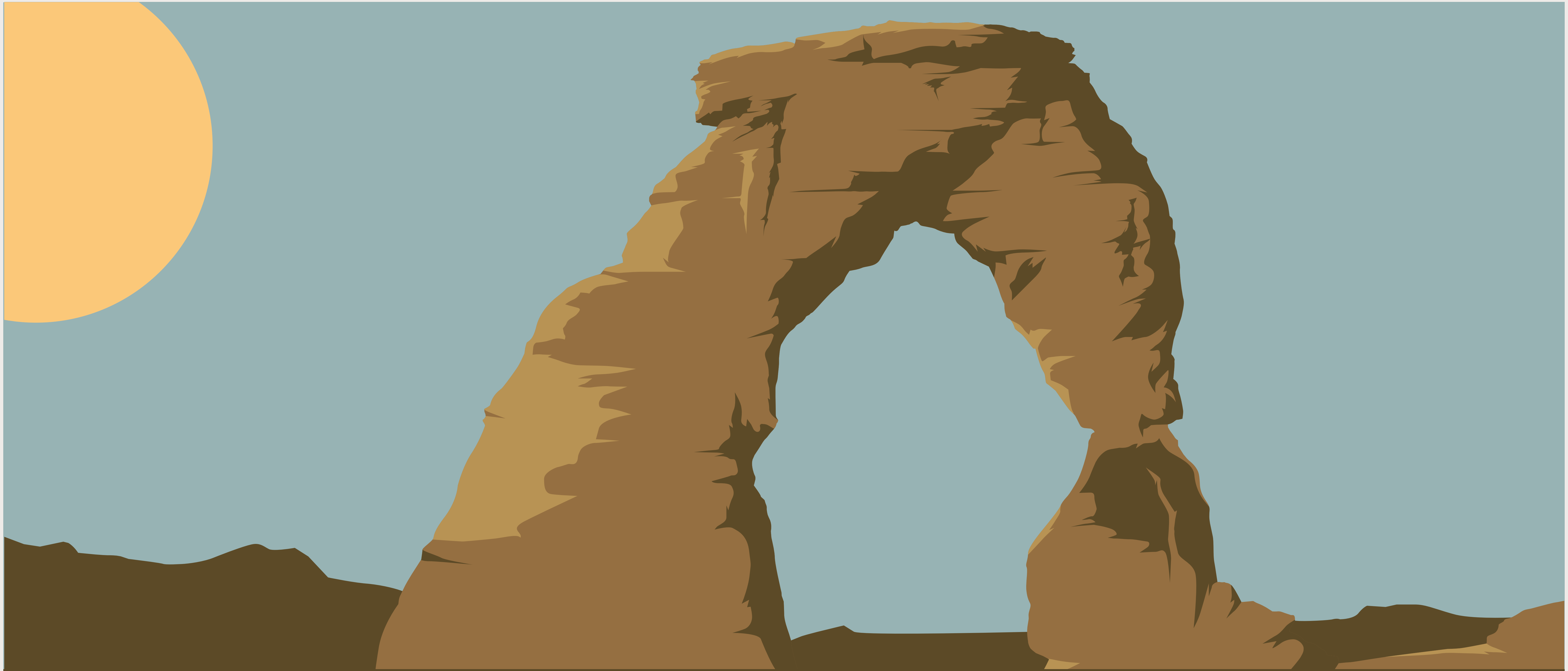


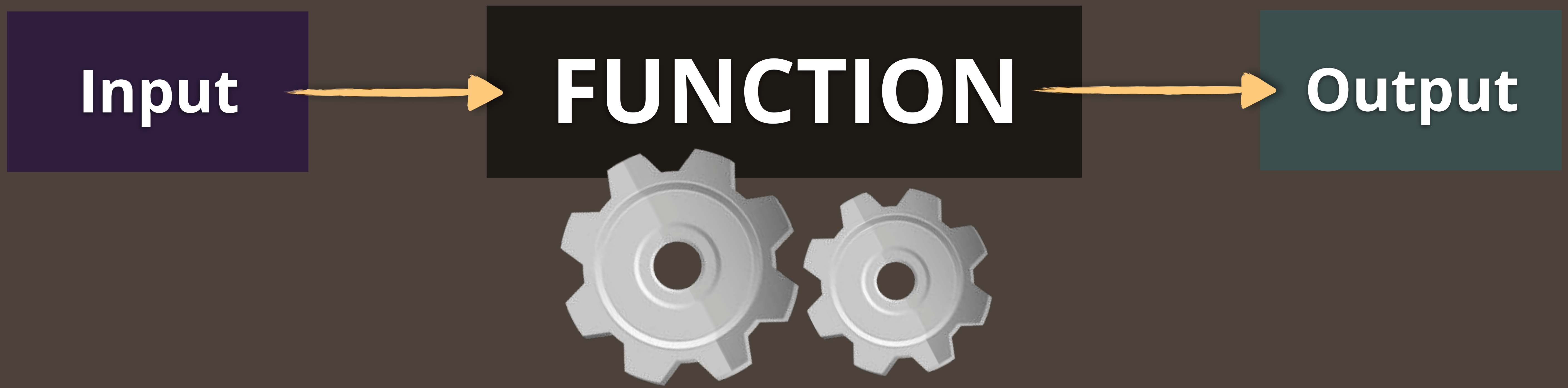
Experience

THE DESERT OF DECLARATIONS

LEVEL 4

THE DESERT OF DECLARATIONS

WHAT'S A FUNCTION FOR?



Input

FUNCTION

Output

Give the function some input...

...it does some stuff to or with the input...

...and it outputs some result.

FUNCTIONS SOLVE PROBLEMS

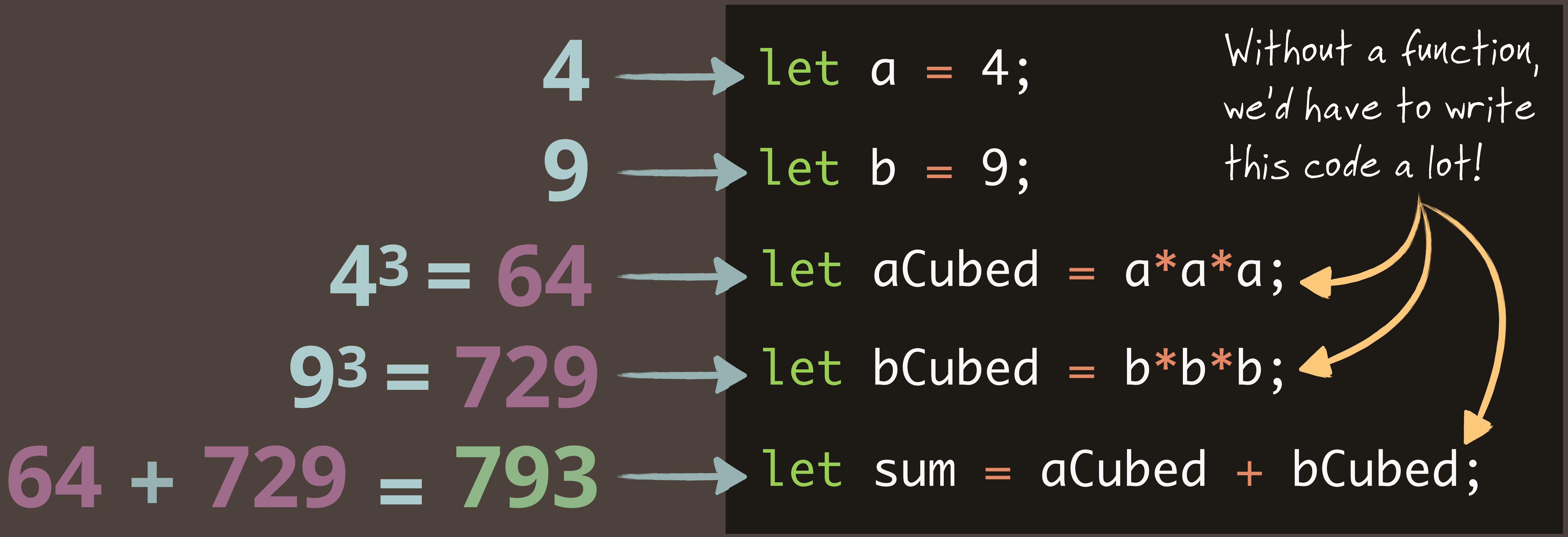
A function “does something” step-by-step that we need to do repeatedly

FUNCTION: The Sum of Two Cubes

1. Get two numbers $4 \qquad 9$
2. Cube each number $4^3 = 64 \qquad 9^3 = 729$
3. Sum the cubes $64 + 729 = 793$
4. Return the answer $\boxed{793}$

WHAT ARE THESE STEPS IN CODE?

