Learning Objectives

- **Add a script to a Django template**
- **Create an alert**
- **Differentiate between `let` and `const`**
- **Contrast JavaScript objects and Python dictionaries**
- **Explain the difference between `==` and `===`**
- **Create a conditional in JavaScript**
- **Add a comment in JavaScript**

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.

Including JavaScript on a Page

Intro

JavaScript is a language primarily for running code inside a web browser. Over the past few years it has been gaining popularity as a server-side language, thanks to the NodeJS project. Nowadays, it can even be used to build native mobile apps using projects like React Native. We're going to keep it in the browser. We'll start with JavaScript fundamentals and work our way towards using the React framework. Let's start out with variables.

Including JavaScript on a Page

There's actually something more fundamental to cover before looking at variables: how do we actually write and execute JavaScript code? There are two ways that you'd usually use to get JavaScript onto a web page. You can either write it inline inside an HTML file, inside a `<script>` element. For example:

```
<script>
    console.log('Hello, world!')
</script>
```

Or, you can include JavaScript from an external file into your page using the `src` attribute on a `<script>` tag instead. This would include a file at the path `/static/js/script.js`.

```
<script src="/static/js/script.js"></script>
```

JavaScript is executed in the order in which it's written on the page (unless you start using special attributes like `async` and `defer`, but they are for advanced usage, so we won't go into them). You can assume that each script tag is executed in its entirety before the next script tag. They're also loaded around/between other parts of the page being loaded. For example, if you included a `<script>` tag at the start of the page which tried to reference elements in the footer of the page, it would fail, as the footer of the page would not have been parsed when the script was loaded. For this reason `<script>`s are often included at the end of the page, or at least after any elements that they might need to refer to.

In this module, we’re going to create an interactive table to list all blog posts. We’ll render the table using React and fetch the data for it from our DRF API.

Let’s start by setting up some of the things we need to start working with JavaScript: a template, view, URL pattern and JavaScript file.

Try It Out

Start by creating a new template in the `blog/templates/blog/` directory, called `post-table.html`. We want it to inherit from `base.html`, then load the `static` template tag library. Then inside the `content` block, it will have a `<script>` tag, which loads a JavaScript file we’ll write soon. You can copy and paste this content to build the template file:

```
{% extends "base.html" %}
{% load static %}
{% block content %}
    <script src="{% static "blog/blog.js" %}"></script>
{% endblock %}
```

Next you can create the view to render this template, we don’t need to pass any context variables to it. Open `blog/views.py` and create a new view function named `post_table`. It should render the template you have just created, like this:

Open blog/views.py

```
def post_table(request):
    return render(request, "blog/post-table.html")
```

We’ll need to add a URL pattern to point to this function. It can be added in `blango/urls.py`, at the path `post-table/`. Set up a rule like this:

Open urls.py

```
    path("post-table/", blog.views.post_table, name="blog-post-
        table"),
```

We’ll finish up by writing the `blog.js` file. It will be served from the directory `blango/blog/static/blog/`, so you’ll need to create both the `static` directory and the `blog` directory inside it. Then, create the `blog.js` file inside that.

Open blog.js

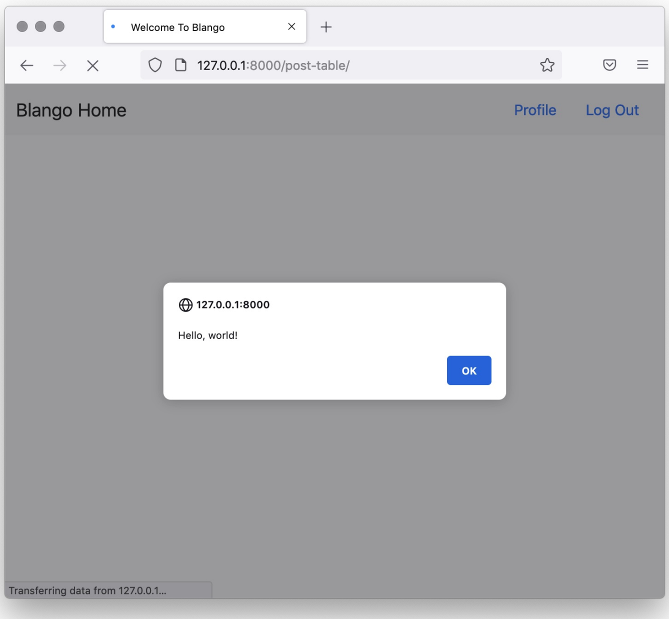
Make the contents of `blog.js` this single line:

```
alert('Hello, world!')
```

In the file, we're calling the JavaScript `alert()` function which takes a single string argument, and displays it as an alert dialog box in the browser.

View Blog

To try it out and check everything is working, just visit the `/post-table/` page in your browser. You should see the alert as in the next screenshot:


Welcome To Blango
127.0.0.1:8000/post-table/
Blango Home
Profile
Log Out
127.0.0.1:8000
OK
Transferring data from 127.0.0.1...

Hello, world!

alert in browser

(Although it may look a bit different depending on your browser.)

Click the button (which might say **OK**, **Close**, or something else depending on your browser) to close the alert.

Now we can get back to talking about JavaScript variables.

Variables in JavaScript

Variables in JavaScript

There are two ways to define variables in JavaScript, using either the `let`, for variables whose value might change, or `const`, for variables whose value won't. Like in Python, variables aren't typed, so their types can be reassigned.

