Learning Objectives

- **Interact with the console by implementing the `error`, `warn`, `count`, `log`, `time`, and `timeEnd`**
- **Compare and contrast functions in Python and JavaScript**
- **Create functions using the `function` keyword, assigning a function to a variable, and using the arrow syntax**
- **Use `setTimeout` to wait before running a function**
- **Create for loops, while loops, do-while loops, and iterate using `forEach`**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Console Logging

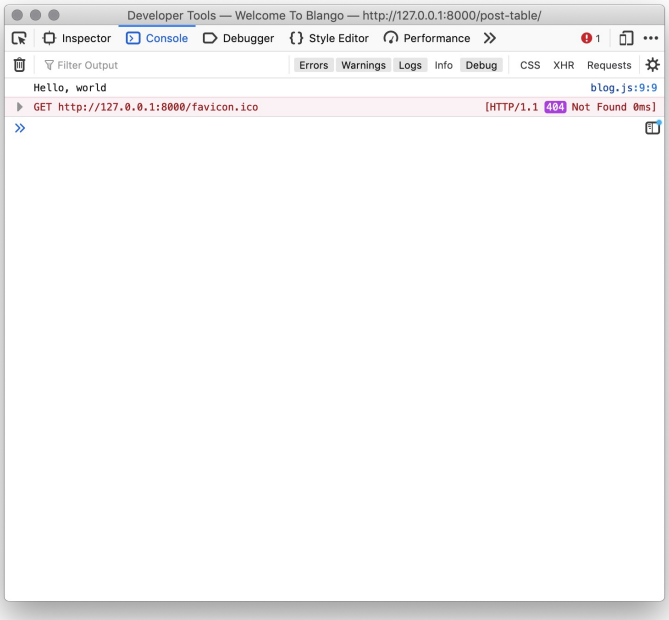
Console Logging

In this section we're going to cover JavaScript functions, and some loops. This will involve outputting some larger amounts of data than we have been already. It would get tedious to go through and dismiss the alert dialog for each message we want to output, so instead we'll be logging to the browser console.

`console` is a special global variable in JavaScript that is an instance of the browser's console, a place to log messages, among other things. Normally to output a message you'd call its `log()` method, like so:

```
console.log('Hello, world')
```

This will display a message under the general console.



console message

But there are some other methods that are also useful:

- `error()`: Output a message at *error* level.
- `warn()`: Output a message at *warning* level.
- `count()`: Called with a label (e.g. `console.count('foo')`) and will output

a count of how many times it has been called with that label.
- `time()`: Called with a name to start a named timer. (e.g. `console.time('myTimer')`), and then to output the time since the start, use…
- `timeEnd()`: Call this with the same label to output the time, in milliseconds, since the corresponding `time()` method was called.

Try It Out

Let's make sure you can find your browser console and see the messages before we get into JavaScript functions. You'll update the `blog.js` file to add a few different uses of the `console` methods to see how they are output.

In the `blog/static/blog/blog.js` file to the left, replace its contents with this code:

```
console.time('myTimer')
console.count('counter1')
console.log('A normal log message')
console.warn('Warning: something bad might happen')
console.error('Something bad did happen!')
console.count('counter1')
console.log('All the things above took this long to happen:')
console.timeEnd('myTimer')
```

Then save it and refresh the `/post-table/` page. You'll see a page with a header but not much else.

View Blog

The method of showing the JavaScript console depends on your browser and operating system.

