CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 0

Introduction

In this course, we're picking up where CS50 left off and diving into the design and creation of web applications. We'll build our web-design skills by working on a number of projects throughout the course, including an open-ended final project where you'll have the chance to create a website of your own!

In this course, you'll need a text editor where you can write code locally on your computer. Some popular ones include Visual Studios Code (https://code.visualstudio.com/), Sublime Text (https://www.sublimetext.com/), Atom (https://atom.io/), and Vim (https://www.vim.org/), but there are many more to choose from!

Web Programming

Course Topics: We'll go into more detail later, but here's a brief overview of what we'll be working on during this course:

1. **HTML and CSS** (a markup language used to outline a webpage, and a procedure for making our sites more visually appealing)
2. **Git** (used for version control and collaboration)
3. **Python** (a widely-used programming language we'll use to make our sites more dynamic)
4. **Django** (a popular web framework we'll use for the backend of our sites)
5. **SQL, Models, and Migrations** (a language used for storing and retrieving data, and Django-specific methods that make it easier to interact with SQL databases)
6. **JavaScript** (a programming language used to make websites faster and more interactive)
7. **User Interfaces** (methods used to make a website as easy to use as possible)
8. **Testing, CI, CD** (learning about different methods used to make sure updates to web pages proceed smoothly)
9. **Scalability and Security** (making sure our websites can be accessed by many users at once, and that they are safe from malicious intent)

HTML (Hypertext Markup Language)

- HTML is a markup language that defines the structure of a web page. It is interpreted by your web browser (Safari, Google Chrome, Firefox, etc.) in order to display content on your screen.
- Let's get started by writing a simple HTML file!

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Hello!</title>
    </head>
```

