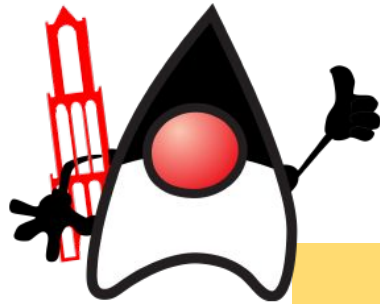




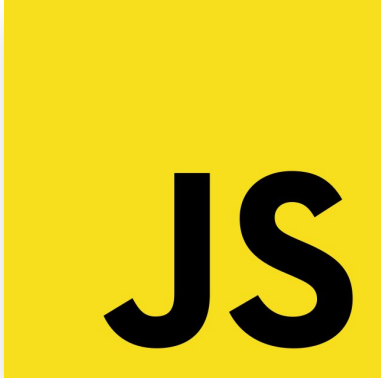
Curso FullStack Python

Codo a Codo 4.0



Javascript

Parte 3

A yellow square with a black shadow, containing the letters 'JS' in a bold, black, sans-serif font.

JS

Funciones

Las **funciones** nos permiten agrupar líneas de código en tareas con un nombre (subprograma), para que, posteriormente, podamos hacer referencia a ese nombre para realizar todo lo que se agrupe en dicha tarea. Para usar funciones hay que hacer 2 cosas:

- **Declarar la función:** crear la función ***es darle un nombre***, definir los datos de entrada (opcional) y decirle las tareas (instrucciones) que realizará y que valor retornará (opcional).
- **Ejecutar la función:** «Llamar» (Invocar) a la función para que realice las tareas de su contenido. Se puede invocar la función la cantidad de veces necesaria en el programa principal.

En el siguiente ejemplo vemos la declaración y la ejecución:

```
// Declaración de la función "saludar"
function saludar() {
  console.log("Hola, soy una función"); // Contenido de la función
}
```

JS

Primer paso:
Declarar la función

```
// Ejecución de la función
saludar();
```

JS

Segundo paso:
Ejecutarla

Funciones

Razones para declarar funciones:

- Cuando un conjunto de instrucciones se va a usar **muchas veces**, se declara una sola vez la función con esas instrucciones y se llama muchas veces. por ejemplo **parseFloat()** y **parseInt()**, **alert()** y **prompt(...)** (son funciones de JS no escritas por el programador). Una función me permite **modularizar**, es decir, armar módulos. Nosotros hasta ahora programamos todo en el cuerpo principal del programa de JavaScript (el código todo junto), pero ahora podremos separar el código en partes, reduciendo un programa complejo en unidades más simples.
- En el momento de **trabajar en equipo** dividimos el trabajo en partes, cada uno realizará una función y alguien o varias personas cumplirán la función de integradores de esas funciones y armarán un programa principal más grande, que haga uso de esas funciones.
- **Por claridad.** Para que programa no tenga muchas líneas y sea difícil seguirlo, lo que se hace es separar fracciones de código en una función con un nombre que las identifique y luego llamarlas, entonces el programa queda mucho más claro (aunque esa función solo se llame una vez), es reutilizable porque solamente tendré que cambiar los valores de entrada.

Funciones

El **nombre** de la función tiene que ser significativo de lo que va a hacer la función También definiremos los datos de entrada (si es necesario) y decirle qué instrucciones va a tener dentro, qué es lo que va a hacer. También se puede definir de manera opcional qué valor retornará, que es lo más frecuente.

Recomendaciones de nombre para la función:

- Nombres simples, claros.
- Verbos en infinitivo (-ar, -er, -ir).
- Si es más de una palabra con nomenclatura camelCase.

Las podemos clasificar como:

- **Funciones sin parámetros:**
 - Que no devuelven valores
 - Que devuelven valores
- **Funciones con parámetros:**
 - Que no devuelven valores
 - Que devuelven valores

Funciones | Ejemplo

Desarrollar la tabla de multiplicar del 1

```
for (i = 1; i <= 10; i++) {  
  console.log("1 x", i, "=", 1 * i);  
}
```

JS

Objetivo: mostrar esta tabla de multiplicar tres veces

```
// Primera vez  
for (i = 1; i <= 10; i++) {console.log("1 x", i, "=", 1 * i);}  
// Segunda vez  
for (i = 1; i <= 10; i++) {console.log("1 x", i, "=", 1 * i);}  
// Tercera vez  
for (i = 1; i <= 10; i++) {console.log("1 x", i, "=", 1 * i);}
```

JS

Esta es una primer aproximación, pero repetimos mucho código

Lo resolveremos usando bucles y funciones:

```
//Declaración de la función tablaDelUno()  
function tablaDelUno(){  
  for (i = 1; i <= 10; i++) {console.log("1 x", i, "=", 1 * i);}  
}  
//Bucle que ejecuta 3 veces la función tablaDelUno()  
for (let i = 1; i <= 3; i++) {tablaDelUno();}
```

JS

La función tablaDelUno() llama a un For de 10 iteraciones. El siguiente For ejecuta la función 3 veces.