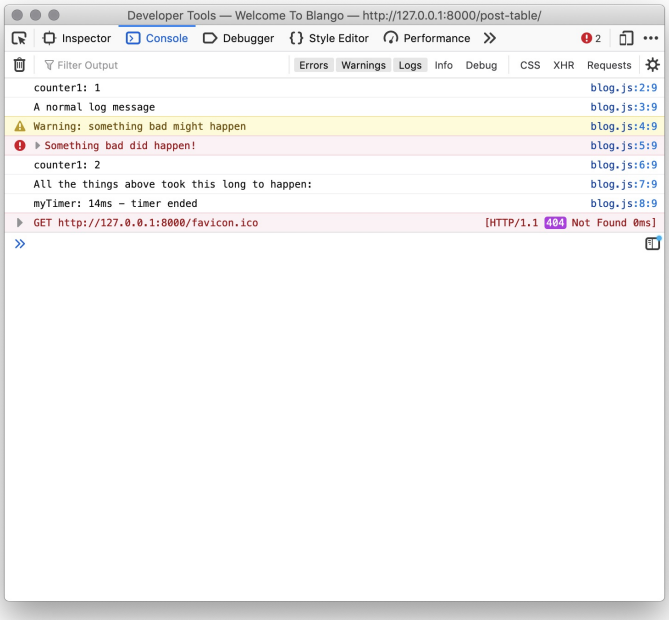
- *Chrome* - On a Mac, it can be found under the **View** menu, then **Developer**, then **JavaScript Console**. On Windows, in the **three dot menu** of the browser choose **More Tools**, then **Developer tools**. You can also press the **F12** key to open the Console.
- *Firefox* - On a Mac, it can be found under the **Tools** menu, go to **Browser Tools** then **Web Developer Tools**. Then, select the **Console** tab. On Windows, click on **Web Development** and then **Web Console**. You can also press the **F12** key.
- In *Safari*, you'll need to enable the **Develop** menu. In the Safari preferences, select the **Advanced** tab then check **Show Develop menu in the menu bar**. Once you've done that, just choose **Show JavaScript Console** from the **Develop** menu.

▼ **Chrome Hotkeys**

Hotkeys are combinations of key strokes to perform a task normally done with a mouse. Here are the hotkeys for opening the console in Chrome:

- Mac - Press Command+Option+J
- Windows, Linux, or Chrome OS - Control+Shift+J

When you've found the console, you should see the messages similarly to the next screenshot (from Firefox):



multiple console messages

Notice how our `counter1` has incremented with each call, how `log` messages display differently from `warn` and `error` messages, and how `error` messages can be expanded to show a stack trace. At the end, we see how long it took between the `time()` and `timeEnd()` calls (14ms in this case).

You'll also notice that some network errors are also displayed. Your browser may try to load a Favicon (the little icon displayed in tabs and the location field) for our site but fails since we don't have one. If you see this error, it is not an issue and we can ignore it.

▼ **Favicon error message**
Although the error isn't actually a problem, if you want to fix it you can add this code inside the

of your `base.html` template. It encodes an empty PNG as base-64 and includes it inline, so the browser will render this instead of trying to fetch a favicon.

```
<link rel="icon" href="data:;base64,iVBORw0KGgo=">
```

Now let's look at functions.

JavaScript Functions

Functions

In JavaScript there are a number of ways to define functions. The first way is similar to Python, using the `function` keyword which is similar to `def` in Python. Function bodies are contained inside curly braces (`{}`) and indentation doesn't matter. Even though the lack of or inconsistent use of indentation will not cause an error, you should continue to use indentation to help with legibility. Also like Python, values can be returned from functions using `return`. Here's a function in Python that adds two numbers together and returns the sum:

```
def add_numbers(a, b):
    return a + b
```

Here's the equivalent in JavaScript:

```
function addNumbers(a, b) {
    return a + b
}
```

You'll notice that in JavaScript the convention for variables and function names is `camelCase`, whereas in Python we'd usually use `snake_case`.

We've already called a few functions, like `alert()` and `console.log()`, so there should be no surprises in calling a function:

```
const result = addNumbers(3, 4)
console.log(result)
```

This will output `7` to the browser console.

Arguments to JavaScript functions are implicitly all optional. For example, in Python we would implement optional arguments like this:

```python
def say_hello(name=None):
    if name is None:
        print("Hello, no name")
    else:
        print("Hello, " + name)

say_hello()  # name is None
say_hello("Lily")  # name is "Lily"
```

In JavaScript, we need to check if a variable is `===` to `undefined`, which means it hasn't been passed in. This JavaScript function is roughly identical to the Python one:

```javascript
function sayHello(name) {
  if (name === undefined) {
      console.log('Hello, no name')
  } else {
      console.log('Hello, ' + name)
  }
}

sayHello()  // name is undefined
sayHello('Lily')  // name is 'Lily'
```

Functions can also be assigned to variables. Another way of defining the `sayHello()` function is like this:

```javascript
const sayHello = function(name) {
    if (name === undefined) {
        console.log('Hello, no name')
    } else {
        console.log('Hello, ' + name)
    }
}

sayHello('Lily')
```

Notice the function is called in the same way. Defining functions in this way is useful if you want to nest functions inside other functions and follow the variable scoping rules that we saw in the last section. Or, if you want to reassign functions to different variables (assuming you used `let` instead of `const` when assigning the function initially).

