Introduction to Functions

Now that we have the basics of JavaScript down, data types, conditional statements, and variables, we need functions to compute them, change them, and do something with them. We can think of functions as small computer programs. Functions are required to get our code to 'run' in JavaScript. Although you may not realize it, you have been using functions this entire time. Functions allow us to write code that will be used over and over again, keeping our code **DRY**. As such, functions will often times take input and give output, although that is not always required. Functions also have their own 'scope', in which we can assign a variable within a function that is not available anywhere else.

So, JavaScript functions perform a task every time you call or invoke them. For example, that task could be to print 'Hello world' to the screen or to calculate something. Functions make it easy for you to execute complex mathematical formulas without manually typing them every single time you need them.

How to Write a Function

There are three ways we can build a function. For this article, we will be focusing on just the first one. For our purposes right now, all three ways will act the same.

a). **Function myfunction**(){}

b). **const myfunction**2 = **functions**(){};

c). **const myfunction**3 = () => {} ;

Anatomy of a Function

a). **Function myfunction**(){}

A function will start with the function keyword. This tells whatever is running your program that what follows is a function and to treat it as such. After that comes the name of the function. We like to give functions names that describe what they do. Then comes an open and a close parenthesis. And finally, open and close brackets. In between these brackets is where all of our function code will go.

```
function add (){
  let x = 20;
  let y = 30;
  console.log(x + y);
}
/* invoke the function */
add()
```

In this example, we declare a function add and set it up to add two variables(x and y). We can then see that in order to run this function, we need to write or invoke its name with the parentheses after it. This is the syntax to run a function. A function always needs parentheses to run.

Parameters and arguments.

Parameters and arguments allow us to pass data into functions and use this data within the function. In this objective, we will learn the difference between the two and how to use them.

One of the main reason we use functions is code re–use. We are always looking for ways to keep our code DRY (Don't Repeat Yourself). Because of this, functions allow us to pass data into them for use inside the function.

We can declare little function variables called parameters and pass that data in when we call the function (the data are then called arguments).

Parameters.

A function parameter will represent the data we pass into a function, for use in the function. Essentially when we write a function we assign the data variable names, even without knowing what the data will be. We set these variables inside of the parentheses when we write the function. There is no limit to the amount of parameters we can include in a function, but each variable name must be separated by a comma. We can then use these variables within our function just as we would any other variable.

```
function name (paremeter1, parameter2, ...){
//function logics
}
```

In our add example we could have it as follows:

```
function add (x,y){
//Addition logics here
}
```

Arguments

Once we have our parameters set up in our function, we can now pass data into the function. In order to do this, we will use the parentheses we write when we call the function. We call these pieces of data arguments. Arguments can be ANY data type (string, number, Boolean, object, array, even other functions!).

Unlike other languages, JavaScript does not require us to set the data type when we write the function, although you should make an effort to understand what type of data will be coming into the function (and if you are using pre-built functions, you should know what data type that function expects).

In the function below name is parameter while Liam is an arguments:

```
function user(name){
console.log(name)
}
user('Liam')
```

Scope

Scope is defined as what variables we currently have access to and where. So far in this course, we have been working in Global scope, in that we can access any variable we have created, anywhere in our code. There are a couple different levels of scope: you may have heard of block level scope (used in if statements and for loops) in which a variable using either letor const is only available within the statement or loop.

Function Level Scope

Functions have something similar, known as function scope. Function scope allows us to create variables inside of functions, that are essentially private to that function. We can not reach into a function from the outside and get access to these variables. But we are free to use these variables anywhere within our function. Conversely, we DO have access to variables outside of the function. It is a one-way street. Functions can reach out and grab variables outside of their scope, but we can not reach into a function to get a variable.

```
const var1 = 'Hello';

function myFunc() {
  const var2 = 'World!';
  console.log(var1);  Hello
  console.log(var2);  World!
}

myFunc();
console.log(var1);  Hello
console.log(var2);  var2 is not defined
```

The return Statement.

We will not console.log everything that comes out of a function. Most likely we will want to return something. There is one way we can access data from within a function. In fact, it is the only way for us to get ANY data from a function, and that is to use the keyword return. Think of the return statement as the only way for data to escape a function. Nothing other than what is returned can be accessed outside of the function. Also note that when a function hits a return statement, the function immediately stops what it is doing and returns.

We can also assign the value of a return statement to another variable, and we will now have access to the data returned from the function.

```
function addTwoNumbers(a, b) {
  const sum = a + b;
  return sum;
  console.log('This will never be reached');
}

const newSum = addTwoNumbers(1, 2);
console.log(newSum);  3
console.log(sum);  sum is not defined
```

We will never be able to have access to the actual variable created in the function. We will only have access to the data that variable was assigned to.

– **Make sure you understand Function Expression, parameter, argument and scope.**

An anonymous function is a function without a name. An anonymous function is often not accessible after its initial creation.

The following shows an anonymous function that displays a message:

```
let show = function () {
 console.log('Anonymous function');
};
show();
```

Note that the anonymous function above has no name between the function keyword and parentheses ()
We often use anonymous functions as arguments of other functions. For example:

```
setTimeout(function () {
   console.log('Execute later after 1 second')
  },
1000);
```

n this example, we pass an anonymous function into the setTimeout() function. The setTimeout() function executes this anonymous function one second later.

Arrow functions

ES6 introduced Arrow functions expression that provides a shorthand for declaring anonymous functions. For example, this function:

```
let show = function () {
    console.log('Anonymous function');
}
```

It can be shortened using the following arrow function:

```
let show = () => console.log('Anonymous function');
```

Question: What is the arrow function version of the following function?

```
let add = function (a, b) {
 return a + b;
};
```

Send your answer to luxtinc@gmail.com

Subject should be your full name as it appears in our records.

Functions are the first-class citizens in JavaScript, so you can pass a function to another as an argument.

Best wishes.

Lux Tech Academy