Ver ejemplo funciones_1 (.html y .js)

Funciones | Parámetros

Las funciones se convierten en mucho más flexibles al pasarles parámetros, que no son más que variables que existirán sólo dentro de dicha función, con el valor pasado desde la ejecución.

```
// Declaración
function tablaMultiplicar(hasta) {
  for (var i = 1; i <= hasta; i++)
    console.log("1 x", i, "=", 1 * i);
}
```

JS

Esta función tiene un solo parámetro que es el que indica hasta qué valor calculará.

```
//Ejecución
tablaMultiplicar(4);
```

JS

```
// Declaración
function saludarDos(miNombre){
  console.log("Hola " + miNombre);
}
```

JS

En este ejemplo la función muestra un texto concatenado a un valor pasado por parámetro.

```
//Ejecución
saludarDos("Juan Pablo"); //Parámetro fijo
var nombre= prompt("Ingrese su nombre"); //Pedimos valores
saludarDos(nombre); //Parámetro variable
```

JS

*Ese valor podrá ser asociado a una variable o ingresado por el usuario. No necesariamente tiene que tener el mismo nombre la variable con la que creé la función (**miNombre**) que la variable que le paso como parámetro (**nombre**).*

Ver ejemplo funciones_2 (.html y .js)

Funciones | Parámetros múltiples

Las funciones también pueden recibir más de un parámetro. En este caso debemos tener en cuenta que hay que respetar el orden en que pasamos los valores y en el que los usamos al llamarla.

```
// Declaración
function tablaMultiplicar(tabla, hasta) {
  for (var i = 1; i <= hasta; i++)
    console.log(tabla + " x " + i + " = ", tabla * i);
}
```

JS

Esta función tiene dos parámetros:

- *La tabla en sí.*
- *Hasta qué valor calculará.*

```
// Ejecución
tablaMultiplicar(1, 10); // Tabla del 1, calcula desde el 1 hasta el 10
tablaMultiplicar(5, 10); // Tabla del 5, calcula desde el 1 hasta el 10
```

JS

Lo que le paso a la función se llama **argumento** y lo que recibe la función se llama **parámetro**.

Funciones | Parámetros múltiples

Este es un ejemplo con 3 parámetros donde se evalúa la mayoría de edad de una persona:

```
// Declaración
function mayoriaEdad(miApellido, miNombre, miEdad){
    console.log("Apellido y nombre: " + miApellido + ", " + miNombre);
    if (miEdad >= 18) {
        console.log("Es mayor de edad " + "(" + miEdad + ")");
    } else{
        console.log("No es mayor de edad " + "(" + miEdad + ")");
    }
}
```

JS

```
//Ejecución
var ape= prompt("Ingrese su apellido");
var nom= prompt("Ingrese su nombre");
var edad= prompt("Ingrese su edad");
mayoriaEdad(ape, nom, edad);
```

JS

*Esta función recibe tres parámetros y en función del valor de uno de ellos (**miEdad**) determina si la persona es mayor de edad (>=18)*

[Ver ejemplo funciones_2 \(.html y .js\)](#)

Funciones | Devolución de valores

Hasta ahora hemos utilizado funciones simples que realizan acciones o tareas (en nuestro caso, mostrar por consola), pero habitualmente, lo que buscamos es que esa función realice una tarea y nos devuelva la información al **exterior de la función**, para así utilizarla o guardarla en una variable, que utilizaremos posteriormente para nuestros objetivos.

