**Function:-**

**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.

**Defining a Function**

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

function nameOfFunction() {

// Code to be executed

}

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](https://www.digitalocean.com/community/tutorials/understanding-syntax-and-code-structure-in-javascript#identifiers). 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](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-javascript) or an [if statement](https://www.digitalocean.com/community/tutorials/how-to-write-conditional-statements-in-javascript).

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!

// Invoke the function

greet();

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

greet.js

// Initialize greeting function

function greet() {

console.log("Hello, World!");

}

// Invoke the function

greet();

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

Output

Hello, World!

**Function Parameters**

**Parameters** are input that get passed into functions as names and behave as local variables.

greetSammy.js

// 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!

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

// Initialize add function

function add(x, y) {

return x + y;

}

// Invoke function to find the sum

add(9, 7);

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**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) 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

// Assign add function to sum constant

const sum = function add(x, y) {

return x + y;

}

// Invoke function to find the sum

sum(20, 5);

Copy

Output

25

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](https://www.ecma-international.org/ecma-262/6.0/). 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;

}

// Invoke function to find product

multiply(30, 4);

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Arrow_functions).

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

square(10);

Copy

Output

100

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.