

Just like how variables can store data and be reusable, functions can also store data and be reusable:

```
// function definition

function catGreeter() {
  console.log("Hey Cat! You're a fine animal!")
  console.log("Meooowwwwwww!!!")
}

// run, call, invoke, execute
catGreeter()
```

```
>
Hey Cat! You're a fine animal!
Meooowwwwwww!!!
```

... as you can see, we don't need to write `console.log()` to print out those statements:

```
// run, call, invoke, execute
catGreeter()
catGreeter()
catGreeter()
catGreeter()
```

```
>
Hey Cat! You're a fine animal!
Meooowwwwwww!!!
Hey Cat! You're a fine animal!
Meooowwwwwww!!!
Hey Cat! You're a fine animal!
Meooowwwwwww!!!
Hey Cat! You're a fine animal!
Meooowwwwwww!!!
```

... so to summarize what we did, we had 2 steps:

- (1) We defined the function
- (2) We executed the function

Another example:

```
function specialGreeter(name) {  
  console.log("Hey " + name + "! You have a cool attitude.");  
}  
  
specialGreeter("Joe")
```

```
Hey Joe! You have a cool attitude.  
undefined
```

... so to summarize what we just did:

- (1) We defined the function with **name** as the parameter.
- (2) We executed the function with **Joe** as the argument.

Another example:

```
function adder(a, b, c, d) {  
  console.log(a + b + c + d);  
}
```

```
adder(2, 2, 2, 2)
```

A dark-themed terminal window showing the output of the function call. The number 8 is displayed in white text, preceded by a small orange cursor icon.

8

We have been using `console.log()` to execute arguments; now we will use `return` :

```
function adder(num1, num2) {  
  return num1 + num2;  
}  
  
adder(2, 4)
```

A terminal window with a dark background. It shows a prompt character (a small orange icon) followed by the number 6, which is the result of the adder function call.

... the `return` statement actually gives us the real output of the function parameters. The `return` statement cannot be executed unless it is within a defined function.

The `console.log()`, however, is independent of any function:

```
console.log("hello")
```

A terminal window with a dark background. It shows a prompt character (a small orange icon) followed by the string "hello", which is the output of the console.log statement.

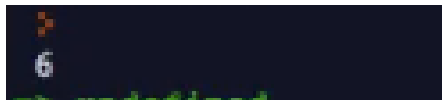
.... we can't do the same thing with `return` :

```
return "hello"
```

A terminal window with a dark background. It shows a prompt character (a small orange icon) followed by a red error message: "SyntaxError: 'return' outside of function (37:0)".

Further example to show difference between `console.log()` and `result` :

```
function adder(num1, num2) {  
    return num1 + num2;  
}  
  
adder(2, 4)  
  
var result = adder(2, 4);  
  
console.log(result)
```

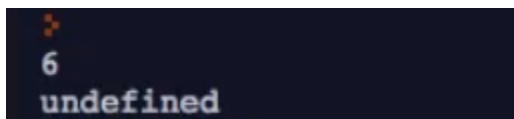


```
>  
6
```

... we get 6 as the result, because `return num1 + num2` outputs the result of 6. But if we write this:

```
function adder(num1, num2) {  
    console.log( num1 + num2);  
}  
  
var result = adder(2, 4);  
  
console.log(result)
```

... we get this:



```
>  
6  
undefined
```

... this is because `console.log(num1 + num2)` doesn't really output the result. It just displays it.

... so the point of all this is to demonstrate why it's best practice to use **return** inside of functions.

Let's try a function with an Array:

```
function doesExist(nums, num) {  
  for (var i=0; i<nums.length; i++) {  
    if (nums[i] === num) {  
      return true;  
    }  
  }  
  
  return false;  
}  
  
doesExist([2, 2, 5, 7], 8) // true
```

... basically we are checking to see if **num** (8) is found within **nums** [2, 2, 5, 7]... if it is found there, we can return true... otherwise it will be false... what is the result?

```
>  
=> false
```

But if we change **num** to 2, what is the result?

```
function doesExist(nums, num) {  
  for (var i=0; i<nums.length; i++) {  
    if (nums[i] === num) {  
      return true;  
    }  
  }  
  
  return false;  
}  
  
doesExist([2, 2, 5, 7], 2) // true
```

```
>  
=> true
```

... we get true, because there IS a 2 within **nums**!