▼ **Declaring variables with var**

You can also assign variables with the keyword `var`, but this is an old way of doing it and not recommended anyway. Variables assigned with `var` are not scoped like `let` and `const`, we'll see an example of this variable scoping shortly.

We'll show some examples which will also introduce a few of the JavaScript syntax features as well.

This is valid code:

```
const framework = 'Django'
const language = 'Python'
alert(framework + ' is written in ' + language)
```

Like in Python, strings can use double or single quotes. The choice is personal preference; we'll use single quotes. Also notice how strings are joined, like in Python, they are concatenated with a `+`. Lines in JavaScript can optionally be terminated with a semicolon (`;`) but we won't be doing that. Again, this comes down to personal or project preference.

This is also valid JavaScript.

```
const name = 'Ben'
let benCount = 0
if (name === 'Ben') {
    benCount = 1
}

alert('There is ' + benCount + ' Ben')
```

Here you can see an `if` statement. The condition must have parentheses around it. For a single line if statement, curly braces are not required, but sometimes it can be preferable to always use them to be very clear which parts of the code are inside the `if` body.

Notice that we're using triple-equals (===) to compare the variables in the condition. Using === means that the types of the items being compared are taken into comparison. For example, in JavaScript we could compare the string `'1'` and number `1` with ==. This would evaluate to true, as JavaScript coerces them to the same type and then compares. With === type coercion doesn't take place, so `'1' === 1` is false.

However, this is not valid:

```
const pi = 3.14159
pi = 3  // trade accuracy for speed
```

As you would expect, `const` values can't be reassigned. Also notice that single-line comments in JavaScript start with a `//`.

This would also be invalid:

```
let fruitCount = 5
let fruitCount = 6
```

Here the variable `fruitCount` is not being reassigned, it's being redefined, which is not allowed.

Variables defined with `const` are allowed to be mutated, so we could add items to an array or reassign values in an object (`object` is the JavaScript equivalent of Python's `dict`). This is valid:

```
const fruit = ['Apple', 'Banana']
fruit.push('Cherry')  // append 'Cherry' to the end of the
        `fruit` list

const fruitCount = {Apple: 0, 'Banana': 1}
fruitCount.Cherry = 2  // add new item to object
fruitCount['Cherry'] = 2  // is equivalent

const myFruit = 'Cherry'
fruitCount[myFruit] = 2 // is also equivalent
```

We've shown a few new things in the above code snippet:

- Arrays can be defined using square brackets `[]`, equivalent to Python's `list`.
- We're able to make changes to the `fruit` array and it doesn't violate the

`const` declaration because it's mutating the object not reassigning the variable.

- Objects are declared with curly braces `{}`, equivalent to Python's `dict`.
- The object keys don't need to be quoted, `{Apple: 0, 'Banana': 1}` is the equivalent of `{'Apple': 0, Banana: 1}`.
- Object items can be accessed with `.` notation, like Python properties/attributes.
- Object items can also be accessed with square bracket notation, but must be quoted, or accessed with a variable. The equivalent of how you access items in a Python `dict`.

Now we'll look at variable scoping. You'll probably need to experiment yourself to see exactly how scoping will work in your code and functions, especially if you're deeply nested in control flow blocks. But, here's a simple example:

```
const theNumber = 1
let name = 'Ben'

if (theNumber === 1) {
  let name = 'Leo'
  alert(name)
}

alert(name)
```

When this script run it will display two alerts, the first will say `Leo` and the second `Ben`. We are allowed to "redeclare" `name` inside the `if` body, because the variable `name` is "redeclared" only inside that block.

Let's take a look at a couple of minor changes to this script to see the effect. First, we won't redeclare `name` inside the `if` body, just reassign it:

```
const theNumber = 1
let name = 'Ben'

if (theNumber === 1) {
  name = 'Leo'
  alert(name)
}

alert(name)
```

In this case, both alerts will say `Leo`, as the variable `name` is being reassigned in the outer scope, as it doesn't exist in the inner scope (inside the `if` body).

In this next change, we only define `name` in the inner scope:

```
const theNumber = 1

if (theNumber === 1) {
  let name = 'Leo'
  alert(name)
}

alert(name)
```

In this example, the first alert will show `Leo` and the second will show an empty alert because the `name` variable is no longer assigned in the outer scope.

Try It Out

Now you can experiment with some variables and their scopes. Look at the `blog/static/blog/blog.js` file in the IDE and remove the current `alert()` line, and enter this code instead:

```
const theNumber = 1
let yourName = 'Ben'

if (theNumber === 1) {
  let yourName = 'Leo'
  alert(yourName)
}

alert(yourName)
```

If you like, you can change the names that are displayed.

View Blog

Now load the `/post-table/` page in your browser and check the alerts that are displayed. You should see `Leo` then `Ben`. Try experimenting with these changes, and watch the results:

- Change `let yourName = 'Leo'` to just `yourName = 'Leo'`
- Remove the `yourName` declaration from the outer scope.
- Change `theNumber` to something other than `1` so that `if` body is not executed.

You'll need to refresh the page in your browser after each change. If you don't see a change, you might need to hold Shift/Control when refreshing (depending on your browser) to force a refresh of the JavaScript file.

Once you feel like you have a hang of variables and scoping in JavaScript, you can move on to the next section on functions.

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 intro and fundamentals"
```

- Push to GitHub:

```
git push
```