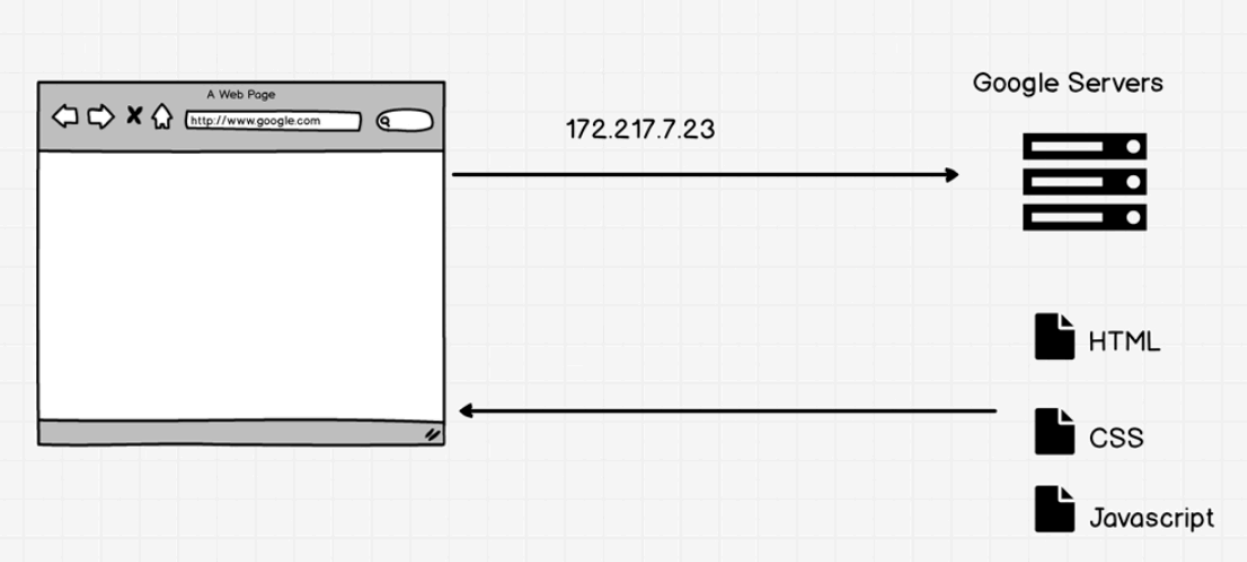
JavaScript: Functions

The Complete Web Developer in 2018

The Complete Web Developer in 2018
Zero to Mastery
Andrei Neagoie
Lecture Notes by Stephanie

Add Javascript to Webpage



HTML file (index.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Javascript</title>

    // link to css file
    <link rel="stylesheet" type="text/css" href="">

</head>
<body>

    <h1>Javascript in HTML</h1>

    // links to javascript file
    <script type="text/javascript" src="script.js">
    </script>

</body>
</html>
```

Can also have multiple javascript links as follows:

```
<!DOCTYPE html>
<html>
<head>
    <title>Javascript</title>
    <link rel="stylesheet" type="text/css"
    href="">
</head>
<body>
    <h1>Javascript in HTML</h1>
    <script type="text/javascript" src="
    script.js">
    </script>
    <script type="text/javascript" src="
    script2.js">
    </script>
    <script type="text/javascript" src="
    script3.js">
    </script>
</body>
</html>
```

We put the javascript link in the end of <body> so that the page renders before pulling the javascript file

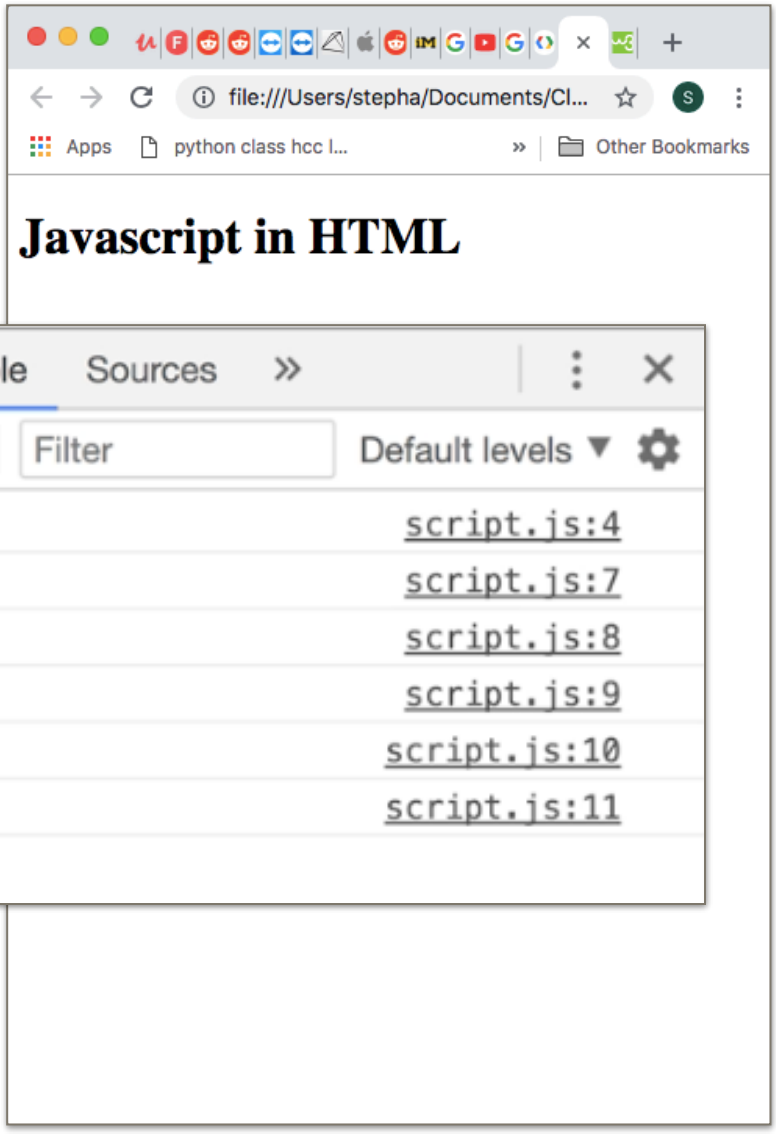
Javascript file (script.js)

```
4 + 3;

if (4+3 === 7) {
    console.log("Hellooooo");
}

console.log("Hellooooo");
console.log("Hellooooo");
console.log("Hellooooo");
console.log("Hellooooo");
console.log("Hellooooo");
```

Use console.log( ) to print to the console of webpage



Javascript Functions

examples:

alert( )
prompt( )
console.log( )

These functions come with javascript

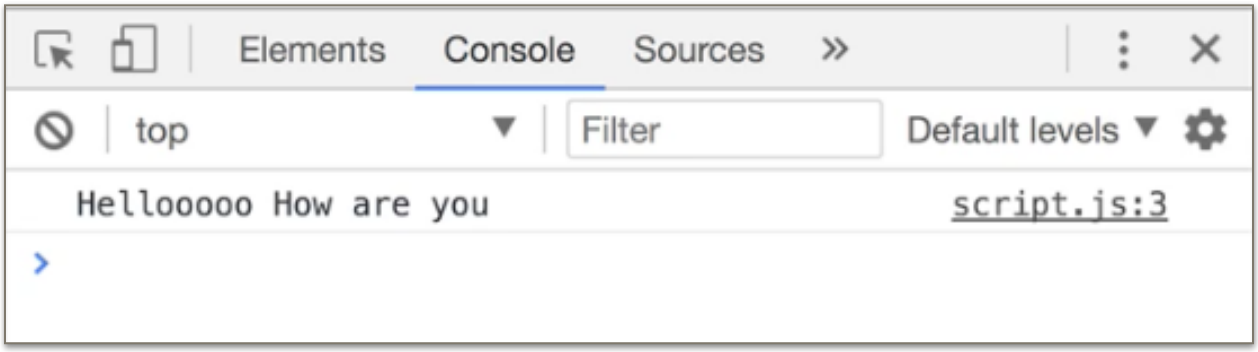
Use parentheses ( ) to call the function
Arguments given to funxn in parentheses

Javascript file (script.js)