Para ello, se utiliza la palabra clave **return**, que suele colocarse al final de la función, ya que con dicha devolución terminamos la ejecución de la función (si existe código después, nunca será ejecutado).

```
// Declaración
function sumar(a, b) {
  return a + b; // Devolvemos la suma de a y b al exterior de la función
}
```

JS

```
// Ejecución
var a = 5, b = 5;
var resultado = sumar(a, b); // Se guarda 10 en la variable resultado
console.log("La suma entre "+ a +" y "+ b +" es: "+ resultado);
```

JS

Funciones | Devolución de valores

Sabemos entonces que una función devolverá un valor cuando utilicemos la palabra clave **return**. Veamos dos funciones que realizan lo mismo, pero una retorna valores y otra no:

```
function sumar(num1, num2){  
    var suma = num1 + num2;  
    console.log("La suma es " + suma);  
}  
sumar(2,5);
```

JS

Esta función devuelve "La suma es ...", pero no está retornando valores, no devuelve un valor, hace lo mismo que una función que imprime por consola.

```
function sumarDos(num1, num2){  
    var suma = num1 + num2;  
    return suma;  
}  
n1 = 2;  
n2 = 3;  
var resultado = sumarDos(n1, n2);  
console.log("El resultado es: " + resultado);
```

JS

*En este caso sí va a devolver un valor, ese valor se almacena en una variable llamada **resultado** que contiene la suma de dos valores realizada por la función sumarDos que ha retornado un valor.*

El nombre de los parámetros de la función no tiene que coincidir con el nombre de variable que le paso como argumento a la función.

A esto se le denomina "pasaje por valor", ya que estoy pasando una copia de la variable.

Funciones | Devolución de valores

Otra alternativa es hacer que la función guarde directamente el resultado que devuelve en una variable:

```
var suma = function sumarTres(numero1, numero2) {  
    return numero1 + numero2;  
}  
console.log(suma(40, 15));
```

JS

*Al retornar un valor éste se guarda en la variable **suma**.*

En general una función se utiliza para hacer una acción o tarea específica. Si quisiéramos hacer la resta de estos dos números nos conviene crear otra función.

```
var numeroMaximo = function (valor1, valor2) {  
    if (valor1 > valor2) {  
        return valor1;  
    }  
    return valor2;  
}  
var v1 = parseInt(prompt("Ingrese un numero entero"));  
var v2 = parseInt(prompt("Ingrese otro numero entero"));  
console.log("El numero maximo es:", numeroMaximo(v1,v2));
```

JS

En este caso se piden dos valores y si la condición no se cumple se asume que el valor2 es el máximo (no es necesario un else)

Ver ejemplo funciones_2 (.html y .js)

Funciones | Función flecha (arrow Function)

En JavaScript, al igual que en Python, existe la forma resumida de escribir las funciones. Se llaman **funciones de tipo flechas**, en Python se llaman funciones Lambda.

Flecha hace alusión a \Rightarrow y hacen que sea un poco más fácil, más rápido y más breve la creación de una función. Es una alternativa más compacta para expresar una función. Es más limitada, como para funciones más chicas se pueden crear y que no ocupen tanto espacio en memoria.

Para crear estas funciones flecha partiremos del ejemplo:

```
// Función tradicional
function cuadrado(x){
    return x*x;
}
console.log(cuadrado(2));
```

JS

*Esta función calcula el cuadrado de un número recibido como parámetro. Retorna $x*x$. Luego imprimo por consola y devolvería 4 porque le pasé el valor 2.*

```
// Función Arrow
var aCuadrado = x => x*x;
```

JS

*Declaro con **var** el nombre de la función porque las funciones son variables, ocupan un espacio en memoria cuando las defino. En la función aCuadrado le colocamos la "a" de "arrow".*

*En esta función X es el parámetro que antes se colocaba entre () y a la derecha de la flecha pongo lo que debe hacer la función ($x*x$), esto que está a la derecha es lo que se va a retornar.*

En conclusión: lo que era **return $x*x$** lo pongo a la **derecha** de la flecha y lo que era **x** (lo que recibo como parámetro) lo pongo a la **izquierda** de la flecha.

Funciones | Función flecha (arrow Function)

Si tenemos más variables que pasarle, por ejemplo con la función sumar, tengo que usar los paréntesis:

```
function sumar (num1,num2) {  
    return num1+num2;  
}  
console.log(sumar(4,6));
```

JS

```
var aSumar = (num1,num2) => num1+num2;  
console.log(aSumar(5,7));
```

JS

En este caso mantenemos los parámetros entre paréntesis y colocamos a la derecha lo que devolverá la función.