```
    <body>
        Hello, world!
    </body>
</html>
```

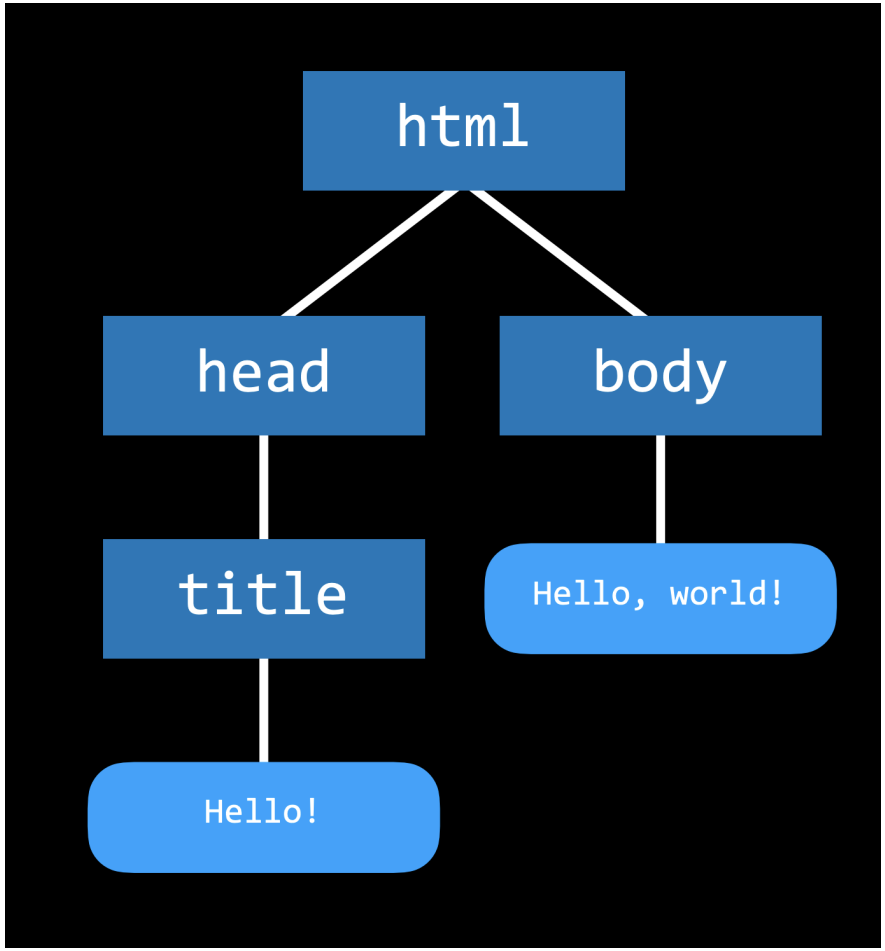
- When we open up this file in our browser, we get:

Hello, world!

- Now, let's take some time to talk about the file we just wrote, which seems to be pretty complicated for such a simple page.
  - In the first line, we are declaring (to the web browser) that we are writing the document in the latest version of HTML: HTML5.
  - After that, the page consists of nested **HTML elements** (such as html and body), each with an **opening and closing tag** marked with either `<element>` for an opening and `</element>` for a closing.
  - Notice how each of the inner elements is indented just a bit further than the last. While this is not necessarily required by the browser, it will be very helpful to keep this up in your own code.
  - HTML elements can include **attributes**, which give the browser extra information about the element. For example, when we include `lang="en"` in our initial tag, we are telling the browser that we are using English as our primary language.
  - Inside the HTML element, we typically want to include both a `head` and a `body` tag. The head element will include information about your page that is not necessarily displayed, and the body element will contain what is actually visible to users who visit the site.

  - Within the head, we have included a `title` for our webpage, which you'll notice is displayed in the tab at the top of our web browser.
  - Finally, we've included the text "Hello, world!" in the body, which is the visible part of our page.

Document Object Model (DOM)



- The DOM is a convenient way of visualizing the way HTML elements relate to each other using a tree-like structure. Above is an example of the DOM layout for the page we just

wrote.

More HTML Elements

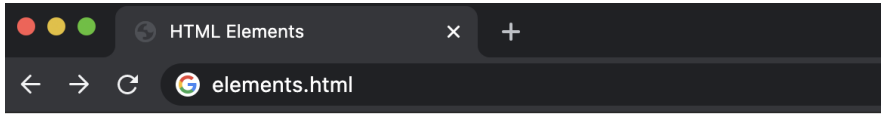
- There are many HTML elements you may want to use to customize your page, including headings, lists, and bolded sections. In this next example, we'll see a few of of these in action.
- One more thing to note: `<!-- -->` gives us a comment in HTML, so we'll use that below to explain some of the elements.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>HTML Elements</title>
    </head>
    <body>
        <!-- We can create headings using h1 through h6 as tags. -->
        <h1>A Large Heading</h1>
        <h2>A Smaller Heading</h2>
        <h6>The Smallest Heading</h6>

        <!-- The strong and i tags give us bold and italics respectively. -->
        A <strong>bold</strong> word and an <i>italicized</i> word!

        <!-- We can link to another page (such as cs50's page) using a. -->
        View the <a href="https://cs50.harvard.edu/">CS50 Website</a>!

        <!-- We used ul for an unordered list and ol for an ordered one. both ordered
        An unordered list:
        <ul>
            <li>foo</li>
            <li>bar</li>
            <li>baz</li>
        </ul>
        An ordered list:
        <ol>
            <li>foo</li>
            <li>bar</li>
            <li>baz</li>
        </ol>

        <!-- Images require a src attribute, which can be either the path to a file on
        An image:
        <img src="../../images/duck.jpeg" alt="Rubber Duck Picture">
```



```
        <!-- We can also see above that for some elements that don't contain other one

        <!-- Here, we use a br tag to add white space to the page. -->
        <br/> <br/>

        <!-- A few different tags are necessary to create a table. -->
        <table>
            <thead>
                <th>Ocean</th>
                <th>Average Depth</th>
                <th>Maximum Depth</th>
            </thead>
            <tbody>
                <tr>
                    <td>Pacific</td>
                    <td>4280 m</td>
                    <td>10911 m</td>
                </tr>
                <tr>
                    <td>Atlantic</td>
                    <td>3646 m</td>
                    <td>8486 m</td>
                </tr>
            </tbody>
        </table>
    </body>
<html>
```

This page, when rendered, looks something like this:



A Large Heading

A Smaller Heading

The Smallest Heading

A **bold** word and an *italicized* word! View the CS50 Website! An unordered list:

- foo
- bar
- baz

An ordered list:

1. foo
2. bar
3. baz

An image: 

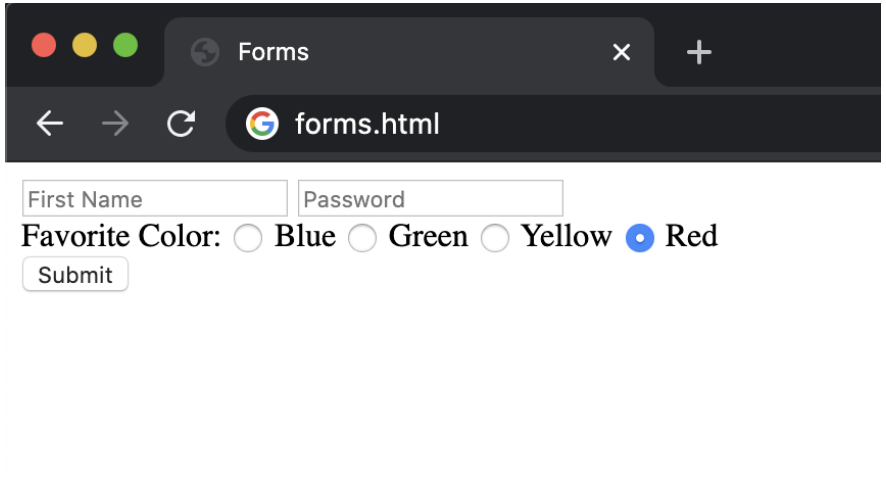
Ocean	Average Depth	Maximum Depth
Pacific	4280 m	10911 m
Atlantic	3646 m	8486 m

- In case you're worried about it, know that you'll never have to memorize these elements. It's very easy to simply search something like "image in HTML" to find the `img` tag. One resource that's especially helpful for learning about these elements is W3 Schools (https://www.w3schools.com/html/html_elements.asp).

Forms

- Another set of elements that is really important when creating a website is how to collect information from users. You can allow users to enter information using an HTML form, which can contain several different types of input. Later in the course, we'll learn about how to handle information once a form has been submitted.
- Just as with other HTML elements, there's no need to memorize these, and W3 Schools is a great resource for learning about them!

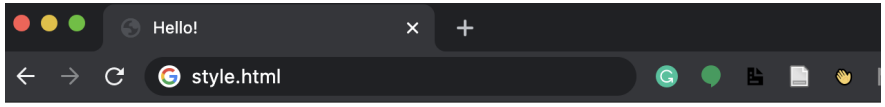
```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Forms</title>
</head>
<body>
    <form>
        <input type="text" placeholder="First Name" name="first">
        <input type="password" placeholder="Password" name="password">
        <div>
            Favorite Color:
            <input name="color" type="radio" value="blue"> Blue
            <input name="color" type="radio" value="green"> Green
            <input name="color" type="radio" value="yellow"> Yellow
            <input name="color" type="radio" value="red"> Red

        </div>
        <input type="submit">
    </form>
</body>
</html>
```



CSS (Cascading Style Sheets)

- CSS is used to customize the appearance of a website.
- While we're just getting, started, we can add a style attribute to any HTML element in order to apply some CSS to it.
- We change style by altering the CSS properties of an element, writing something like `color: blue` or `text-align: center`
- In this example below, we make a slight change to our very first file to give it a colorful heading:

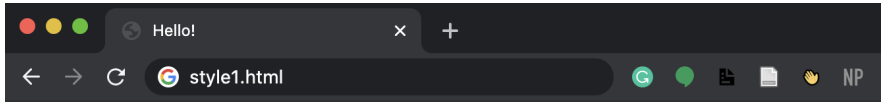
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Hello!</title>
    </head>
    <body>
        <h1 style="color: blue; text-align: center;">A Colorful Heading!</h1>
        Hello, world!
    </body>
<html>
```

Hello, world!

- If we style an outer element, all of the inner elements automatically take on that style. We can see this if we move the styling we just applied from the header tag to the body tag:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Hello!</title>
    </head>
    <body style="color: blue; text-align: center;">
        <h1 >A Colorful Heading!</h1>
        Hello, world!
    </body>
<html>
```



A Colorful Heading!

