In mathematics, a function is defined as a relationship between defined values and one or more variables. In JavaScript, a function is defined as a relationship between defined values and one or more bindings. That said, the output of a function can result in an entirely new value, which itself can be the binding for a new function, ad infinitum (or until you run out of memory!) Don't get caught up on the definition of a function, but instead try and think of them as defining new vocabulary to the language. Functions in practice are really just used to organize chunks of code that can be reused repeatedly.

For example, you already have been working with this function:

```javascript
console.log();
```

This function (defined for our nodejs environment here) prints out to the console based on the arguments we've been giving it. In some cases, we will give an argument in the form of a string:

```javascript
console.log('double rainbow');
//double rainbow
```

The string `'double rainbow` is the argument we're passing to the console.log() function. Other times, we pass a binding as an argument:

```javascript
const catName = 'patches'
console.log(`hi ${catName}!`);
```

In this case, we are passing both a binding `catName` and the strings `'hi` and `!'`. We can also pass a function as an argument (passing functions to functions is very cool in JavaScript):
Yes, you can do that. We just made the computer do some serious calculations. Recall in the previous lab that some us wrote the following in order to calculate $2^{64}$:

```javascript

But some of us found a built-in function:

```javascript
console.log(Math.pow(2, 64));
```

Now we see that there’s even a way to create our own function to perform this:

```javascript
const power = function(base, exponent) {
  let result = 1;
  for (let i=0; i < exponent; i++) {
    result *= base;
  }
  return result;
};

console.log(power(2, 10));
// 1024
```

**CHALLENGE #1:** Comment the function above so that a layperson can understand what it does, include the portion of the code that calls the console.log() function as well.

---

**Built-In Functions**

Now is a good time to talk about built-in functions in popular JavaScript runtime environments (such as Chrome's or NodeJS). We've typed `Math.something()` a couple times already without really knowing where it came from, but you probably assumed it didn't come from thin air :)

Let’s suppose we wanted to create a simple function for customers of our coffee shop that gives them a chance to win a free coffee, we want this function to `return` a 'you win!' or 'sorry! next time' string based on the outcome of a random number being higher then some threshold (e.g., if we generate a number randomly between 0 and 100, we want to specify a threshold, say, 90 where the function will return the string 'you win!').
Check out the Math Object here (note: we haven’t talked about Objects, but functions in JavaScript are Objects, this is because they have properties and functions, don’t get too confused, you’ll learn to love objects like family soon enough).

So, we found this function, seems like a start.

```javascript
Math.random(); //returns a random number
```

Creating our code block

Functions in JavaScript generally have code blocks that are enclosed in two brackets `{ }`. This allows us to put multiple statements inside of a single block of code (we’ve done this many times already!).

`Math.random()` returns a number between 0 and 1, so we need to multiply the result by 100 in order to get a number between 0 and 100.

```javascript
console.log(Math.random() * 100));
//50.8459349606934
```

Those trailing decimals are not cool (we want to eventually create a spinning wheel graphic that shows whole numbers).
CHALLENGE #2: Create a function chain (one function calling another) that takes the random number and converts it to a whole number between 0 and 100.

```javascript
console.log(Math.floor(Math.random() * 100));
```

Cool. You just used JavaScript's ability to create **higher-order** functions. A higher-order function is a function that can take another function as an argument, or that returns a function as a result.

Now, we have all of the tools we need to create a lucky draw program!

---

**---solution below---**

```javascript
let randomNumber = Math.floor(Math.random() * 100);

if(randomNumber > 50) {
  console.log(`your number was ${randomNumber}, you win!`);
} else {
  console.log(`your number was ${randomNumber}, sorry!`);
}
```

**Defining a function**

We've gone so far without really telling you how you write a define a function (remember I mentioned that syntax is easy—it's a matter of practicing your typing). Let's change that.

You define a function syntactically in Javascript in a variety of ways. By defining a function, you are creating a binding where the value of the binding is a function. For example, this code defines `volumeOfCube` as a
function that gives us the volume of a cube of length \( \text{length} \) when we pass in the value for \( \text{length} \) as an argument.

