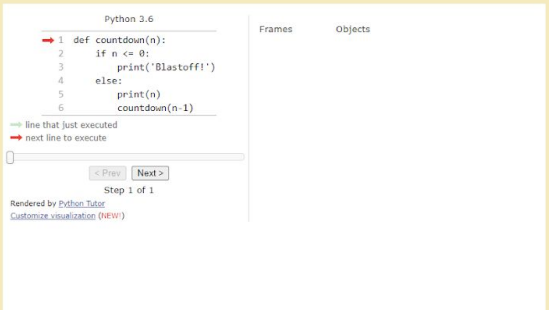
Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a relational operator. There is no such thing as =< or =>.

NEXT

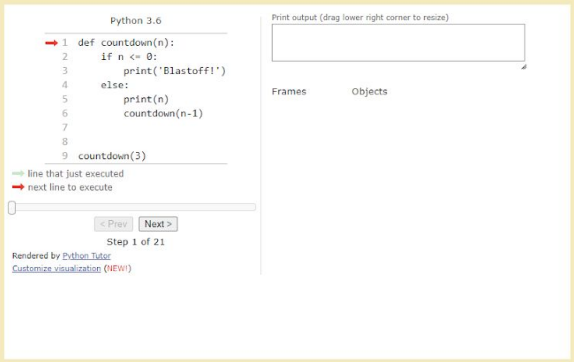
5.8 Recursion

Code EDIT

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:



If n is 0 or negative, it outputs the word, "Blastoff!" Otherwise, it outputs n and then calls a function named countdown—itself—passing n-1 as an argument. What happens if we call this function like this?



The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself... The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself... The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, "Blastoff!" and then returns. The countdown that got n=1 returns.

The countdown that got n=2 returns.

The countdown that got n=3 returns.

And then you're back in __main__. So, the total output looks like this:

A function that calls itself is recursive; the process is called recursion.

As another example, we can write a function that prints a string n times.

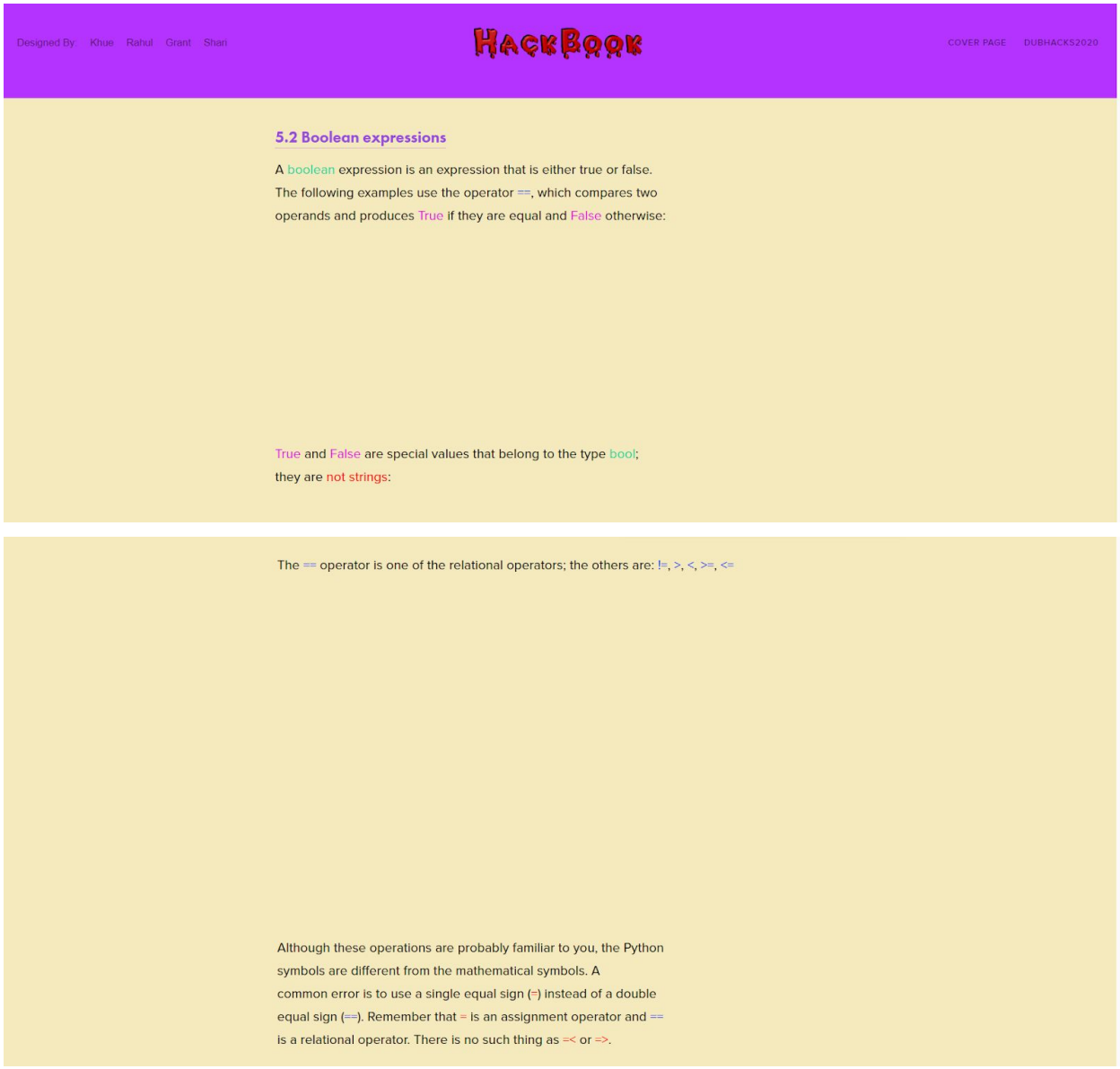
If n <= 0 the return statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

The rest of the function is similar to countdown: if n is greater than 0, it displays s and then calls itself to display s n−1 additional times. So the number of lines of output is 1 + (n - 1), which adds up to n.

For simple examples like this, it is probably easier to use a for loop. But we will see examples later that are hard to write with a for loop and easy to write with recursion, so it is good to start early.

LEARN MORE

Actual Output:



5.8 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

If n is 0 or negative, it outputs the word, "Blastoff!" Otherwise, it outputs n and then calls a function named countdown—itself—passing n-1 as an argument. What happens if we call this function like this?

The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

- The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself...
- The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...
- The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...
- The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...
- The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...
- The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, "Blastoff!" and then returns.
- The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, "Blastoff!" and then returns.
- The countdown that got n=1 returns.
- The countdown that got n=2 returns.
- The countdown that got n=3 returns.

And then you're back in __main__. So, the total output looks like this:

A function that calls itself is recursive; the process is called recursion.

As another example, we can write a function that prints a string n times.