

## Expected Output:

Navigation

EDIT

Designed By: Khue, Rahul, Grant, Shari

HACKBOOK

COVER PAGE DUBHACKS2020

### 5.2 Boolean expressions

A **boolean** expression is an expression that is either true or false.

The following examples use the operator `==`, which compares two operands and produces **True** if they are equal and **False** otherwise:

Python 3.6

```
1 print(5 == 5)
2 print(5 == 6)
```

Print output (drag lower right corner to resize)

Frames Objects

Step 1 of 2

Rendered by Python Tutor

Page Content

EDIT

SETTINGS

**True** and **False** are special values that belong to the type **bool**; they are **not strings**:

Python 3.6

```
1 print(True)
2 print(type(True))
3 print(type(False))
```

Print output (drag lower right corner to resize)

Frames Objects

Step 1 of 3

Rendered by Python Tutor

[Customize visualization \(NEW!\)](#)

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

Python 3.6

```
1 x = 2
2 y = 3
3
4 print(x != y) # x is not equal to y
5 print(x > y) # x is greater than y
6 print(x < y) # x is less than y
7 print(x >= y) # x is greater than or equal to y
8 print(x <= y) # x is less than or equal to y
```

Print output (drag lower right corner to re-size)

Frames Objects

line that just executed  
next line to execute

Step 1 of 7

Rendered by [Python Tutor](#)  
[Customize visualization](#) (NEW!)

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

SAVE CANCEL

Editing 'Page Content'

## 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:

Python 3.6

```
1 def countdown(n):
2     if n <= 0:
3         print('Blastoff!')
4     else:
5         print(n)
6         countdown(n-1)
```

line that just executed

next line to execute

< Prev

Next >

Step 1 of 1

Frames

Objects

Rendered by [Python Tutor](#)  
[Customize visualization](#) (NEW)

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?

```
Python 3.6
1 def countdown(n):
2     if n <= 0:
3         print('Blastoff!')
4     else:
5         print(n)
6         countdown(n-1)
7
8
9     countdown(3)

==> line that just executed
-> next line to execute
```

Print output (drag lower right corner to resize)

Frames Objects

Step 1 of 21

Rendered by Python Tutor  
[Customize visualization \(NEW\)](#)

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.

```
Python 3.6
1 def print_n(s, n):
2     if n <= 0:
3         return
4     print(s)
5     print_n(s, n-1)

==> line that just executed
-> next line to execute
```

Frames Objects

Step 1 of 1

Rendered by Python Tutor  
[Customize visualization \(NEW\)](#)

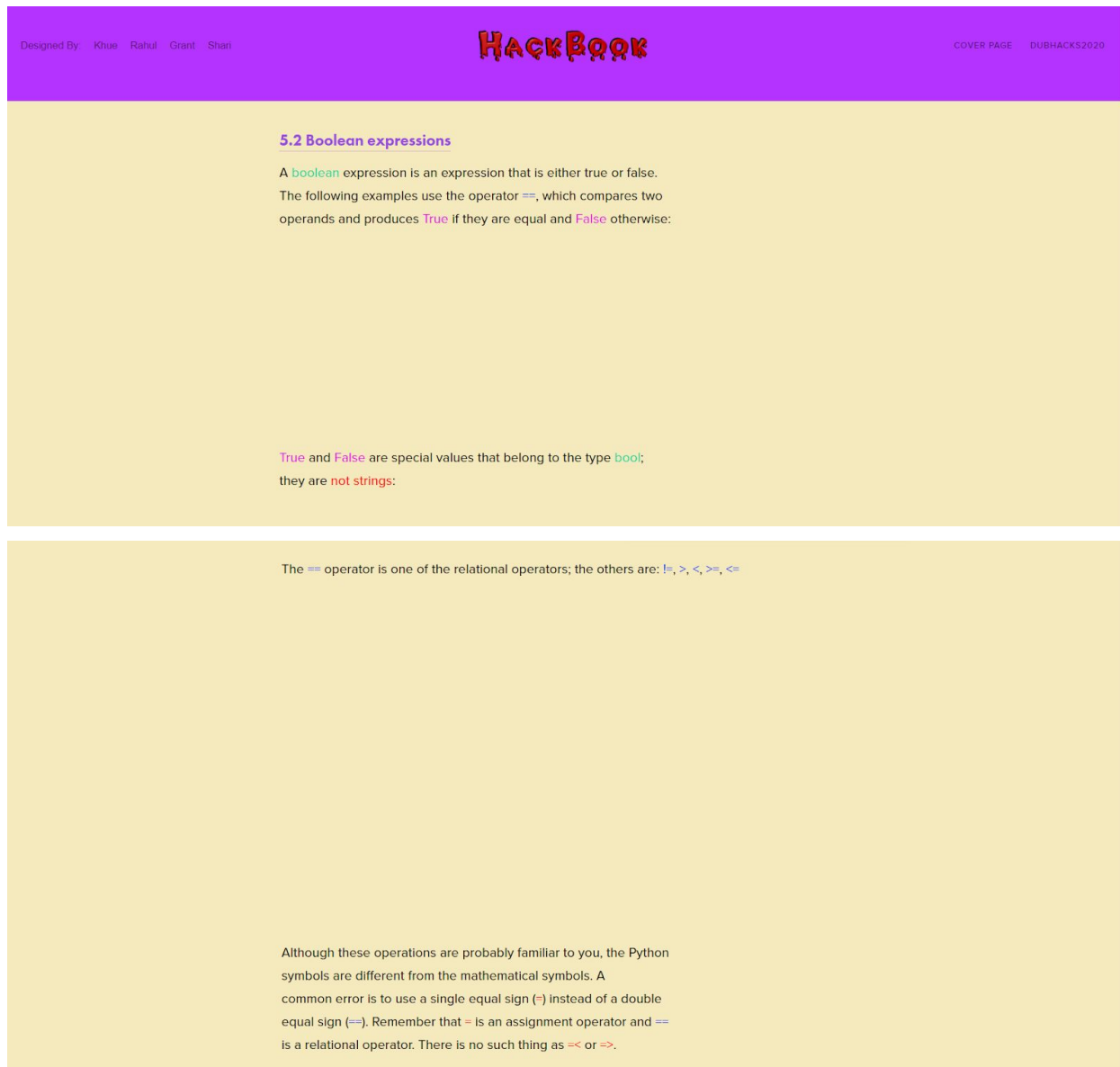
If  $n \leq 0$  the `return` statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

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  $n=0$ , 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.