```
console.log("Hellooooo", "How are you");
```

1st argument 2nd argument

Console when webpage loads



Javascript Functions

function name( ) { }
var a = function name( ) { }
return
( ) => (new in ECMAScript 6)

Function Declaration: function name( ) { }

```
> function sayHello(){
      console.log("Hello");
  }
<· undefined
```

Call the function

```
> sayHello()
  Hello                                    pathturbo.js:1
```

Function Expression: var a = function name( ) { }

"anonymous function": funxn assigned to var but has no name

```
> var sayBye = function() {
      console.log("Bye");
  }
<· undefined
```

Call the function

```
> sayBye()
  Bye                                          pathturbo.js:1
<· undefined
```

Can name the function byeBye( ), but it has limited use
Call the function

```
> var sayBye = function byeBye() {
      console.log("Bye");
  }
```

```
> sayBye()
  Bye                                          pathturbo.js:1
<· undefined
> byeBye()
✕ ▸Uncaught ReferenceError: byeBye is not defined VM52366:1
      at <anonymous>:1:1
```

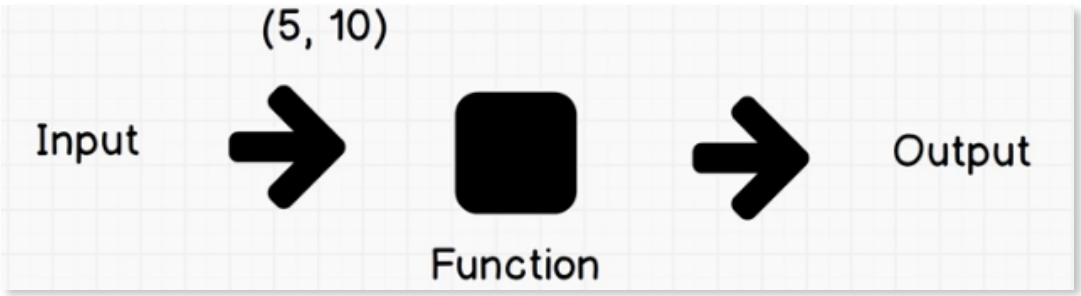
DRY - “Don’t Repeat Yourself”

Use arguments to make functions more “extensible”

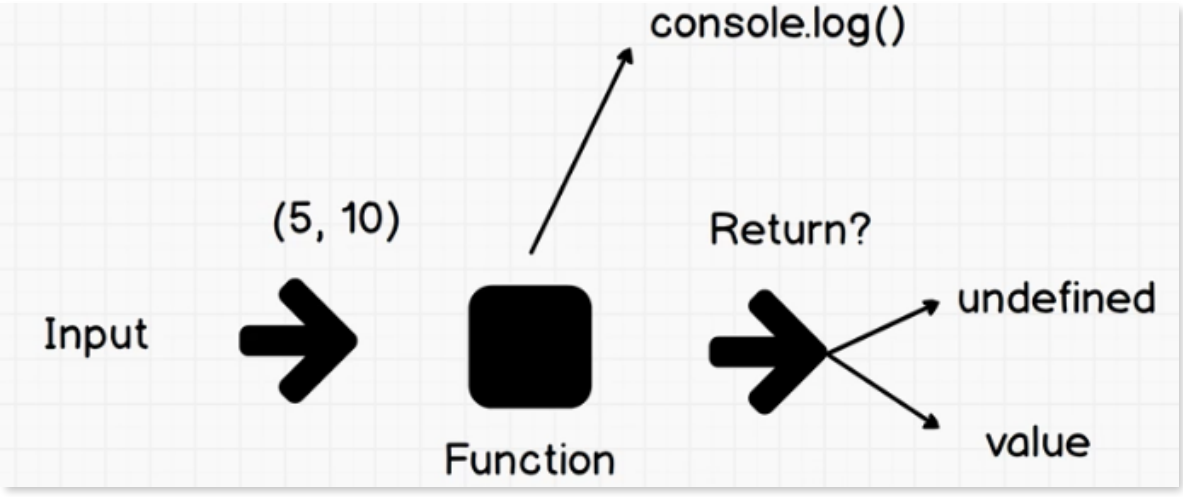
```
> function sing(song){
      console.log(song);
  }