Si tengo más de una línea lo puedo colocar de esta manera:

```
// Función tradicional  
function multiplicar (num1,num2) {  
    producto= num1*num2;  
    return producto;  
}  
console.log(multiplicar(2,3));
```

JS

Esta forma me permite crear una función de tipo flecha más elaborada.

```
// Función Arrow  
var aMultiplicar = (num1,num2) =>  
{  
    producto= num1*num2;  
    return producto;  
}  
console.log(aMultiplicar(6,7));
```

JS

[Ver ejemplo funciones_flecha \(.html y .js\)](#)

Funciones | Función flecha (arrow Function)

Una expresión de función flecha es una alternativa compacta a una expresión de función tradicional, pero es limitada y no se puede utilizar en todas las situaciones.

Nota: Cada paso a lo largo del camino es una "función flecha" válida.

```
// Función tradicional
```

```
function (a){  
  return a + 100;  
}
```

```
// Desglose de la función flecha
```

```
// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y las llaves de  
apertura.
```

```
(a) => { return a + 100; }
```

```
// 2. Quita los llaves{} del cuerpo y la palabra "return" – el return está implícito.
```

```
(a) => a + 100;
```

```
// 3. Suprime los paréntesis de los argumentos
```

```
a => a + 100;
```

JS

Funciones | Sintaxis función flecha

JS

```
//Sintaxis básica
//Un parámetro. Con una expresión simple no se necesita return:
param => expression

//Varios parámetros requieren paréntesis. Con una expresión simple no se necesita
return:
(param1, paramN) => expression

//Las declaraciones de varias líneas requieren llaves y return:
param => {
  let a = 1;
  return a + b;
}

//Varios parámetros requieren paréntesis. Las declaraciones de varias líneas requie
ren llaves y return:
(param1, paramN) => {
  let a = 1;
  return a + b;
}
```




Scope (alcance)

El **scope** (*alcance*) determina la accesibilidad (visibilidad) de las variables. Define ¿en qué contexto las variables son visibles y cuándo no lo son?

Una variable que no está “al alcance actual” no está disponible para su uso.

En JavaScript hay **dos tipos de alcance**:

- Alcance local
- Alcance global (entorno completo de JavaScript)

JavaScript tiene un alcance de función: cada función crea un nuevo alcance.

Las variables definidas dentro de una función **no son accesibles** (visibles) desde fuera de la función, o lo que es lo mismo, solamente se puede acceder a ellas dentro de la función. Esto sucede así porque la función “crea un ámbito cerrado” de forma tal que no se pueda acceder a una variable definida **exclusivamente dentro de la función** desde fuera de ella o dentro de otras funciones.

Scope (alcance)

En el siguiente ejemplo creamos una variable llamada carName a la cual le asignamos un valor:

```
// aca no puedo usar la variable carName
function myFunction() {
  var carName = "Volvo";
  // aca si puedo usar la variable carName
}
// aca no puedo usar la variable carName
```

JS

Podremos acceder al contenido de esa variable solamente dentro de la función

Ver primer ejemplo de scope (.html y .js)

Este tipo de variables son de **alcance local**, porque solamente valen en el ámbito de la función, y no en el ámbito a nivel de programa.

Argumentos de función

Los argumentos de la función (parámetros) funcionan como **variables locales** dentro de las funciones.

Scope (alcance)

Variables globales de JavaScript

Una variable declarada fuera de una función se convierte en **global**. Esto quiere decir que tiene alcance global: todos los scripts y funciones de una página web pueden acceder a ella.

```
var carName2 = "Fiat";  
    // aqui si puedo usar carName2  
function myFunction() {  
    // aqui tambien puedo usar la variable carName2  
}
```

JS

En este caso podremos acceder al contenido de esa variable tanto desde fuera como desde adentro de la función

[Ver segundo ejemplo de scope \(.html y .js\)](#)

Como vimos en el tema “variables”, en JavaScript, los objetos y las funciones también son variables. El alcance determina la accesibilidad de variables, objetos y funciones de diferentes partes del código.

Scope (alcance)

Automáticamente global

Si asignamos un valor a una variable que no ha sido declarada, automáticamente se convertirá en una variable **global**.

