CONTENTS

RELATED

CodeIgniter: Getting Started With a Simple Example

View

How To Install Express, a Node.js Framework, and Set Up Socket.io on a VPS

View

Tutorial Series: How To Code in JavaScript

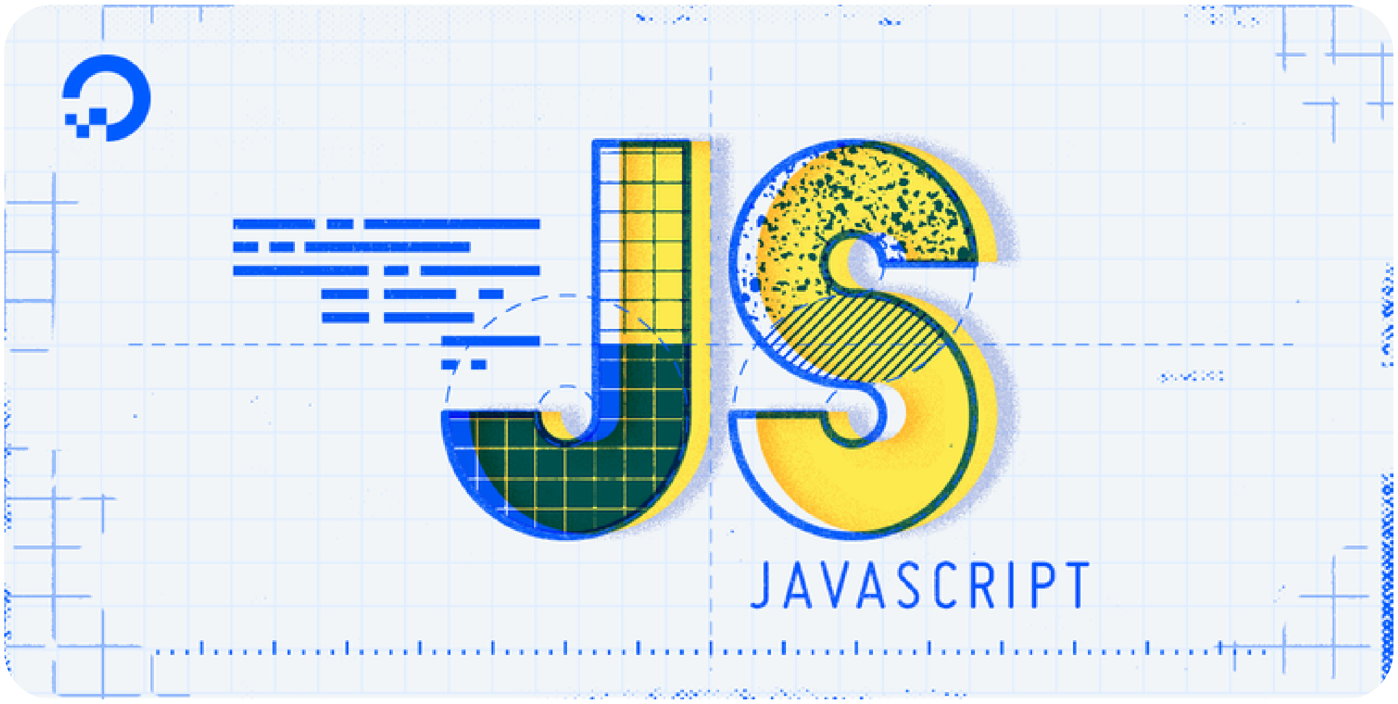
// Tutorial //

How To Define Functions in JavaScript

Published on October 9, 2017 · Updated on August 26, 2021

JavaScript Development

By Tania Rascia



# Introduction

A **function** is a block of code that performs an action or returns a value. Functions are custom code defined by programmers that are reusable, and can therefore make your programs more modular and efficient.

In this tutorial, we will learn several ways to define a function, call a function, and use function parameters in JavaScript.