Syntax for finding a sum of cubes



4

9

$4^3 = 64$

$9^3 = 729$

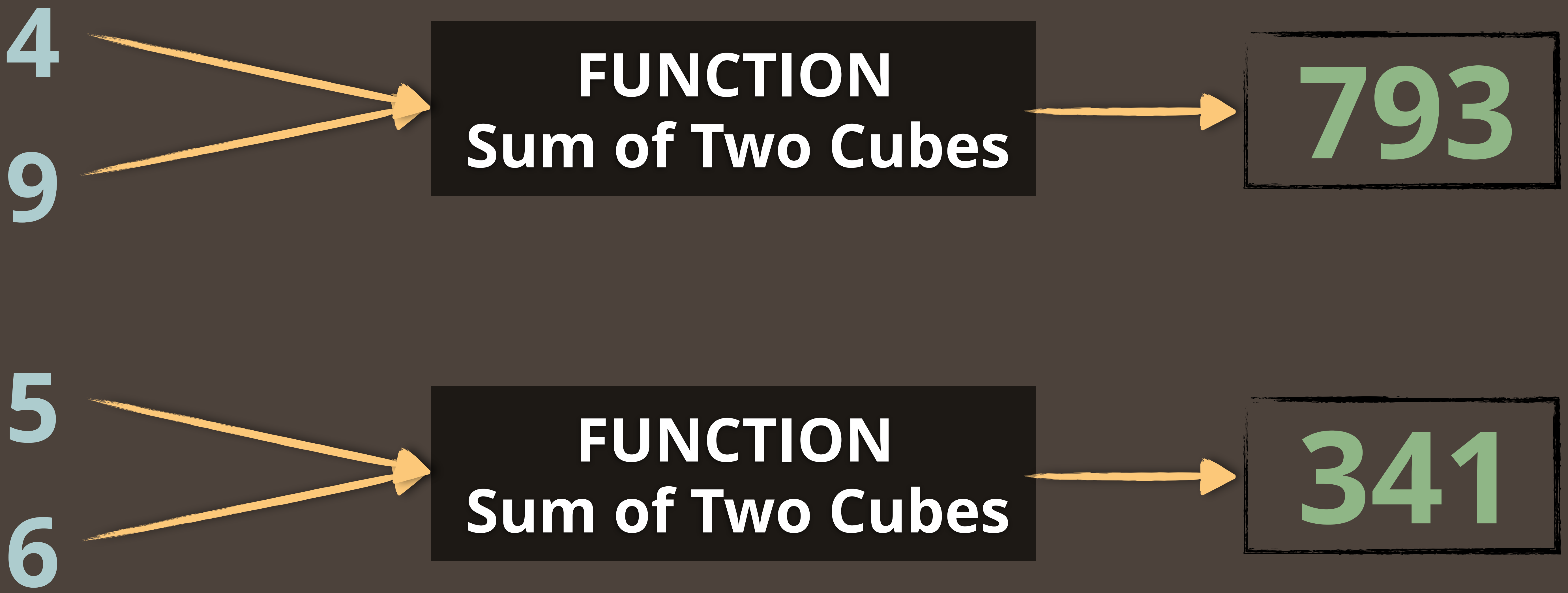
$64 + 729 = 793$

```
let a = 4;

let b = 9;

let aCubed = a*a*a;

let bCubed = b*b*b;

let sum = aCubed + bCubed;
```

Without a function, we'd have to write this code a lot!