  sing("La dee da");
  sing("Cardiiiiii");
  sing("Rave alarmmmmmmmm");
```

```
  La dee da                                    pathturbo.js:1
  Cardiiiiii                                   pathturbo.js:1
  Rave alarmmmmmmmm                            pathturbo.js:1
<· undefined
```



Function can create console log, return a value, or not return anything (undefined)



Use **return** to have function return value

```
> function multiply(a, b){
      return a * b;
  }
```

```
> multiply(5, 10);
<- 50
```

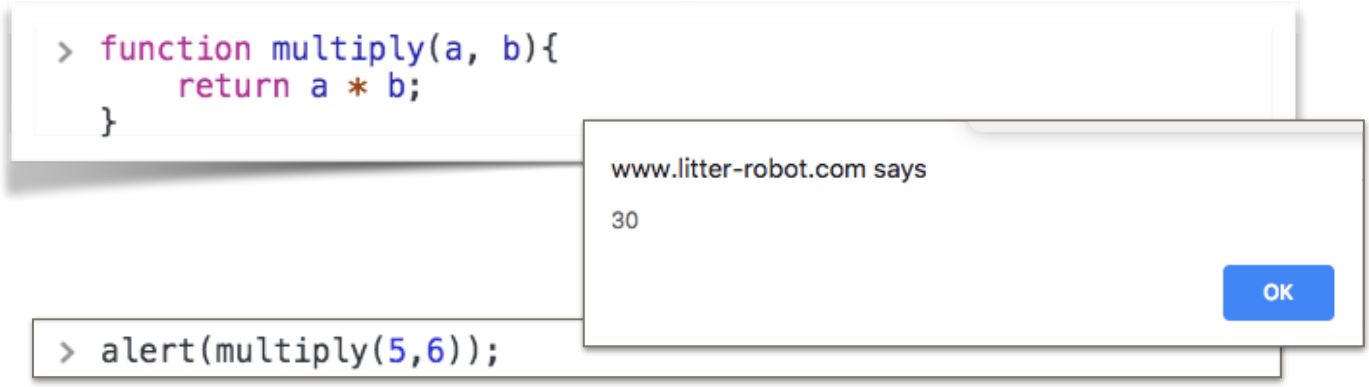
Functions always exit after first return.
Returns a and then exits. Other two lines of code not read at all.

```
> function multiply(a, b){
      return a;
      return a * b;
      return b;
  }
```

```
> multiply(5, 10);
<- 5
```

Use multiple returns in situations like:

```
> function multiply(a, b){
      if (a > 10 || b > 10) {
          return "that's too hard";
      } else {
      return a * b;
      }
  }
<- undefined
> multiply(5, 10);
<- 50
> multiply(15, 10);
<- "that's too hard"
```



Inner functions - nest functions, alert(multiply( ))

Parameters vs arguments:
Parameters - a, b
Arguments - 5, 6

// Exercise 5
// Make a keyless car EVEN BETTER!
// We are improving our car from previous exercise now.

```
var age = prompt("What is your age?");

if (Number(age) < 18) {
        alert("Sorry, you are too young to drive this car. Powering off");
} else if (Number(age) > 18) {
        alert("Powering On. Enjoy the ride!");
} else if (Number(age) === 18) {
        alert("Congratulations on your first year of driving. Enjoy the ride!");
}
```

//1. Make the above code have a function called checkDriverAge(). Whenever you call this function, you will get prompted for age. Use **Function Declaration** to create this function.

```
> function checkDriverAge() {
      var age = Number(prompt("What is your age?"));
      if (age < 18) {
          alert("too young");
      } else if (age > 18) {
          alert("you may drive");
      } else if (age === 18) {
          alert ("happy 18th bday");
      }
  }
```

Notice the benefit in having checkDriverAge() instead of copying and pasting the function everytime?

//2. Create another function that does the same thing, called checkDriverAge2() using Function Expression.