Functions are defined, or declared, with the `function` keyword. Below is the syntax for a function in JavaScript.

```
function nameOfFunction() {
        // Code to be executed
}
```

Copy

The declaration begins with the `function` keyword, followed by the name of the function. Function names follow the same rules as variables — they can contain letters, numbers, underscores and dollar signs, and are frequently written in camel case. The name is followed by a set of parentheses, which can be used for optional parameters. The code of the function is contained in curly brackets, just like a for statement or an if statement.

In our first example, we'll make a **function declaration** to print a greeting statement to the console.

```
// Initialize greeting function
function greet() {
        console.log("Hello, World!");
}
```

Copy

Here we have the code to print `Hello, World!` to the console contained inside the `greet()` function. However, nothing will happen and no code will execute until we **invoke**, or call the function. You can invoke a function by writing the name of the function followed by the parentheses.

```
// Invoke the function
greet();
```

Copy

Now we will put those together, defining our function and invoking it.

greet.js

```
// Initialize greeting function
function greet() {
        console.log("Hello, World!");
}
```

Copy

With the call for `greet();`, the function will run and we will receive the `Hello, World!` as the program's output.

```
Output
Hello, World!
```

Now we have our `greet()` code contained in a function, and can reuse it as many times as we want.

Using parameters, we can make the code more dynamic.

# Function Parameters

In our `greet.js` file, we created a basic function that prints `Hello, World` to the console. Using parameters, we can add additional functionality that will make the code more flexible. **Parameters** are input that get passed into functions as names and behave as local variables.

When a user logs in to an application, we may want the program to greet them by name, instead of just saying, "Hello, World!".

We'll add a parameter into our function, called `name`, to represent the name of the person being greeted.

```
// Initialize custom greeting function
function greet(name) {
        console.log(`Hello, ${name}!`);
}
```

Copy

The name of the function is `greet`, and now we have a single parameter inside the parentheses. The name of the parameter follows the same rules as naming a variable. Inside of the function, instead of a static string consisting of `Hello, World`, we have a template literal string containing our parameter, which is now behaving as a local variable.

You'll notice we haven't defined our `name` parameter anywhere. We assign it a value when we invoke our function. Assuming our user is named Sammy, we'll call the function and

The value of `"Sammy"` is being passed into the function through the `name` parameter. Now every time `name` is used throughout the function, it will represent the `"Sammy"` value. Here is the whole code.

greetSammy.js

Copy

```
// Initialize custom greeting function
function greet(name) {
	console.log(`Hello, ${name}!`);
}

// Invoke greet function with "Sammy" as the argument
greet("Sammy");
```

When we run the program above, we'll receive the following output.

```
Output
Hello, Sammy!
```

Now we have an example of how a function can be reused. In a real world example, the function would pull the username from a database instead of directly supplying the name as an argument value.

In addition to parameters, variables can be declared inside of functions. These variables are known as **local variables**, and will only exist inside the *scope* of their own function block. Variable scope determines variables' accessibility; variables that are defined inside of a function are not accessible from outside of the function, but they can be used as many times as their function is used throughout a program.

# Returning Values

More than one parameter can be used in a function. We can pass multiple values into a function and return a value. We will create a function to find the sum of two values, represented by `x` and `y`.

sum.js

```
// Invoke function to find the sum
add(9, 7);
```

In the program above, we defined a function with the parameters `x` and `y`, and then passed the values of `9` and `7` to the function. When we run the program, we'll receive the sum of those numbers as the output.

```
Output
16
```

In this case, with `9` and `7` passed to the `sum()` function, the program returned `16`.

When the `return` keyword is used, the function ceases to execute and the value of the expression is returned. Although in this case the browser will display the value in the console, it is not the same as using `console.log()` to print to the console. Invoking the function will output the value exactly where the function was invoked. This value can be used immediately or placed into a variable.

# Function Expressions

In the last section, we used a function declaration to get the sum of two numbers and return that value. We can also create a **function expression** by assigning a function to a variable.