USEFULNESS THROUGH REUSABILITY

Wrapping our code in a function will allow us to reuse it



4

9

FUNCTION
Sum of Two Cubes

793

5

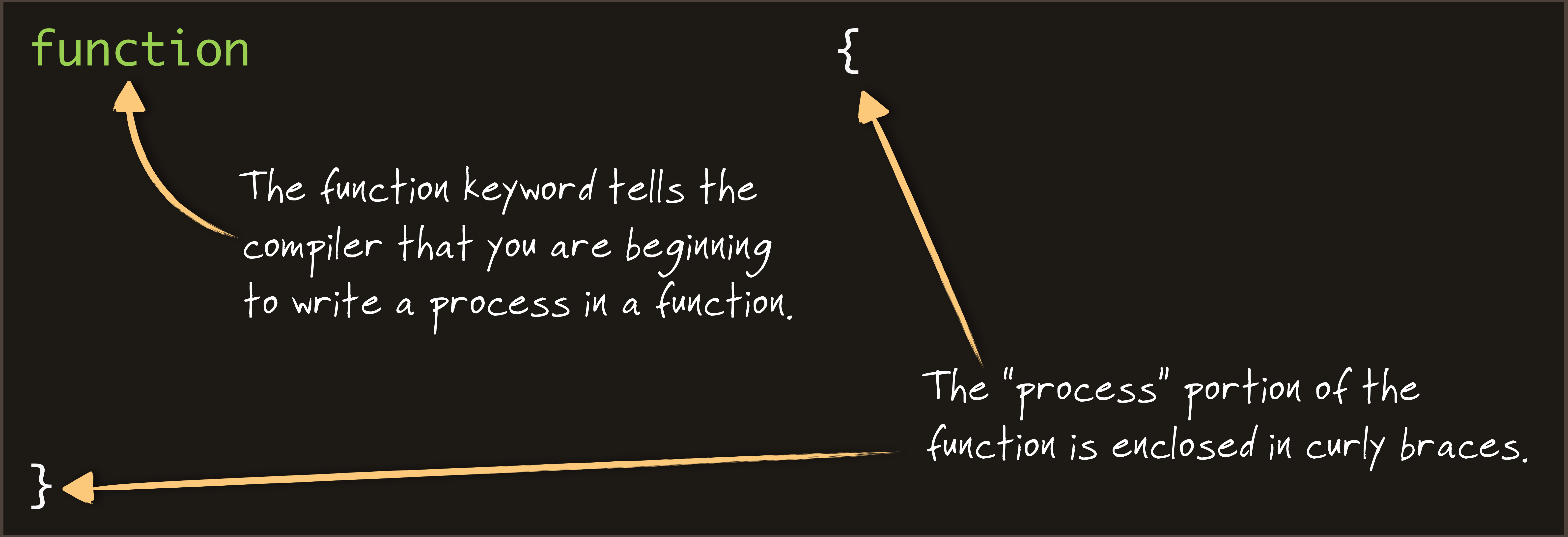
6

FUNCTION
Sum of Two Cubes

341

FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure



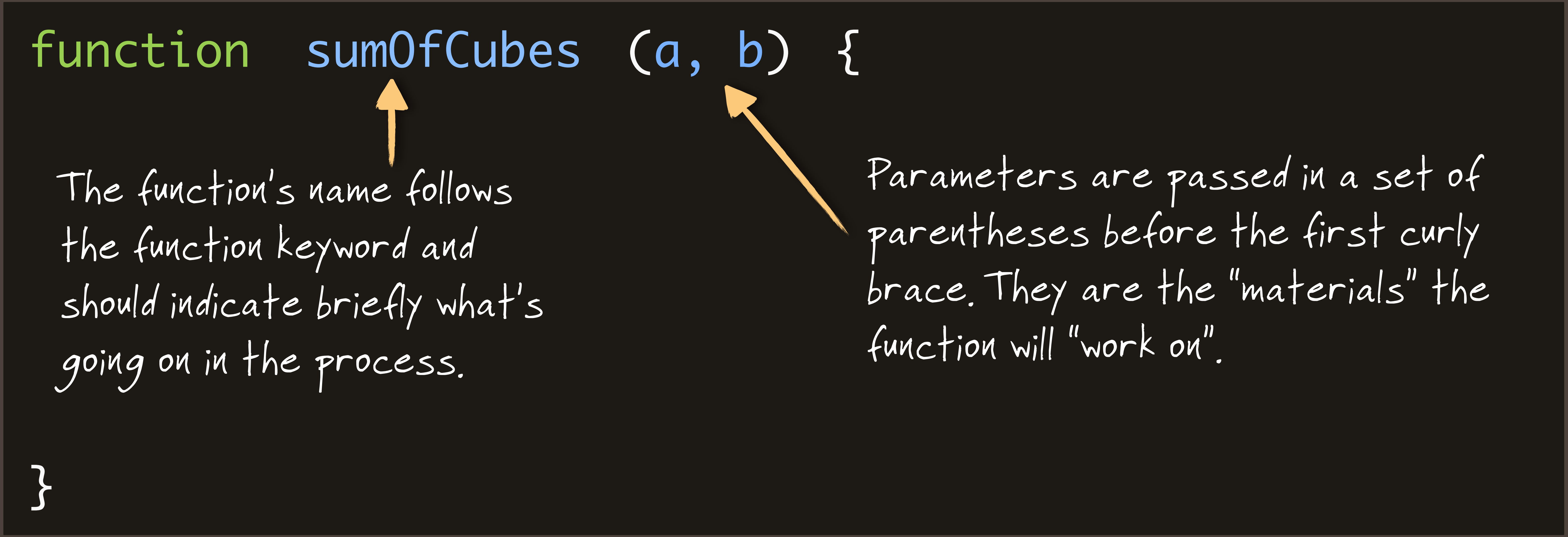
```
function                          {



}
```

The function keyword tells the compiler that you are beginning to write a process in a function.

The "process" portion of the function is enclosed in curly braces.

FUNCTIONS IN JAVASCRIPT CODE

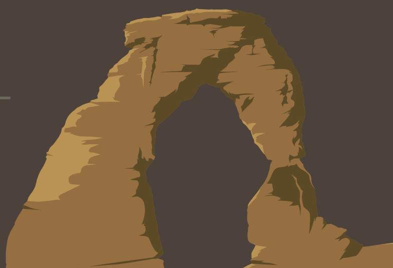
The syntax for a basic function structure



```
function  sumOfCubes  (a, b)  {



}
```

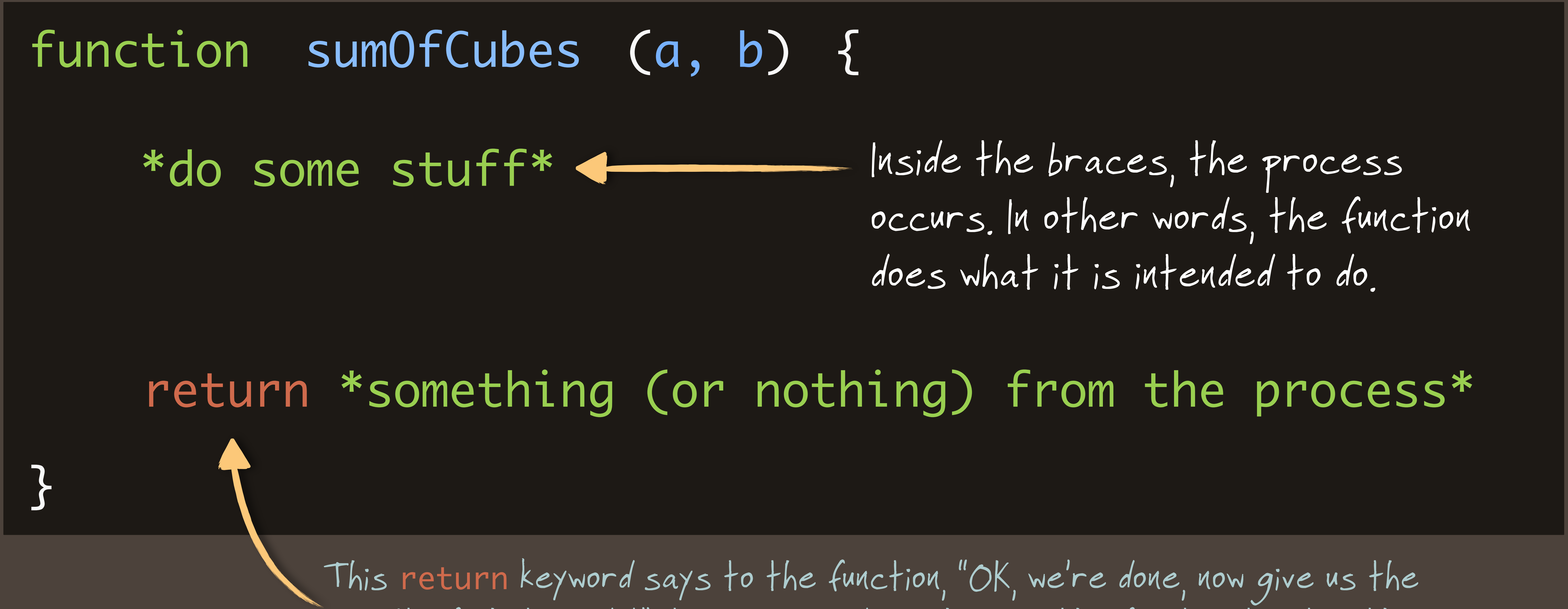
The function's name follows the function keyword and should indicate briefly what's going on in the process.

Parameters are passed in a set of parentheses before the first curly brace. They are the "materials" the function will "work on".



FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure



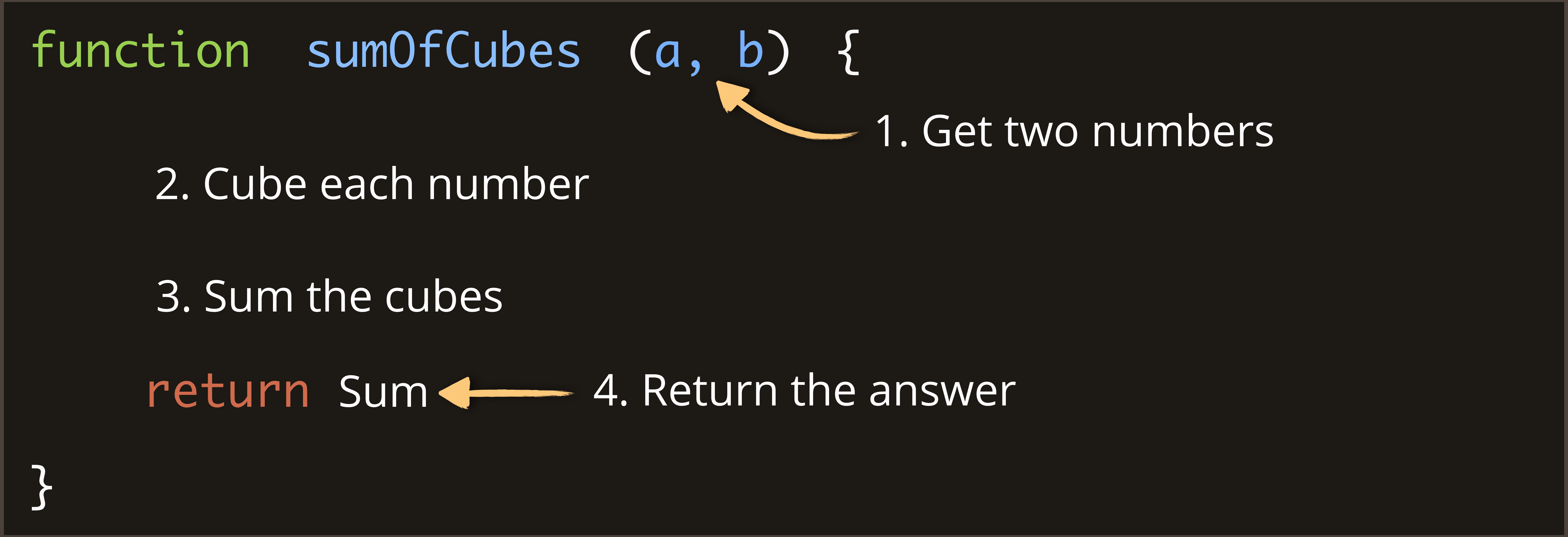
```
function  sumOfCubes  (a, b)  {

    *do some stuff*

    return *something (or nothing) from the process*
}
```

Inside the braces, the process occurs. In other words, the function does what it is intended to do.

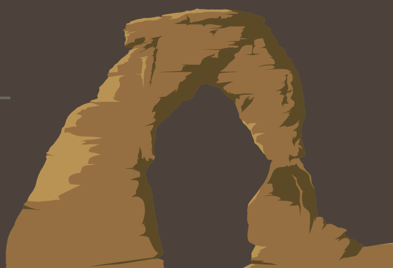
This return keyword says to the function, "OK, we're done, now give us the result of what we did." It can be used anywhere in the function to stop the function's work. Here, that happens to be at the very end.

BUILDING OUR SUMOFCUBES FUNCTION

Assigning steps of the process to the function syntax



```
function  sumOfCubes  (a, b)  {
                                          1. Get two numbers
        2. Cube each number

        3. Sum the cubes

       return  Sum             4. Return the answer

}
```

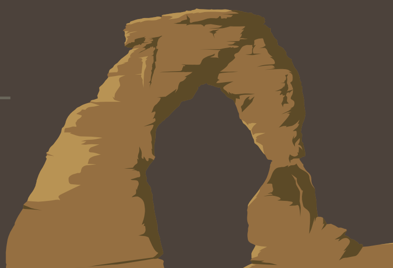


BUILDING OUR SUMOFCUBES FUNCTION

Assigning steps of the process to the function syntax

```
function  sumOfCubes  (a, b)  {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    let sum = aCubed + bCubed;

    return sum;

}
```

Once the parameters are passed into the function, they are accessible at any point within the process.



CALLING OUR SUMOFCUBES FUNCTION

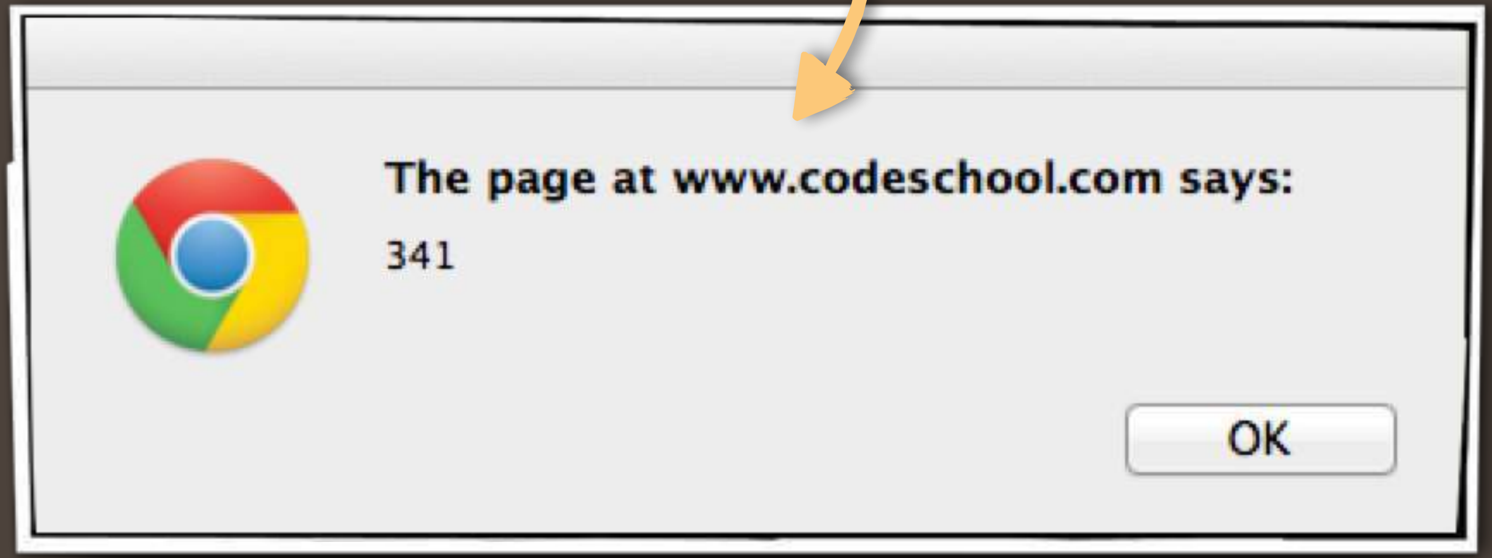
Now we can call the function using any parameter values we want!

```
function  sumOfCubes  (a, b)  {

   let aCubed = a*a*a;
   let bCubed = b*b*b;
   let sum = aCubed + bCubed;

   return sum;

}
```

```
sumOfCubes(4, 9);
```

→ 793

```
let mySum = sumOfCubes(5, 6);
alert(mySum);
```



The page at www.codeschool.com says:

341

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```javascript
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    let sum = aCubed + bCubed;
    return sum;
}
```

Our function does what it is supposed to, but it's not as efficient as it could be memory-wise. We've made three unnecessary variables that all have to be allocated in memory.

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

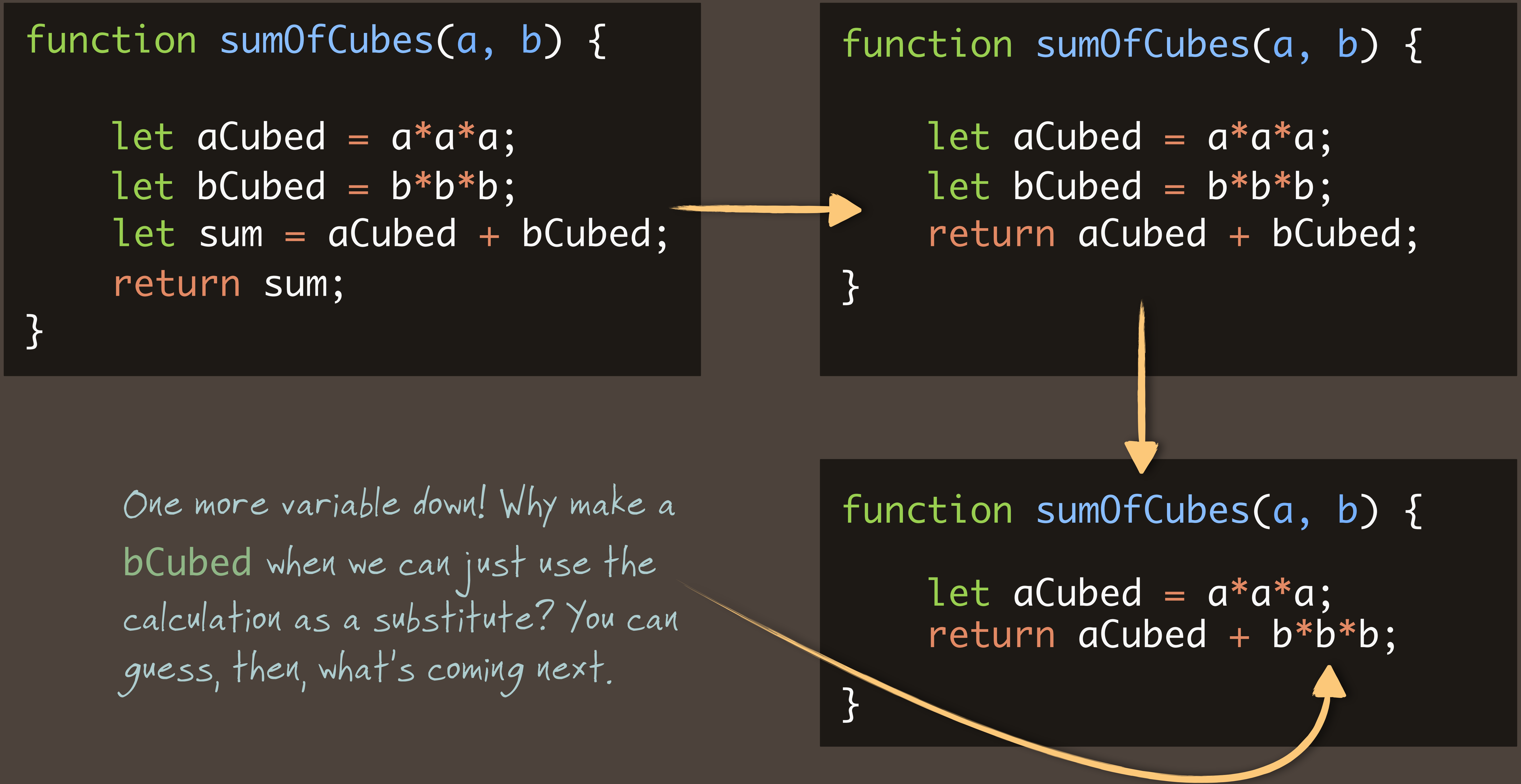
```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    let sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    return aCubed + bCubed;
}
```

The return keyword can calculate the results of an expression before actually returning from the function. One variable down!

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations



```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    let sum = aCubed + bCubed;
    return sum;
}
```

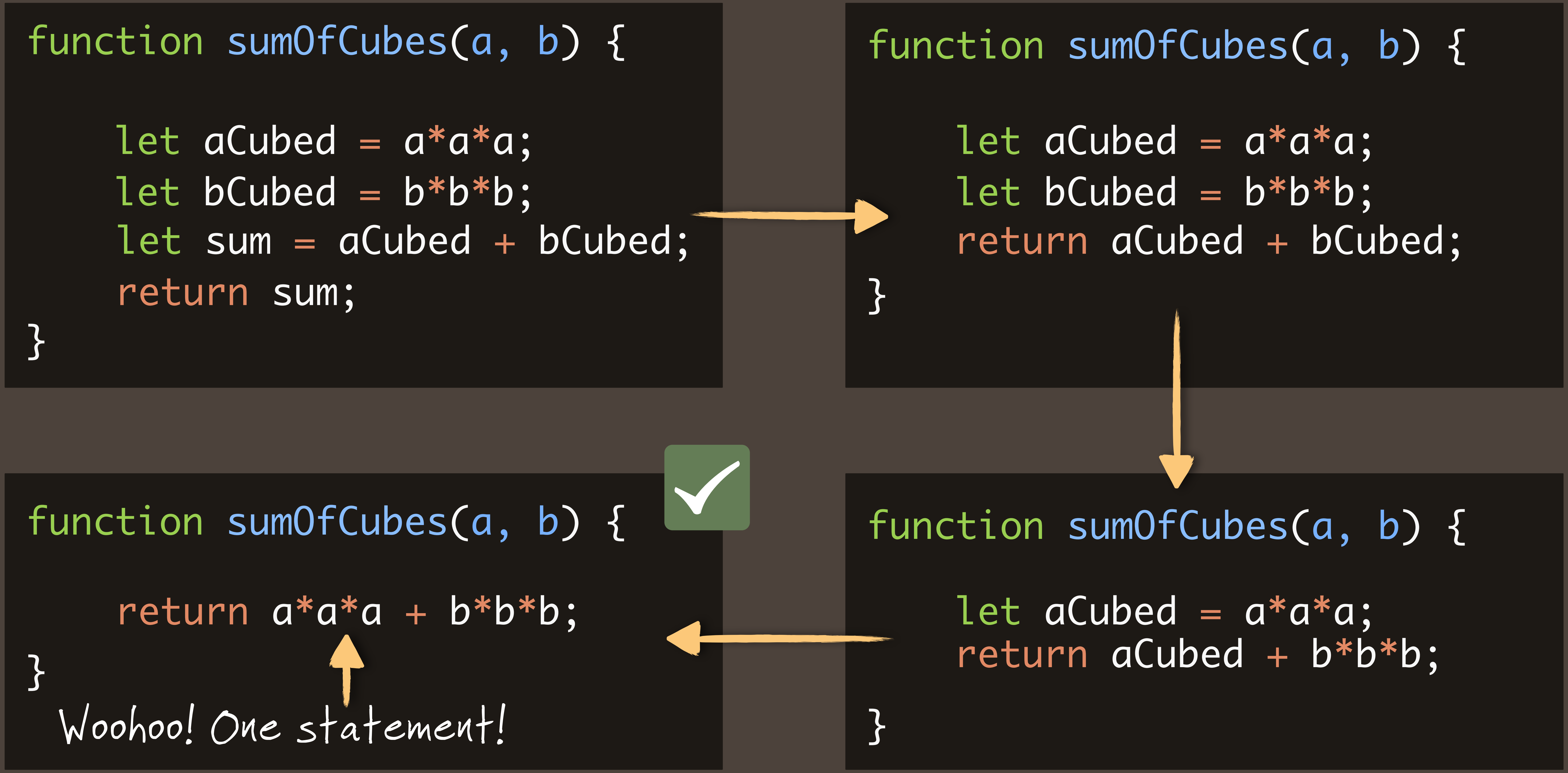
```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    return aCubed + bCubed;
}
```

```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    return aCubed + b*b*b;
}
```

One more variable down! Why make a bCubed when we can just use the calculation as a substitute? You can guess, then, what's coming next.

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations



```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    let sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    let bCubed = b*b*b;
    return aCubed + bCubed;
}
```

```
function sumOfCubes(a, b) {

    let aCubed = a*a*a;
    return aCubed + b*b*b;

}
```

✓

```
function sumOfCubes(a, b) {

    return a*a*a + b*b*b;
}
```

Woohoo! One statement!

OUR FUNCTION IN ACTION

Calling a function involves the function name and some parameters

```
function sumOfCubes(a, b) {
     return a*a*a + b*b*b;
}
```

✓

```
sumOfCubes(4, 9);
```

→ 793

Parameters can also be expressions, which the function will resolve before starting:

```
sumOfCubes(1+2, 3+5);
```

→ 539

Same as (3, 8)

```
let x = 3;
sumOfCubes(x*2, x*4);
```

→ 1494

Same as (6, 12)

NOW FOR A MORE COMPLEX FUNCTION!

Let’s design a function that counts “E’s” in a user-entered phrase

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
          *alert the user*
          *exit function with a failure report*

    }




}
```

By the way, sometimes functions don’t need any parameters!