```
> var checkDriverAge2 = function() {
      var age = Number(prompt("What is your age?"));
      if (age < 18) {
          alert("too young");
      } else if (age > 18) {
          alert("you may drive");
      } else if (age === 18) {
          alert ("happy 18th bday");
      }
  }
```

//BONUS: Instead of using the prompt. Now, only use the return keyword and make the checkDriverAge() function accept an argument of age, so that if you enter:
checkDriverAge(92);
it returns "Powering On. Enjoy the ride!”

```
> function checkDriverAge(age) {
      if (age < 18) {
          return "too young";
      } else if (age > 18) {
          return "you may drive";
      } else if (age === 18) {
          return "happy 18th bday";
      }
  }
```

```
> checkDriverAge(10)
<- "too young"
> checkDriverAge(18)
<- "happy 18th bday"
> checkDriverAge(28)
<- "you may drive"
```

Advanced JavaScript Functions

Function Closures

Closure is a function that access to the parent scope, even after the parent function has closed.

Can use closures to create **private variables** (aka **local**) that are not accessible in the root scope.

Child scope always has access to parent scope.
Parent scope does not have access to child scope

Parent function runs only once.

```
function parentFunc () {        // parent funxn declaration
      var greet = 'Hi';         // private var
      console.log("ParentFunc Run")
      function childFunc() {    // child funxn declaration
            alert(greet);       // access private var (child can access parent scope)
            console.log("ChildFunc Run")
      }
      return childFunc;                 // parent funxn returns child funxn declaration
}

var newFunc = parentFunc(); // runs parentFunc() & returns childFunc declaration
                            // stores childFunc declaration in newFunc

newFunc();                  // executes childFunc(), which has access to parent scope
```

Every time we run newFunc()….

```
> newFunc()
  ChildFunc Run                                                    VM2218:6
```

ES5/6 syntax:

```
const first = () => {
	const greet = 'Hi';
	const second = () => {
		alert(greet);
	}
	return second;
}

const newFunc = first(); // parent funxn runs once and returns child funxn
newFunc(); // invokes child funxn, which still has access to parent scope
```

Example: Create a counter that can be incremented using a function, add(). We want counter variable to be local to the add() so it cannot be changed from the root scope

```
function parentAdd() {
	var counter = 0;    // private var
	console.log("counter start: ",counter);

	function childAdd(){
		counter +=1;
		console.log("counter: ",counter);
	}

	return childAdd;
}

var Add = parentAdd();
Add();    // increment counter by one
```

```
counter start:  0                                    VM2261:3
counter:  1                                          VM2261:7
<· undefined
> Add()
counter:  2                                          VM2261:7
<· undefined
> Add()
counter:  3                                          VM2261:7
<· undefined
```

Function Currying

Currying converts a function that takes multiple arguments into a function that takes them one at a time

Instead of

```
function multiply(a, b) {
  return a * b;
};
```

We do...

```
function curriedMultiply(a) {
  return function (b) {
    return a * b;
  };
};
```

ES5/6 syntax:

```
const multiply = (a, b) => a * b;
const curriedMultiply = (a) => (b)
=> a * b;
curriedMultiply(3)(4);
```

To use it..

```
> curriedMultiply(3)(4)
<- 12
```

Allows for extensibility.

```
var multiplyBy5 = curriedMultiply(5);
```

```
> multiplyBy5(10)
<- 50
> multiplyBy5(20)
<- 100
```

Compose Functions

Compose is the act of putting two functions together to form a third function, where the output of one function is the input of the other.