Using our same `add` function example, we can directly apply the returned value to a variable, in this case `sum`.

functionExpression.js

Copy

```
// Assign add function to sum constant
const sum = function add(x, y) {
        return x + y;
}

// Invoke function to find the sum
sum(20, 5);
```

Now the `sum` constant is a function. We can make this expression more concise by turning it into an **anonymous function**, which is an unnamed function. Currently, our function has the name `add`, but with function expressions it is not necessary to name the function and the name is usually omitted.

anonymousExpression.js

```
// Assign function to sum constant
const sum = function(x, y) {
        return x + y;
}

// Invoke function to find the sum
sum(100, 3);
```

Copy

```
Output
103
```

In this example, we've removed the name of the function, which was `add`, and turned it into an anonymous function. A named function expression could be used to aid in debugging, but it is usually omitted.

# Arrow Functions

So far, we have gone through how to define functions using the `function` keyword. However, there is a newer, more concise method of defining a function known as **arrow function expressions** as of ECMAScript 6. Arrow functions, as they are commonly known, are represented by an equals sign followed by a greater than sign: `=>`.

Arrow functions are always anonymous functions and a type of function expression. We can create a basic example to find the product of two numbers.

arrowFunction.js

```
// Define multiply function
const multiply = (x, y) => {
        return x * y;
```

Copy

```
Output
120
```

Instead of writing out the keyword `function`, we use the `=>` arrow to indicate a function. Otherwise, it works similarly to a regular function expression, with some advanced differences which you can read about under Arrow Functions on the Mozilla Developer Network.

In the case of only one parameter, the parentheses can be excluded. In this example, we're squaring `x`, which only requires one number to be passed as an argument. The parentheses have been omitted.

```
// Define square function
const square = x => {
        return x * x;
}

// Invoke function to find product
square(8);
```

Copy

```
Output
64
```

Note: In the case of no parameters, an empty set of parentheses `()` is required in the arrow functions.

With these particular examples that only consist of a `return` statement, arrow functions allow the syntax to be reduced even further. If the function is only a single line `return`, both the curly brackets and the `return` statement can be omitted, as seen in the example below.

```
// Define square function
const square = x => x * x;

// Invoke function to find product
```

Copy

All three of these types of syntax result in the same output. It is generally a matter of preference or company coding standards to decide how you will structure your own functions.

# Conclusion

In this tutorial, we covered function declarations and function expressions, returning values from functions, assigning function values to variables, and ES6 arrow functions.

Functions are blocks of code that return a value or perform an action, making programs scalable and modular.

Next in series: Understanding Prototypes and Inheritance in JavaScript ->

JavaScript is a high-level, object-based, dynamic scripting language popular as a tool for making webpages interactive.

Subscribe

JavaScript Development

Browse Series: 37 articles

Expand to view all

About the authors

Tania Rascia Author

Lisa Tagliaferri Editor

Was this helpful? Yes No

Comments

5 Comments

Leave a comment...

This textbox defaults to using `Markdown` to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

engdanishjan619 • January 24, 2022

it is very nice to understand and easy to use the given and work with it .

Reply

jainayushjain95 • March 8, 2019

That's a great article. Inspired me to write a complete series over JS:

4)[Arrays In JavsScript: Part 3](https://fullstackgeek.blogspot.com/2019/01/filtering-sorting-arrays-javascript_31.html)

Reply

Cassyemmanuels • November 27, 2018

Hello everyone i need an assist on something in javascript please, the issue at hand is take for instance a php website with multiple pages, were the header and footer is in different files using php include statement to include the header and footer i. e <?php include "header" ; ?> And an external javascript file linked in the header or footer. I. e for instance <javascript src="mm.Js"> And a button located in the header page, since the header is included in all pages, that makes the one button appear in all pages, the button is to trigger a javascript function inside the linked javascript file for instance a function to alert "hello world ", now when the button is clicked on the index page it triggers the function but when the same button is clicked in the other pages the function Will not be triggered…

Reply

Leonardo Martinez • November 7, 2017

Excellent, it is clear that anyone can understand it, I appreciate very much your effort, thank you.

Reply

ravikiran yadav • October 31, 2017

Reply

Try DigitalOcean for free

Click below to sign up and get **$200 of credit** to try our products over 60 days!