Always check that a user input is valid before any operations

Using the return keyword here will allow us to exit and inform the program of an invalid entry.

NOW FOR A MORE COMPLEX FUNCTION!

Let’s design a function that counts “E’s” in a user-entered phrase

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

    }*otherwise*{




    }
}
```

This block will be where the function begins to actually check the phrase out and count the E's.

NOW FOR A MORE COMPLEX FUNCTION!

Let’s design a function that counts “E’s” in a user-entered phrase

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*
    }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                  *if the character is an 'E' or an 'e'*{
                        *increment the E counter*
                  }
            }
            *alert the amount of E's in the phrase and return success*
    }
}
```

We have to count lowercase as well as uppercase!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

    }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*
    }
}
```

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     let phrase = prompt("Which phrase would you like to examine?");
     *if the entry is invalid*{
          *alert the user*
          *exit function with a failure report*

     }*otherwise*{
          *make a counter for the E's*
          *for each character in the user's entry*{
               *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
               }
          }
          *alert the amount of E's in the phrase and return success*
     }
}
```

The prompt() method helps us get the user's entry.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {

    }
     *otherwise*{
          *make a counter for the E's*
          *for each character in the user's entry*{
               *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
               }
          }
          *alert the amount of E's in the phrase and return success*
    }
}
```

The typeof keyword allows us to determine whether the user has entered a valid string. This != expression returns true or false.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we’ve described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
          alert("That's not a valid entry!");
          return false;
    }
     *otherwise*{
          *make a counter for the E's*
          *for each character in the user's entry*{
               *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
               }
          }
          *alert the amount of E's in the phrase and return success*
    }
}
```

If the entry is not a string, we alert the user and exit the function, returning false.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
            alert("That's not a valid entry!");
            return false;
    } else {

            *make a counter for the E's*
            *for each character in the user's entry*{
                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*

    }
}
```

Else-blocks help us do the "otherwise" case!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
            alert("That's not a valid entry!");
            return false;
    } else {

            let eCount = 0;
            for (let index = 0; index < phrase.length; index++) {

                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*

    }
}
```

We want to start at index 0, and go until one less than the length of the user's string. Remember that strings have zero-based indices!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
          alert("That's not a valid entry!");
          return false;
    } else {

          let eCount = 0;
          for (let index = 0; index < phrase.length; index++) {
              if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E'){
                  eCount++;

              }
          }
          *alert the amount of E's in the phrase and return success*
    }
}
```

This complex conditional checks whether the spot we're currently at along the string is either an E or an e.

If we found one, we'll increase our counter.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
        alert("That's not a valid entry!");
        return false;
    } else {

        let eCount = 0;
        for (let index = 0; index < phrase.length; index++) {
            if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E'){
                eCount++;
            }
        }

        *alert the amount of E's in the phrase and return success*
    }
}
```

This complex conditional checks whether the spot we're currently at along the string is either an E or an e.

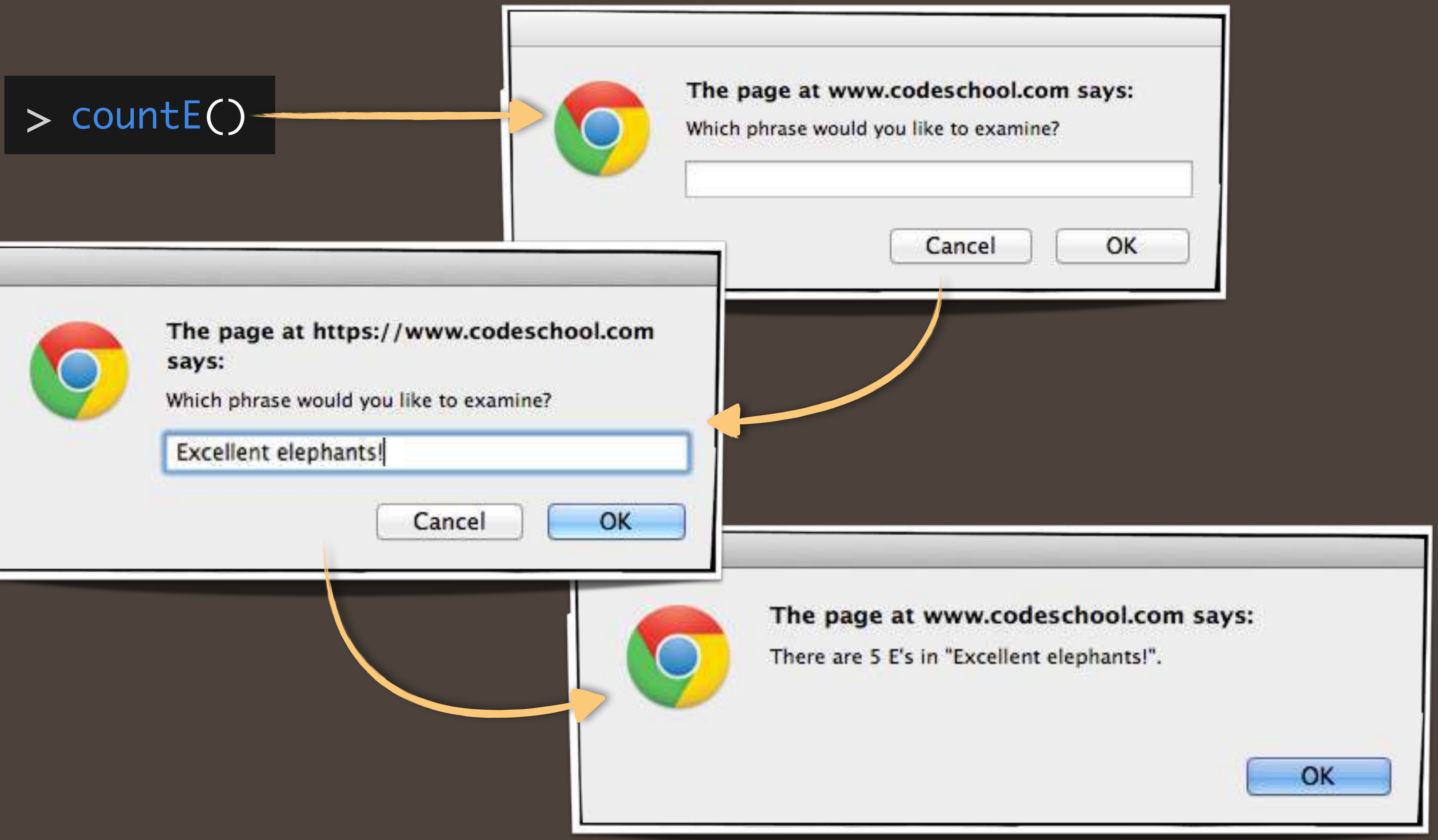
FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    let phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
        alert("That's not a valid entry!");
        return false;
    } else {

        let eCount = 0;
        for (let index = 0; index < phrase.length; index++) {
            if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E'){
                eCount++;
            }
        }
        alert("There are " + eCount + " E's in \"" + phrase + "\".");
        return true;
    }
}
```

After our for loop, eCount will contain the total number of E's and e's in our loop.

THE SEQUENCE OF ENTRY