Hello, world!

- While we can style our web page as we've done above, to achieve better design, we should be able to move our styling away from the individual lines.
  - One way of doing this is to add your styling between `<style>` tags in the `head`. Inside these tags, we write which types of elements we want to be style, and the styling we wish to apply to them. For example:

```
<html lang="en">
<!DOCTYPE html>
<head>
    <title>Hello!</title>
    <style>
        h1 {
            color: blue;
            text-align: center;
        }
    </style>
</head>
<body>
    <h1 >A Colorful Heading!</h1>
    Hello, world!
</body>
</html>
```

  - Another way is to include in a `<link>` element in your `head` with a link to a styles.css file that contains some styling. This means the HTML file would look like:


```
<html lang="en">
<!DOCTYPE html>
<head>
    <title>Hello!</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1 >A Colorful Heading!</h1>
    Hello, world!
</body>
</html>
```

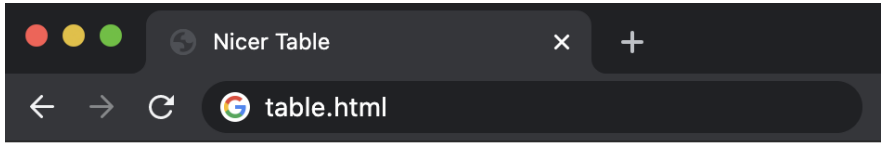
And our file called `styles.css` would look like:

```
h1 {
    color: blue;
    text-align: center;
}
```

- There are far too many CSS properties to go over here, but just like HTML elements, it's typically easy to Google something along the lines of "change font to blue CSS" to get the result. Some of the most common ones though are:
  - `color`: the color of text
  - `text-align`: where elements are placed on the page
  - `background-color`: can be set to any color
  - `width`: in pixels or percent of a page
  - `height`: in pixels or percent of a page
  - `padding`: how much space should be left inside an element
  - `margin`: how much space should be left outside an element
  - `font-family`: type of font for text on page
  - `font-size`: in pixels
  - `border`: size type (solid, dashed, etc) color
- Let's use some of what we just learned to improve upon our oceans table from above. Here's some HTML to start us off:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Nicer Table</title>
    </head>
    <body>
        <table>
            <thead>
                <th>Ocean</th>
                <th>Average Depth</th>
                <th>Maximum Depth</th>
```

```
            </thead>
            <tbody>
                <tr>
                    <td>Pacific</td>
                    <td>4280 m</td>
                    <td>10911 m</td>
                </tr>
                <tr>
                    <td>Atlantic</td>
                    <td>3646 m</td>
                    <td>8486 m</td>
                </tr>
            </tbody>
        </table>
    </body>
<html>
```

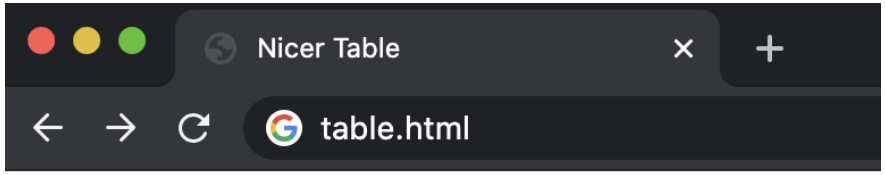


Ocean	Average Depth	Maximum Depth
Pacific	4280 m	10911 m
Atlantic	3646 m	8486 m

- The above looks a lot like what we had before, but now, either by including a `style` tag or a `link` to a stylesheet in the head element, we add the following css:


```
table {
    border: 1px solid black;
    border-collapse: collapse;
}

td {
    border: 1px solid black;
    padding: 2px;
}

th {
    border: 1px solid black;
    padding: 2px;
}
```

Which leaves us with this nicer-looking table:



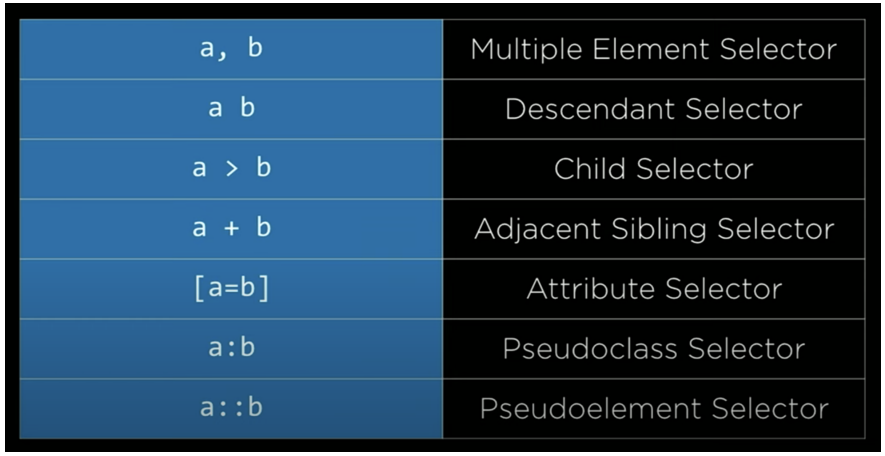
Ocean	Average Depth	Maximum Depth
Pacific	4280 m	10911 m
Atlantic	3646 m	8486 m

- You may already be thinking that there's some needless repetition in our CSS at the moment, as `td` and `th` have the same styling. We can (and should) condense this down to the following code, using a comma to show the styling should apply to more than one element type.

```
table {
    border: 1px solid black;
    border-collapse: collapse;
}
```

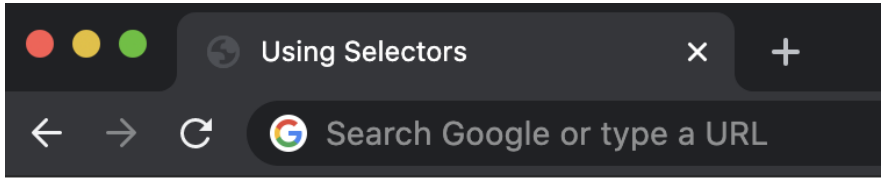
```
td, th {
    border: 1px solid black;
    padding: 2px;
}
```

- This is a good introduction into what are known as CSS selectors (https://www.w3schools.com/cssref/css_selectors.asp). There are many ways to determine which HTML elements you are styling, some of which we'll mention here:
  - **element type**: this is what we've been doing so far: styling all elements of the same type.
  - **id**: Another option is to give our HTML elements an id like so: `<h1 id="first-header">Hello!</h1>` and then applying styling using `#first-header{...}` using the hashtag to show that we're searching by id. Importantly, no two elements can have the same id, and no element can have more than one id.
  - **class**: This is similar to id, but a class can be shared by more than one element, and a single element can have more than one class. We add classes to an HTML element like this: `<h1 class="page-text muted">Hello!</h1>` (note that we just added two classes to the element: `page-text` and `muted`). We then style based on class using a period instead of a hashtag: `.muted {...}`
- Now, we also have to deal with the problem of potentially conflicting CSS. What happens when a header should be red based on its class but blue based on its id? CSS has a specificity order that goes:
  1. In-line styling
  2. id
  3. class
  4. element type
- In addition to the comma for multiple selectors, there are several other ways to specify which elements you would like to style. This table from lecture provides a few, and we'll go through a few examples below:



Descendant Selector: Here, we use the descendant selector to only apply styling to list items found within an unordered list:

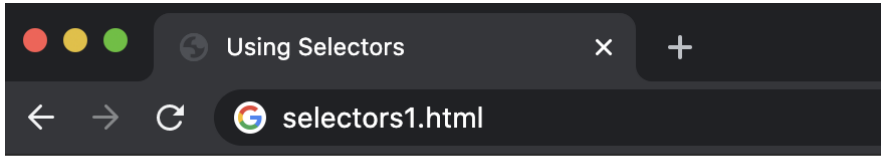
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using Selectors</title>
        <style>
            ul li {
                color: blue;
            }
        </style>
    </head>
    <body>
        <ol>
            <li>foo</li>
            <li> bar
                <ul>
                    <li>hello</li>
                    <li>goodbye</li>
                    <li>hello</li>
                </ul>
            </li>
            <li>baz</li>
        </ol>

    </body>
<html>
```



