Functions

The [previous chapter](/3-programming) was very conceptual. Loops are tricky to get a hang of, but once you visualize how looping can solve problems, you’re definitely on your way to thinking algorithimically, which is to say, programmatically. This chapter is a bit more focused, but it builds on the idea of a **block** of code, like the loop or the if statement. Blocks in JavaScript are always surrounded by braces ({}), and that is true for **functions**, as well.

## Function, function, what's your…

Just as looping is useful because it automates repetitive tasks, functions break up your code into smaller pieces. This means that it’s easier to reason about your work abstractly and find problems. Additionally, functions also automate repetitive tasks, and that’s the way I’ll introduce them to you.

Imagine if you want your computer to make burritos. Wouldn’t it be great to just tell it “make a burrito” every time you wanted one, instead of saying “get out the tortillas, take one out, place it on the griddle,” and so on? We make little tasks in our daily life abstract using the huge figurative power of language as well as our own memories. Computers can’t think as abstractly, however. And though they do have memories, we still need to spell things out in detail. Yet once we do it, we’re set. We can ask the computer to make us a burrito from then on.

Of course, I don’t think computers can make good burritos, but stick with me.

All those burrito-making instructions can be collapsed into a function. That might look something like:

// This code is for illustration only. It will crash your console.  
//  
let makeABurrito;  
makeABurrito = function(){  
 prepareTortilla();  
 addBeans();  
 addOnionsAndCilantro();  
 // etc.  
 rollUpTortilla();  
}

As you can see, there are syntactic similarities between functions and loops. Both use blocks with braces, and both have parentheses. In fact, this snippet of code is perfectly fine JavaScript. That’s because the makeABurrito() function does nothing but **call** other functions. We can imagine that a function like prepareTortilla() has even more specific steps inside it. But with the function in place, you just have to execute makeABurrito() once and be done with it. The internals of the function take care of everything else.

Quickly before moving on, that line // etc. is a **comment**. Comments are very useful in programming because they can serve as little messages to yourself (or to other programmers) about what is going on in your program. In JavaScript, everything after two slashes (//) to the end of the line is **commented out**. If you want to comment out a whole section of multiple lines, begin it with /\* and close it with \*/. Or you can put a // in front of every line. I’ll start commenting the code I provide, where necessary.

## Parameters

Back to the function. Did you notice the parentheses that follow the function name? Where have you seen this kind of of syntax before? We’ve already gone over prompt(), for example, in the [previous chapter](/3-programming/), and that is, of course, a function.[[1]](#footnote-25) But recall how we typed it:

let userInput;  
userInput = prompt("What do you want to have for dinner?", "Type your answer here.");

The parentheses aren’t empty. Instead, they have two strings in them. From experience, we know that that the first string is what appears at the top of the prompt box, and the second string is what appears where we type our answer. We can abstract out the function, then, as prompt(promptText, defaultText), where promptText and defaultText are two variables. And, in fact, if we were to rewrite the above as:

let promptText, defaultText, userInput;  
promptText = "What do you want to have for dinner?";  
defaultText = "Type your answer here.";  
userInput = prompt(promptText, defaultText);

It would work in exactly the same way. These two variables, promptText and defaultText are **parameters** that we **send** to the function.[[2]](#footnote-27) Giving functions parameters lets us change the internals of the function to let it react to specific instances. Now sometimes I want black beans in my burrito, and sometimes I want pinto beans. Let’s add a parameter to makeABurrito() to let us specify which beans to use on the fly:

// This code is for illustration only. It will crash your console.  
//  
let makeABurrito;  
makeABurrito = function(beansVariable){  
 let beansResponse;  
 prepareTortilla();  
 addBeans();  
 addOnionsAndCilantro();  
 beansResponse = "You ordered " + beansVariable + " beans. Good choice!";  
 $("#response").html(beansResponse);  
 // etc.  
 rollUpTortilla();  
}

If we were to execute:

// This code is for illustration only. It will crash your console.  
//  
let blackBeans;  
blackBeans = "black";  
makeABurrito(blackBeans);

We would see that the webpage would now read “You ordered black beans. Good choice!” Don’t actually try this, yet, though. Can you see why that is the case? We define a variable, blackBeans and assign it to the string "black". Next, we send (or **pass**) that variable as a parameter to makeABurrito(). Now, inside the function, we see that it makes reference to a beansVariable, that has the value “black,” which it then prints in #response, like we did last chapter.

But where did beansVariable come from? And how did it get set to “black”? There’s no beansVariable = "black", after all. The answer is that the variable is defined at the same time as the function is. makeABurrito = function(beansVariable){} defines both the function, makeABurrito(), and the parameter, beansVariable, which can be used as a variable inside the function.

The number of parameters you can define is up to you. Imagine you had different kinds of tortillas, like whole wheat and regular wheat. You can redefine the function as:

// This code is for illustration only. It will crash your console.  
//  
let makeABurrito;  
makeABurrito = function(beansVariable, tortillaVariable){  
 prepareTortilla(tortillaVariable);  
 addBeans(beansVariable);  
 addOnionsAndCilantro();  
 // etc.  
 rollUpTortilla();  
}

What would happen if you executed makeABurrito("black", "whole wheat")? Notice how this looks rather similar to prompt(promptText, defaultText);?