```
> countE()
```

The page at www.codeschool.com says:

Which phrase would you like to examine?

The page at https://www.codeschool.com says:

Which phrase would you like to examine?

Excellent elephants!

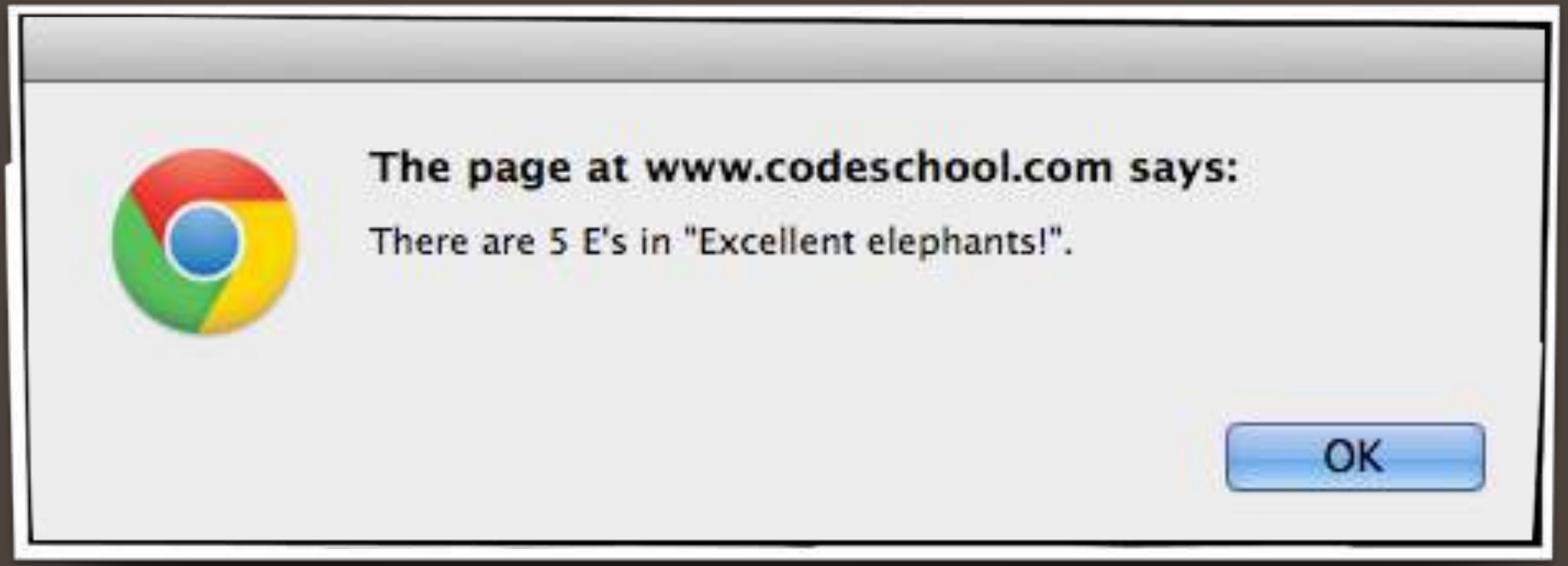
The page at www.codeschool.com says:

There are 5 E's in "Excellent elephants!".

TRACING OUR E-COUNTER

Following our function's code as it counts E's in "Excellent elephants!"

index	LOOP: index < length?	charAt (index)	is charAt(index) an E or e?	eCount
0	TRUE	E	TRUE	1
1	TRUE	x	FALSE	1
2	TRUE	c	FALSE	1
3	TRUE	e	TRUE	2
4	TRUE	l	FALSE	2
5	TRUE	l	FALSE	2
6	TRUE	e	TRUE	3
7	TRUE	n	FALSE	3
8	TRUE	t	FALSE	3
9	TRUE	(space)	FALSE	3
10	TRUE	E	TRUE	4
11	TRUE	l	FALSE	4
12	TRUE	e	TRUE	5
13	TRUE	p	FALSE	5
14	TRUE	h	FALSE	5
15	TRUE	a	FALSE	5
16	TRUE	n	FALSE	5
17	TRUE	t	FALSE	5
18	TRUE	s	FALSE	5
19	TRUE	!	FALSE	5
20	FALSE	STOP!		

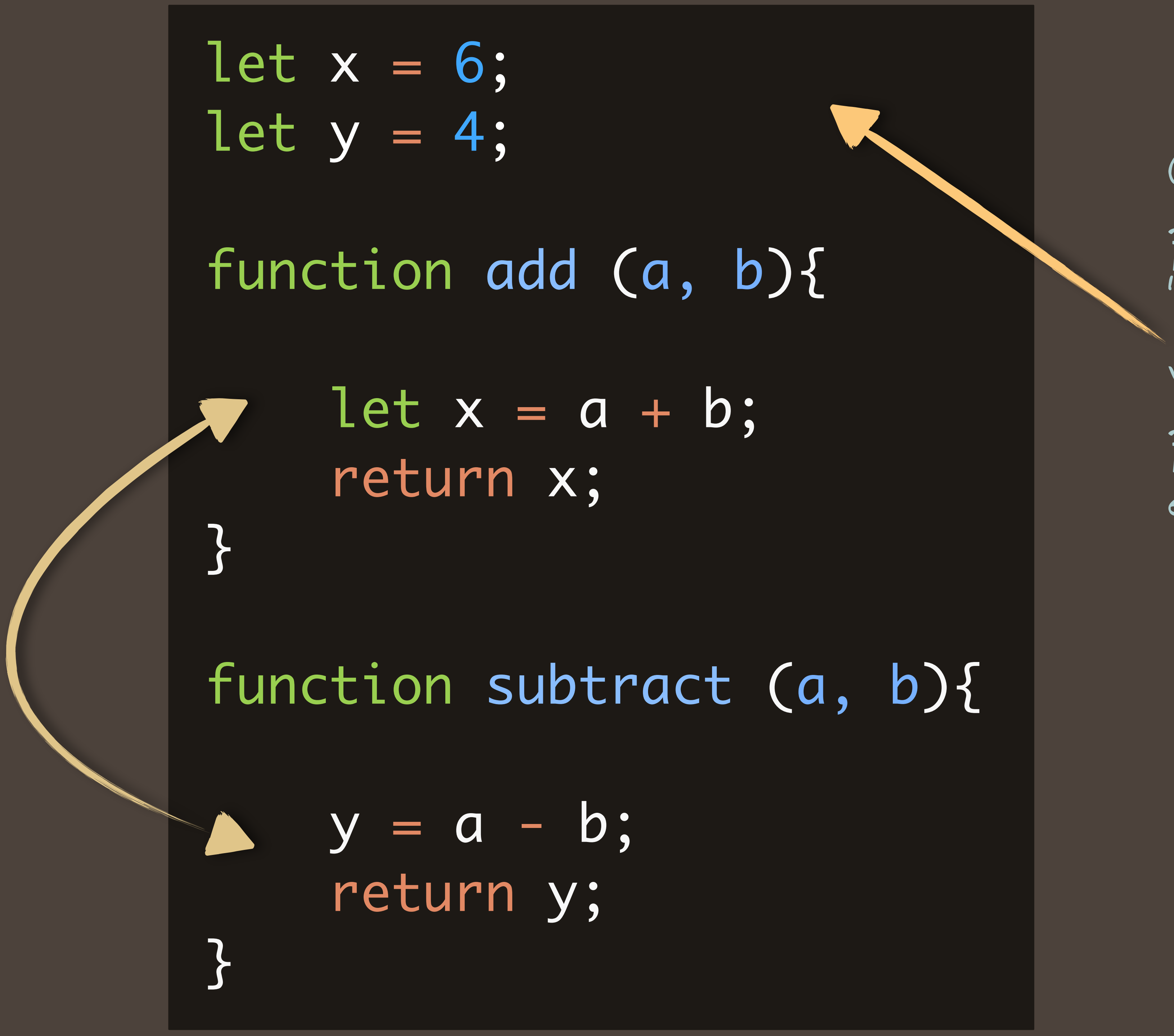


The page at www.codeschool.com says:

There are 5 E's in "Excellent elephants!".

UNDERSTANDING LOCAL AND GLOBAL SCOPE

Visualizing worlds within worlds...