1. foo
2. bar
    - hello
    - goodbye
    - hello
3. baz

Attributes as Selectors: We can also narrow down our selection based on the attributes we assign to HTML elements using brackets. For example, in the following list of links, we choose to only make the link to Amazon red:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Using Selectors</title>
        <style>
            a[href="https://www.amazon.com/"] {
                color: red;
            }
        </style>
    </head>
    <body>
        <ol>
            <li><a href="https://www.google.com/">Google</a></li>
            <li><a href="https://www.amazon.com/">Amazon</a> </li>
            <li><a href="https://www.facebook.com/">Facebook</a></li>
        </ol>

    </body>
<html>
```

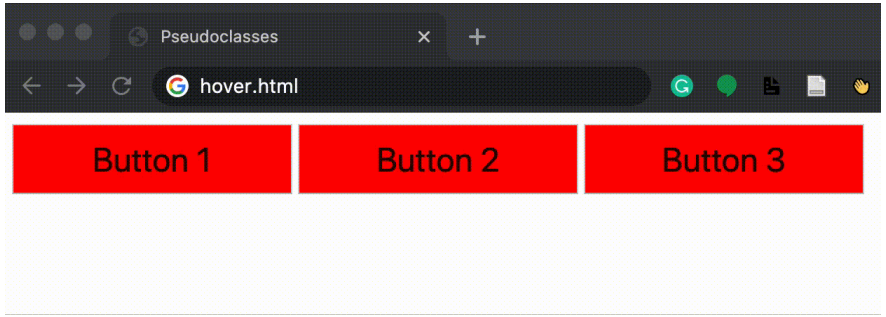



1. Google
2. Amazon
3. Facebook

- Not only can we use CSS to change what an element looks like permanently, but also what it looks like under certain conditions. For example, what if we wanted a button to change color when we hover over it? We can acheive this using a CSS pseudoclass (https://www.w3schools.com/css/css_pseudo_classes.asp), which provides additional styling during special circumstances. We write this by adding a colon after our selector, and then adding the circumstance after that colon.
- In the case of the button, we would add `:hover` to the button selector to specify the design only when hovering:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Pseudoclasses</title>
        <style>
            button {
                background-color: red;
                width: 200px;
                height: 50px;
                font-size: 24px;
            }

            button:hover {
                background-color: green;
            }
```

```
        </style>
    </head>
    <body>
        <button>Button 1</button>
        <button>Button 2</button>
        <button>Button 3</button>

    </body>
</html>
```