The third way of defining functions is to use anonymous or *arrow* functions, so called because they're denoted with the `=>` ("arrow") operator, and they'll be anonymous or unnamed unless they're assigned to a variable. Here's the `sayHello` function again:

```
const sayHello = (name) => {
  if (name === undefined) {
      console.log('Hello, no name')
  } else {
       console.log('Hello, ' + name)
  }
}
```

What's the difference between using `=>` or the `function` keyword? The context in which the function executes is different. We'll come back to this later when dealing with classes, as it's mostly only important then.

Using the arrow function also allows us to define lambda functions, that is, basic one-line functions that implicitly return a value. These are defined without the use of curly braces or the `return` keyword.

For example, here's an arrow function that doubles a number:

```
const doubler = (x) => { return x * 2 }

console.log(doubler(2))  // outputs 4
```

And here's the equivalent as a lambda:

```
const doubler = x => x * 2
```

Note that lack of curly braces or `return`. When using a single argument, the parentheses around the argument are optional.

Let's look at a practical example of when anonymous functions are useful. They're very short to write as callbacks, and a lot of JavaScript code is callback-based.

JavaScript provides a `setTimeout()` function. This will wait a number of milliseconds and then call a function that is passed to it. In this example, `setTimeout` wait 2 seconds (2,000 ms) and then calls the `showAnAlert()` function:

```
function showAnAlert() {
  alert('Timeout finished.')
}


setTimeout(showAnAlert, 2000)
```

This works OK for this function that takes no arguments, but how can we use `setTimeout` if a function needs arguments? A common pattern in JavaScript is to pass anonymous functions and use them as wrappers.

Let's return to our `sayHello()` function again. Let's say we want to wait 2 seconds, then greet the user with their name. We could do something like this:

```
const name = 'Ben'

setTimeout(() => {
    sayHello(name)
  }, 2000
)
```

Here we're defining an anonymous function that takes no arguments (the empty parentheses `()` before the `=>`), and immediately passing it to `setTimeout()`. When it's called, it accesses the `name` variable from the outer scope, and passes it to `sayHello()`.

Let's see this in action.

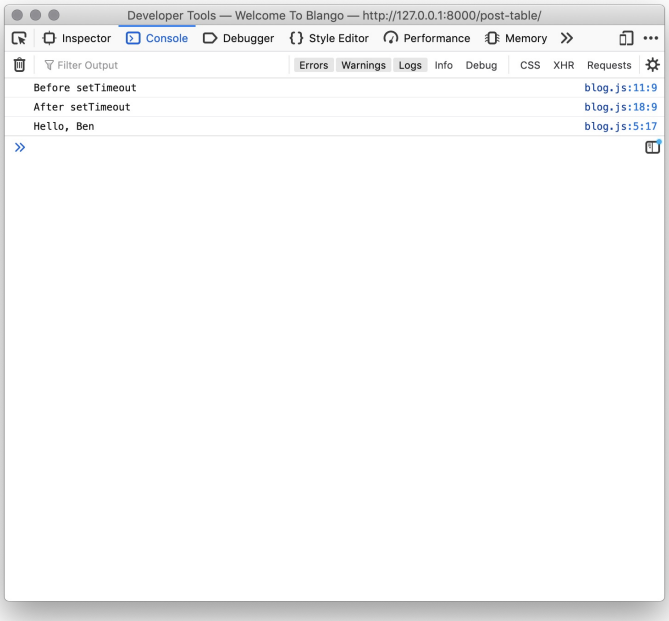
Try It Out

Delete the contents of the `blog.js` file in your Blango project. Insert this instead:

```
function sayHello(yourName) {
  if (yourName === undefined) {
      console.log('Hello, no name')
  } else {
       console.log('Hello, ' + yourName)
  }
}

const yourName = 'Your Name'  // Put your name here

console.log('Before setTimeout')

setTimeout(() => {
    sayHello(yourName)
  }, 2000
)

console.log('After setTimeout')
```

Load the `/post-table/` page and open the JavaScript console.

View Blog



console with set timeout

Notice how the anonymous function has accessed the `yourName` variable in the outer scope. Also pay attention to the order in which the messages are printed to the console. The execution doesn't stop when `setTimeout()` is

called. It continues on and the `After setTimeout` message is displayed immediately. Two seconds later, the `sayHello()` function is called and the greeting is displayed.

This illustrates another JavaScript paradigm, which is the judicious use of callback functions and asynchronous code, which we will see more of in the section on *promises*.

Now let's have a quick look at some of the ways of looping in JavaScript.