Este ejemplo de código declarará una variable global carName, incluso si el valor se asigna dentro de una función.

```
myFunction();  
// aquí puede se puede usar carName  
function myFunction() {  
    carName = "Volvo"; // variable no declarada  
}
```

JS

*En este caso podremos acceder al contenido de esa variable tanto desde fuera como desde adentro de la función, por ser **automáticamente global**.*

Ver tercer ejemplo de scope (.html y .js)

La vida útil de una variable comienza cuando se declara. Las variables locales se eliminan cuando se completa la función.

let y var

La instrucción **let** declara una variable de alcance local con ámbito de bloque, la cual, opcionalmente, puede ser inicializada con algún valor.

let te permite declarar variables limitando su alcance (scope) al bloque, declaración, o expresión donde se está usando, a diferencia de la palabra reservada **var** la cual define una variable **global o local** en una función *sin importar el ámbito del bloque*.

```
//let vs var
var a = 5;
var b = 10;

if (a === 5) {
  let a = 4; // El alcance es dentro del bloque if
  var b = 15; // El alcance es global, sobrescribe a 10

  console.log(a); // 4, por alcance a nivel de bloque
  console.log(b); // 1, por alcance global
}

console.log(a); // 5, por alcance global
console.log(b); // 1, por alcance global
```

JS

Fuente:

<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Statements/let>

Ver cuarto ejemplo de scope
(.html y .js)

Funciones | Información adicional

Funciones básicas:

<https://lenguajejs.com/javascript/introduccion/funciones-basicas/>

Curso Básico de Javascript 9.- Funciones:

https://www.youtube.com/watch?v=AvMFiQl7AU0&list=PLhSj3UTs2_yVC0iaCGf16glrrfXuiSd0G&index=9

Fundamentos sobre funciones:

<https://lenguajejs.com/javascript/fundamentos/funciones/>

Funciones | Información adicional

Funciones Flecha:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Arrow_functions

Funciones Arrow (de Flecha) Javascript 2018:

https://www.youtube.com/watch?v=eXwEYSRk73U&ab_channel=Bluuweb%21

Qué es una función de flecha - JavaScript Arrow Functions:

https://www.youtube.com/watch?v=aIKL5tQP25Y&ab_channel=DominiCode

JavaScript Arrow Function (W3Schools)

https://www.w3schools.com/js/js_arrow_function.asp

Ejercicios

- Del archivo **“Actividad Práctica - JavaScript Unidad 2”** están en condiciones de hacer los ejercicios: 1 al 18. Los ejercicios **NO** son obligatorios.

Callback (devolución de llamada)

Las funciones en JavaScript son objetos. Como cualquier otro objeto, puede pasarlos como parámetro. Por lo tanto, en JavaScript podemos **pasar una función como argumento de otra función**. Esto se llama función de devolución de llamada (*callback*). Las funciones también se pueden devolver como resultado de otra función.

```
function saludar(nombre) {  
    alert('Hola ' + nombre);  
}  
  
function procesarEntradaUsuario(callback) {  
    var nombre = prompt('Por favor ingresa tu nombre.');
```

JS

```
    callback(nombre);  
}  
procesarEntradaUsuario(saludar);
```

El ejemplo anterior es una callback sincrónica, ya que se ejecuta inmediatamente.

¿Qué es un callback en JavaScript?:

https://www.youtube.com/watch?v=DaXuPcdKqQ4&ab_channel=CodigoMentor

Callbacks: <https://lenguajejs.com/javascript/fundamentos/funciones/#callbacks>

Clousure (cierre)

Un cierre es una variable o función local que usa otra función y las referencias a la función se devuelven a la función. Es decir, devolvemos una función en una función externa que hace referencia a las variables locales de la función externa. Esto es posible si tenemos funciones anidadas en otra función y devueltas como referencia. En la función interna, podemos usar las variables de la función externa. Debido al alcance de las variables locales, las funciones internas pueden acceder a las variables de la función externa. Cuando devolvemos la función interna en la función externa, las referencias a las variables locales de la función externa todavía están referenciadas en la función interna.

JS

```
function iniciar() {  
  var nombre = "Codo a Codo"; // La variable nombre es una variable local creada por  
  iniciar.  
  function mostrarNombre() { // La función mostrarNombre es una función interna, una cl  
    ausura.  
    alert(nombre); // Usa una variable declarada en la función externa.  
  }  
  mostrarNombre();  
}  
iniciar();
```

La función **iniciar()** crea una variable local llamada **nombre** y una función interna llamada **mostrarNombre()**. Por ser una función interna, esta última solo está disponible dentro del cuerpo de **iniciar()**. Notemos a su vez que **mostrarNombre()** no tiene ninguna variable propia; pero, dado que las funciones internas tienen acceso a las variables de las funciones externas, **mostrarNombre()** puede acceder a la variable **nombre** declarada en la función **iniciar()**.