Sign up →

Popular Topics

Ubuntu

Linux Basics

JavaScript

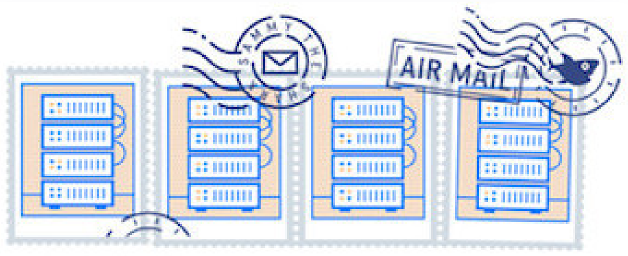
Python

MySQL

Docker

Kubernetes

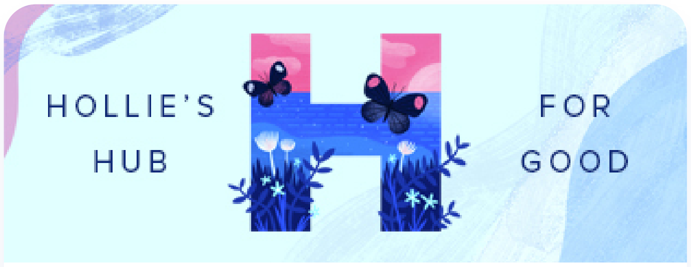
All tutorials →



Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

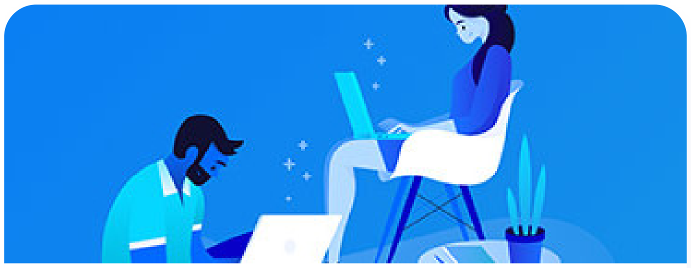
Sign up →



Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →



Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

Kubernetes Course Learn Python 3 Machine Learning in Python
Getting started with Go Intro to Kubernetes

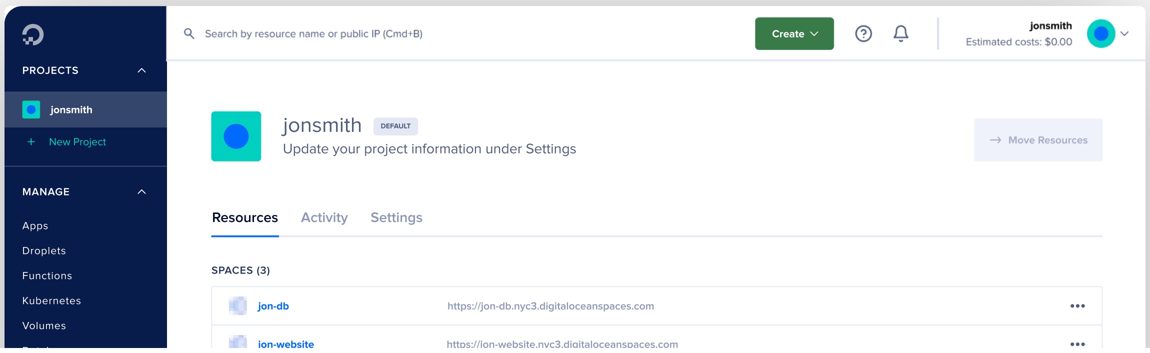
DigitalOcean Products

Cloudways Virtual Machines Managed Databases Managed Kubernetes
Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn more →



Company	Products	Community	Solutions	Contact
About	Products	Tutorials	Website Hosting	Support

Partners

Channel Partners

Referral Program

Affiliate Program

Press

Legal

Security

Investor Relations

DO Impact

Functions

Cloudways

Managed Databases

Spaces

Marketplace

Load Balancers

Block Storage

Tools & Integrations

API

Pricing

Documentation

Release Notes

Uptime

Currents Research

Hatch Startup Program

deploy by DigitalOcean

Shop Swag

Research Program

Open Source

Code of Conduct

Newsletter Signup

Meetups

Streaming

VPN

SaaS Platforms

Cloud Hosting for Blockchain

Startup Resources