```javascript
const volumeOfCube = function(length) {
  return length * length * length;
};

console.log(volumeOfCube(2));
// 8
```

**Declaring a function in Javascript**

Here are some (not all) of the ways you can define a function in Javascript.

```javascript
function A(x){}; // function declaration
let B = function(x){}; // function expression
const C = function makeCoffee(x){}; // named function expression
let D = x => x * 2; // ES6 arrow function
```

With the exception of the last function definition (which uses the arrow syntax) a function is created with an expression that starts with the keyword `function` (note that the last 3 create bindings to functions). Functions have a set of parameters (in this case, only \( x \)) and a body, which contains the statements that are to be executed when the function is called. The function body of a function created this way must always be wrapped in braces, even when it consists of only a single statement.

The last 3 functions are actually not functions, but `function expressions`. A `function expression` is similar to function declaration, with the exception that identifier can be omitted, creating an anonymous function. Function expressions are often stored in a binding. You can identify a function expression by the absence of a function name immediately trailing the function keyword.

```javascript
let B = function(x){}; // function expression
const C = function makeCoffee(x){}; // named function expression
let D = x => x * 2; // ES6 arrow function
```

A function can have multiple parameters or **no parameters at all**. In the following example, `runForCover` does not list any parameter names, whereas `makeCoffee` lists two:
let isBear = true;
let distance = 11;

const runForCover = function() {
    if (isBear && distance > 10) {
        return 'it\'s a bear, you should really get out of there!';
    } else if (isBear && distance < 10) {
        return 'nice knowing you';
    }
};

consloe.log(runForCover());

//it's a bear, you should really get out of there!

Function declaration syntax

Here is a bit more about the declaration syntax, you'll see versions of these in sample code all of the time.

Old School

function makeCoffee() {
    //write some code to make coffee

function wow(speed) { text: 'WOW' }
The function declaration above is old school Javascript. This creates a binding (variable) `A` which is accessible in the current scope. *JavaScript has two scopes—global and local.* Any variable declared outside of a function belongs to the global scope, and is therefore accessible from anywhere in your code. Each function has its own scope, and any variable declared within that function is only accessible from that function and any nested functions.

If you declare a function like this, remember:

- You always need to have an identifier (in our case 'A')
- You can’t use these functions inside if statements or loops

Actually, try and avoid using these, use *function expressions* instead

**Function Expressions**

```javascript
let makeCoffee = function(){
   //write some code to make coffee
};
```

A function expression looks similar to function declarations, *except that the function is assigned to a variable name.* Though functions are not primitive values in JavaScript (undefined, null, boolean, string and number) this is the way they can be utilized to their full effect in this functional language.

**Named function expression**

```javascript
let baristaJohn = function makeCoffee(){
   //write some code to make coffee
};
```

Named function expressions are basically just like a function expression, but instead of assigning the variable to an anonymous function, we’re assigning it to a named function (with the name `makeCoffee`).

- Notice we haven’t exposed our `makeCoffee` function to the global scope, it’s only available through `baristaJohn` binding.
- These are useful if you get errors in your code, the compiler will tell you the function name.

**Arrow Functions**
let D = x => x * 2;

// ES6 arrow function

Arrow functions are new, and pretty cool, well, they bring no brand new functionality but they look better syntactically and they can return without brackets if you only have one statement. Arrow functions are particularly handy in cases where we would’ve had one-line functions before, as in this case with ES5 JavaScript.

- They preserve their this context, which will be very useful to us later on.

Here’s an example of how we might use the arrow syntax in our coffee example (with additional switch statement thrown in!)