```
let x = 6;
let y = 4;

function add (a, b){

    let x = a + b;
    return x;
}

function subtract (a, b){

    y = a - b;
    return y;
}
```

Inside functions, the scope is "local", like cities within a state. Each has their own "government" and stuff that happens in here stays in here.

Out here, in the main program, the scope is "global", which means that variables declared are potentially accessible from everywhere.

FUNCTIONS CREATE A NEW SCOPE

Variables declared in a function STAY in the function

```
let x = 6
function add (a, b){

    let x = a + b;
    return x;
}
```

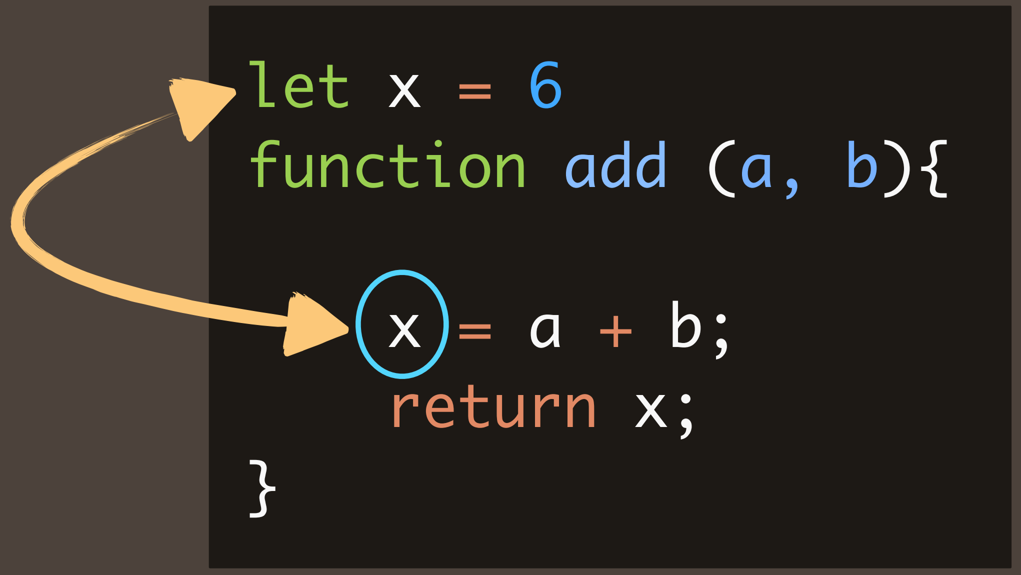
```
add(9, 2);
```

→ 11

```
console.log(x)
```

→ 6

The circled variable only exists in the function's local scope. Because it has been declared with var, it doesn't modify the same-named variable "outside" the function.



```
let x = 6
function add (a, b){

    x = a + b;
    return x;
}
```

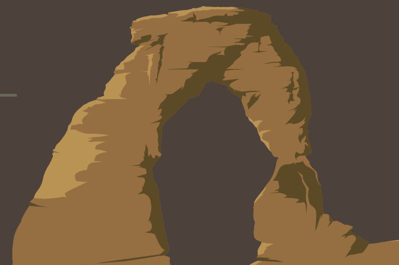
```
add(9, 2);
```

→ 11

```
console.log(x)
```

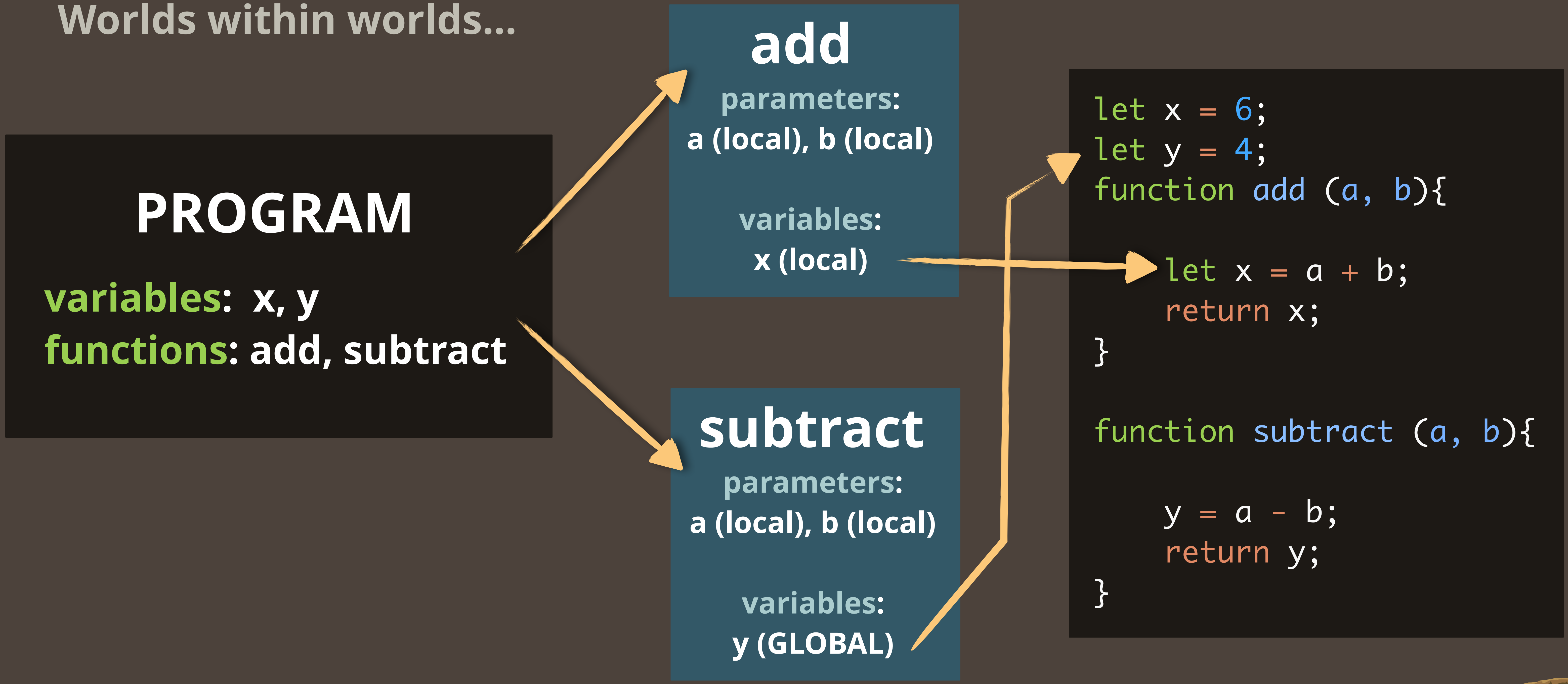
→ 11

If the x were not declared with var, it would "shadow" the same-named variable from the nearest external scope!



VISUALIZING LOCAL AND GLOBAL SCOPE

Worlds within worlds...



PROGRAM

variables: x, y

functions: add, subtract

add

parameters:
a (local), b (local)

variables:
x (local)

subtract

parameters:
a (local), b (local)

variables:
y (GLOBAL)

```
let x = 6;
let y = 4;
function add (a, b){

    let x = a + b;
    return x;
}

function subtract (a, b){

    y = a - b;
    return y;
}
```