```
function compose(f, g) {
  return function (a) {
    return f(g(a));
  };
};

function addOne(num) {
  return num + 1;
};

function squareIt(num) {
  return num * num;
};
```

ES5/6 syntax:

```
const compose = (f, g) => (a) => f(g(a));
// f is a funxn
// g is a funxn
// a is num

const addOne = (num) => num + 1;
const squareIt = (num) => num * num;
```

Examples:

```
> compose(addOne, addOne)(5);
<- 7
```

(5+1)+1 = 7

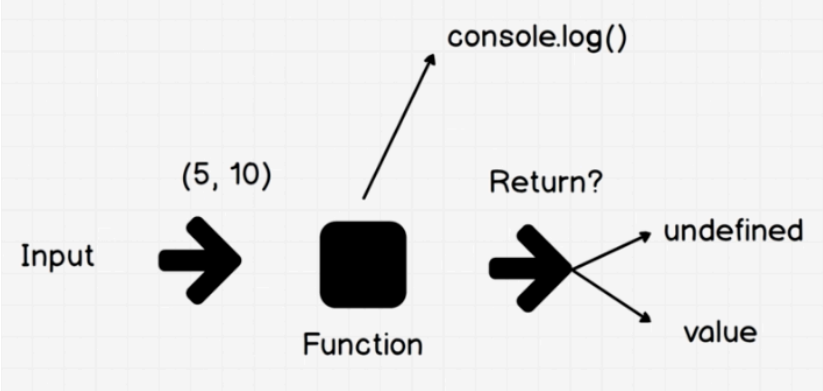
```
> compose(addOne, squareIt)(5)
<- 26
```

(5*5)+1 = 26

```
> compose(squareIt, addOne)(5)
<- 36
```

(5+1)*(5+1) = 36

Side Effects + Functional Purity



Function does something that we define. We want each function as it's own universe with it's own scope. It returns a value or returns 'undefined'.

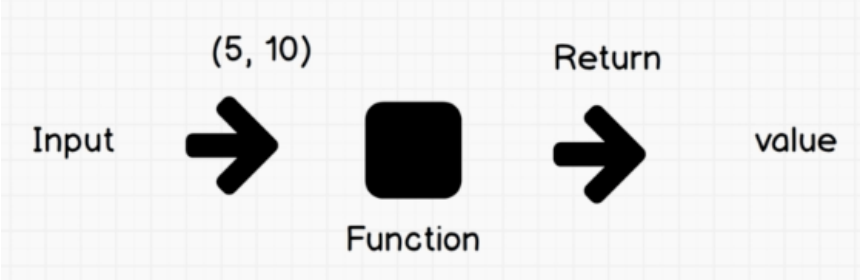
Side Effects are actions that affect the outside world, such as console logs, writing to an external variable, interacting with the rest of the program, etc. Function does not depend on any state, or data, change during a program's execution. It must only depend on its input elements. It's best to avoid side affects

```
var a = 1;
function b() {
      a = 2; // has side effect!
}
```

Deterministic - given a particular input, function will always produce the same output (this is best, reduces bugs and errors)

Pure Functions - avoid side effects, deterministic

Best practices is to create pure functions that minimize side effects and are deterministic!



Exercise: Advanced Functions

Section 13, Lecture 140

It's time to code some javascript! Get your sublime text ready for this exercise, and use Google Chrome javascript console to test your code. You can find the exercise file and the solution file attached. Good luck!

Resources for this lecture

exercise4-SOLUTIONS.js
exercise4.js

```
//#1 Create a one line function that adds adds two parameters
const AddFunc = (a, b) => a + b

//What does the last line return?
const addTo = x => y => x + y
var addToTen = addTo(10)
addToTen(3)
```

```
13
```

```
//Currying: What does the last line return?
const sum = (a, b) => a + b
const curriedSum = (a) => (b) => a + b
curriedSum(30)(1)
```

```
31
```

```
//Currying: What does the last line return?
const sum = (a, b) => a + b
const curriedSum = (a) => (b) => a + b
const add5 = curriedSum(5)
add5(12)
```

```
17
```

```
//Composing: What does the last line return?
const compose = (f, g) => (a) => f(g(a));
const add1 = (num) => num + 1;
const add5 = (num) => num + 5;
compose(add1, add5)(10)
```

```
<- 16
```

Pure Functions

What are the two elements of a **pure function**?

1. **Deterministic** -- always produces the same results given the same inputs

2. **No Side Effects** -- It does not depend on any state, or data, change during a program's execution. It must only depend on its input elements.

(see notes on prior to exercise for detailed explanation)