```javascript
let makeCoffee = (quantity, coffeeStrength) => {
    switch(coffeeStrength) {
        case 'Strong': {console.log(`grind ${(quantity * 5)} pounds of beans`); break;}
        case 'Medium': {console.log(`grind ${(quantity * 3)} pounds of beans`); break;}
        case 'Weak': {console.log(`grind ${(quantity * 1)} pounds of beans`); break;}
        default: console.log('please enter a valid pounds of coffeesStrength');}
    makeCoffee(3, 'Strong');
    //grind 15 pounds of beans
}
```

**Returning**

You've seen a few examples of a keyword return in some of our code.

So far, we've been using a lot of console.log() in our code. Using console.log() as the result of a function isn't the best use of a function. The purpose of a function is to take some input, perform some task on that input, then return a result.

To return a result, we can use the return keyword. Take a look at our function from the last exercise, now re-written slightly:

```javascript
function makeCoffee(quantity, coffeeStrength){
    beanAmount = quantity * (coffeeStrength * 5);
    return beanAmount
}

console.log(makeCoffee(5, 10));
```
What's going on here? Well, instead of calling the `console.log()` function, we simply return the value of `beanAmount`. That's all this function does. In order to see the value of `beanAmount`, we need to call the `console.log()` function and pass in the `makeCoffee` function with the two arguments we defined in the function (`quantity` and `coffeeStrength`).

---

CHALLENGE 3: Let's put together what we now know about creating functions and returning values. Your task is to create a function to annoy your roommate when he uses your computer.

It will accept a number (named `x`) and string (named `message`) as arguments, and use the `alert()` function of your web browser to display the message passed to it, with the message repeated `x` times.

Note (the `alert` function has some built-in spam protection, so don't make `x` larger than 10)

---A SOLUTION---:

```javascript
const annoyBob = function(x, message) {
  for(i=0;i<x;i++) {
    message = message + '\n';
    alert(message);
  }
}
annoyBob(10,'Hi!');
```

Biding and scope

Global Scope

Each binding has a scope, which is the part of the program in which the binding is visible. For bindings that you define outside of a function or code block, the scope is the whole program—you can refer to such bindings wherever you want, even deep inside code blocks. These are called **global**.

Local Scope

Bindings created for function parameters or declared inside a function can be referenced only in that function. These are called local bindings. Every time the function is called, new instances of these bindings are created.

```javascript
let x = 5;
if (true) {
  let y = 10;
  var z = 20;
  console.log(x + y + z);
  //35
}
```
Each scope can see out into the scope around it, so $x$ is visible inside the block in the example, but $y$ is not visible. Note that because we used `var` to declare $z$, it is still visible to the global scope. This is why cool kids don’t use `var`!

---

**Side effects**

So when and why should you call a function in your program? Functions can be roughly divided into those that are called for their side effects and those that are called for their return value.

We’ve written a lot of functions that are called for their side effects. All of those that return a `console.log()` function. They just print a line. The second version of a function one that returns some value. Functions that create values are easier to combine with other functions than functions that directly perform side effects like `console.log()` some text.

That said, side effects are often useful. `console.log()` is extremely helpful.

**Summary**

- *Functions* are written to perform a task.
- Don’t re-create the wheel,
• Functions take data, perform a set of tasks on the data, and then return the result.

• We can define parameters to be used when calling the function.

• When calling a function, we can pass in arguments, which will set the function's parameters.

• We can use return to return the result of a function which allows us to call functions anywhere, even inside other functions.

Further Resources

We have only just touched on functions. You'll be creating and using many of them throughout the course. Here are a few resources for you to dive deeper

• A fantastic primer on functions by the Mozilla Developers Network

• Arrow functions and why people love them (site has annoying popups, I recommend a popup blocker like uBlock Origin).

• AirBnB's Javascript Style Guide contains some great pointers on proper syntax. That said, from what I can tell, JavaScript is built to be a very forgiving language so code how you want :)