Loops

Loops

Let's have a look at some of the ways of looping in JavaScript.

For loop

The first is to use a `for()` loop, which you'll be familiar with if you've used other programming languages like C or Java.

A for loop consists of four parts:

- An initializer, which is executed when the loop starts.
- A condition, which is checked before each loop iteration. The loop only continues if the condition is true, otherwise the loop stops.
- An advancement statement, which is executed at the end of each iteration to change the state of variables.
- A loop body that is executed each iteration.

All parts of the for loop are optional.

Here's a for loop that initializes with the statement `let i = 0`, which declares the variable `i` and sets it to `0`. The condition is `i < 10`, which means the loop will run until `i` equals 9. The advancement is `i = i + 1`, that is, `i` is incremented after each iteration:

```
for(let i = 0; i < 10; i += 1) {
  console.log(i)
}
```

This prints the numbers `0` through `9` (inclusive) to the browser console.

While loop

`while()` loops behave the same as their Python equivalent. The loop body is executed until the condition is false. This `while()` loop behaves the same as the previous for loop we saw.

```
let i = 0

while(i < 10) {
  console.log(i)
  i += 1
}
```

Do-while loop

Do-while loops are like while loops, in that they execute until a condition is false, the difference is they will always be executed at least once. The condition comes at the end of the loop body. In this next example, the number `10` is printed once. Even though the condition is false, the loop body is already executed once before it is checked.

```
let i = 10

do {
  console.log(i)
  i += 1
} while(i < 10)
```

Other iterators

Instead of writing loops, we can iterate over arrays, which can be more idiomatic and a little bit more like Python. The two ways of doing this are the `forEach()` and `map()` methods. `forEach()` executes a function for each item in the array. `map()` also executes a function for each item in the array, but returns a new array of the same size, which contains the result(s) of the function.

Here's how we could print the numbers `0` to `9` using `forEach()`.

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers.forEach((value => {
  console.log(value)
}))
```

Here's how we could use `map` to double each of our `numbers`, into a new array called `doubled`.

```
const doubled = numbers.map(value => value * 2)

console.log(doubled)
```

Notice here the use of an anonymous, lambda function.

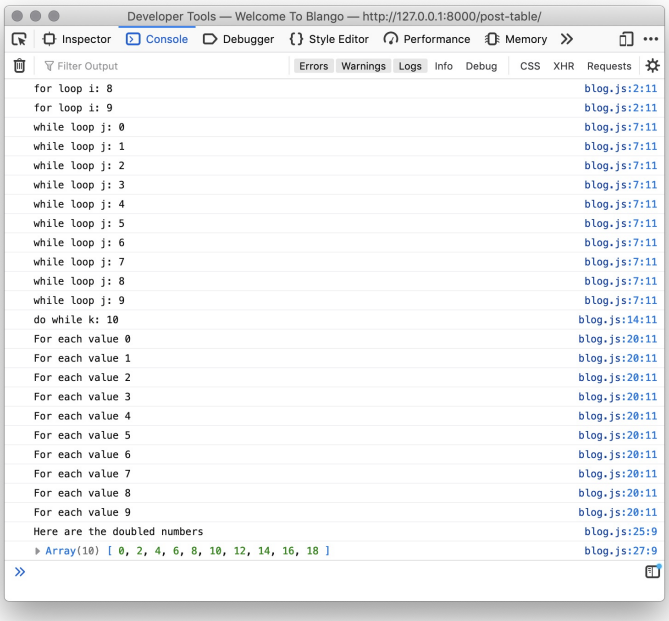
Try It Out

Now you can try out some loops and iterators for yourself. Open `blog.js` and delete its content, then insert this:

```
for(let i = 0; i < 10; i += 1) {
  console.log('for loop i: ' + i)
}

let j = 0
while(j < 10) {
  console.log('while loop j: ' + j)
  j += 1
}

let k = 10

do {
  console.log('do while k: ' + k)
} while(k < 10)

const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers.forEach((value => {
  console.log('For each value ' + value)
}))

const doubled = numbers.map(value => value * 2)

console.log('Here are the doubled numbers')

console.log(doubled)
```

Open `/post-table/` in a browser console and you should have a lot of output.

View Blog



console loop output

This has covered the basics of functions and flow control with loops, with a small taster of asynchronous programming.

In the next section we'll look at classes in JavaScript.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish JavaScript functions"
```

- Push to GitHub:

```
git push
```