## Back to numbers

makeABurrito() is a great function, and it’s making me hungry, so let’s abandon it for a bit and go back to using numbers. Back in [Chapter 2](/2-calculator), we made a tipping calculator. We can build on that example with real, usable code.

First, what information do you need in order to know how much to tip? You need to know the total and the tipRate, which is a percentage, like 15 or 20%. The barebones function looks like this, then:

let tipCalculator;  
tipCalculator = function(total, tipRate){  
 // 1. Calculate the percentage of the total   
 // as a variable “tipAmount”  
 //  
 // 2. Change #response to tell us the tip  
 // amount.  
}

In fact, type this into your scripts.js file. It can go after the earlier code you’ve written, or it can replace it. Note what’s in the comments; it’s a sketch of what the function will do. The first step is that it’ll do some math—an easy calculation. In the second step, it will tell us what the result of the calculation is.

let tipCalculator;  
tipCalculator = function(total, tipRate){  
 // step 1:  
 let tipAmount;  
 tipAmount = tipRate \* total;  
 // and step 2:  
 $("#response").html("Your tip is $" + tipAmount);  
};   
  
// Now call (or “execute”) the function, passing a   
// total of $50.00 and a tipRate of 20%:  
  
tipCalculator(50.00, 0.2);

Save and reload, and the webpage should now inform you that you owe $10. If it does, commit. In the exercises, we’ll expand on this function.

## Scope

Alongside the idea of a function block, that is, the set of curly braces, we also have the idea of **scope**. Beginner programmers often get tripped up by scope, but that’s ok, so do veteran programmers. Note that though conditionals and loops also use blocks, they don’t affect scope in the same way.

Just like a “scope” is used to see things that are far away (tele*scope*) or are really tiny (micro*scope*), in JavaScript, scope also has to do with visbility. Variables defined with the let keyword have block-level scope.[[3]](#footnote-31) That means:

> let global;  
> global = "🌏";  
> if (true) { console.log(global); };  
//--> 🌏  
> let globalFunction;  
> globalFunction = function(){ console.log(global); };  
> globalFunction();  
//--> 🌏

When we define global, it’s *visible* to us inside subsequent if statements and functions. But notice this:

> let global;  
> global = "🌏";  
> if (true) {  
 let blocky;  
 blocky = "📓";  
 console.log("global is " + global);  
 console.log("blocky is " + blocky);  
 }  
//--> global is 🌏  
//--> blocky is 📓  
> console.log("Wait, the value of blocky is really " + blocky + "?");

This last line will crash with a ReferenceError. blocky, it turns out, is only visible *within* the if statement block. Once the code leaves the {}, blocky is no longer available. The same works with functions:

> let global;  
> global = "🌏";  
> let showMeABurrito = function(){  
 let burrito;  
 burrito = "🌯";   
 global = "I’m global!";  
 console.log("global is " + global);  
 console.log("burrito is " + burrito);  
 }  
> showMeABurrito();  
//--> global is I’m global!  
//--> burrito is 🌯  
> console.log(global);  
//--> I’m global!  
> console.log("Wait, the value of burrito is really " + burrito + "?");

Crash. Again. Sadly, burrito is not defined. But notice that global was changed inside the function, and that change persisted outside the function. global is visible—and, as a result, changeable (or **mutable**)—everywhere. But burrito is not.

As Molly Bloom asks, “who’s he when he’s at home?” and we may, also, ask, “where are we when we’re in the console?” That is, if burrito is defined in the function block, where on earth is global defined? Even though we talk about typing “in” the console, we’re actually always within a special object called window. It is the [window of the browser](https://www.w3schools.com/jsref/obj_window.asp), and when we open up the console, we’re getting closer access to that window. In fact, prompt() and alert(), two functions you’ve already seen, are actually **methods** that belong to window; window.alert() and alert() will do the same thing. But more on methods [next chapter](/5-collections). The window is typically ignored, as it’s the frame that is unavoidable. Its ubiquity gives it the privilege of being silent.

Back to the purpose of this section. Scope helps you keep your code tidy, because there is less risk of variables’ being accessed where they shouldn’t be. Just remember, whenever you type let to define a variable inside a function, that variable is only available inside that function.

In closing, functions are powerful things, as we can see. And though, ultimately, the goals of this course are not to write code that is as modular as the use of functions would make possible, you will still be typing the word function a lot.

## Exercises

1. Add functionality to the tip calculator so that you can enter “20” *or* “.2” for 20%, and the calculator understands the difference.

## Footnotes

1. console.log() also looks like a function, but as I mentioned back in [chapter 2](2-calculator/#fn:consolelog), .log() is a method belonging to the console object. [↑](#footnote-ref-25)
2. Parameters are also often called **arguments**, but to my ears, that term is more opaque. [↑](#footnote-ref-27)
3. This is one difference between var and let, but I’m not teaching var except for those of you with old browsers. Other than the slightly contrived examples in this section, my examples won’t be making a lot of use of block-scoping. [↑](#footnote-ref-31)