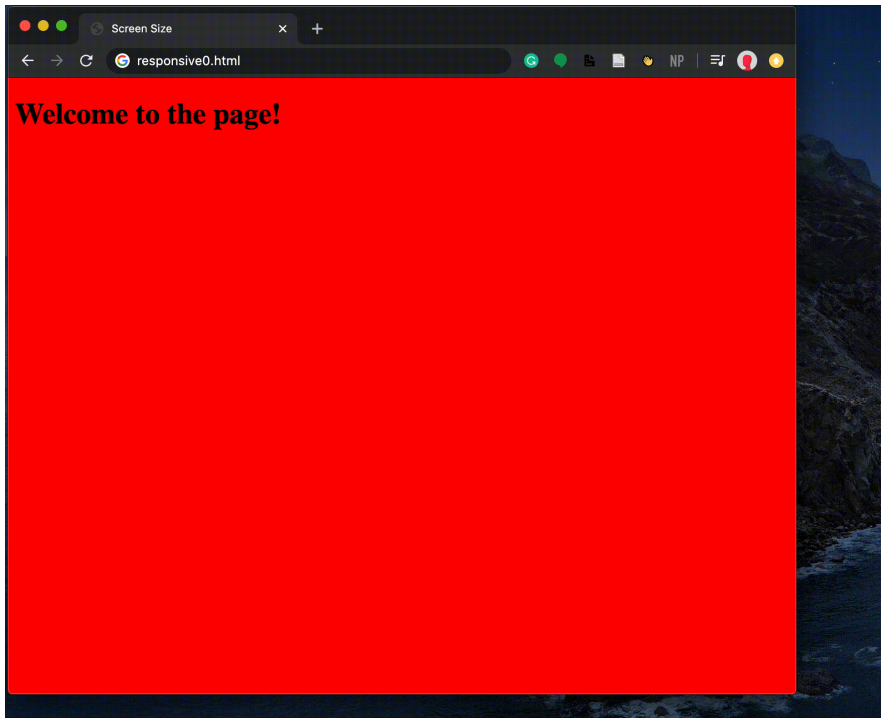
Responsive Design

- Today, many people view websites on devices other than computers, such as smartphones and tablets. It's important to make sure your website is readable to people on all devices.
- One way we can achieve this is through knowledge of the **viewport**. The viewport is the part of the screen that is actually visible to the user at any given time. By default, many webpages assume that the viewport is the same on any device, which is what leads to many sites (especially older ones) being difficult to interact with on mobile devices.
- One simple way to improve the appearance of a site on a mobile device is to add the following line in the head of our HTML files. This line tells the mobile device to use a viewport that is the same width as that of the device you're using rather than a much larger one.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

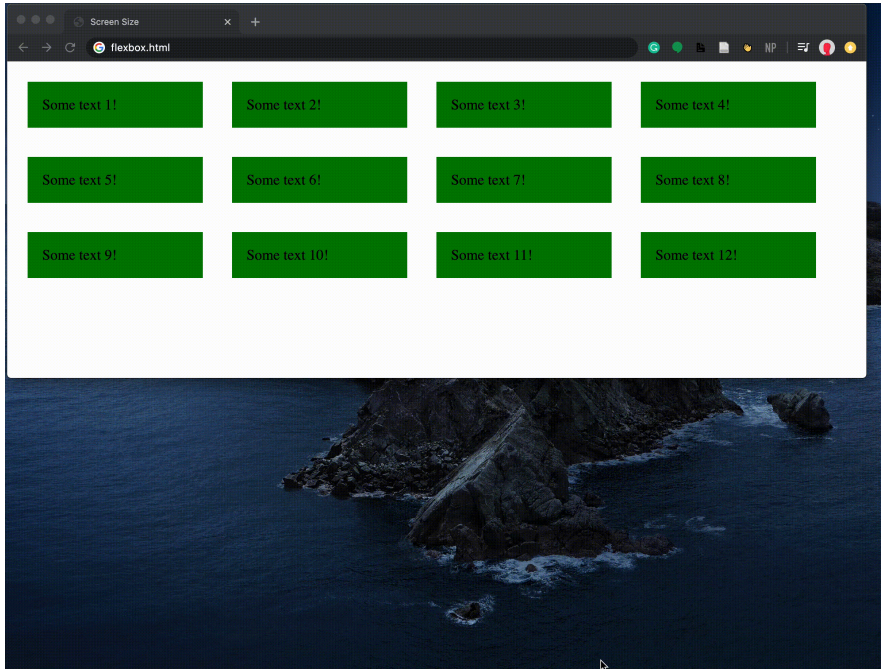
- Another way we can deal with different devices is through media queries (https://www.w3schools.com/cssref/css3_pr_mediaquery.asp). Media queries are ways of changing the style of a page based on how the page is being viewed.
- For an example of a media query, let's try to simply change the color of the screen when it shrinks down to a certain size. We signal a media query by typing `@media` followed by the type of query in parentheses:


```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Screen Size</title>
        <style>
            @media (min-width: 600px) {
                body {
                    background-color: red;
                }
            }

            @media (max-width: 599px) {
                body {
                    background-color: blue;
                }
            }
        </style>
    </head>
    <body>
        <h1>Welcome to the page!</h1>
    </body>
</html>
```



- Another way to deal with differing screen size is using a new CSS attribute known as a flexbox (https://www.w3schools.com/css/css3_flexbox.asp). This allows us to easily have elements wrap around to the next line if they don't fit horizontally. We do this by putting all of our elements in a `div` that we'll call our container. We then add some styling to that div specifying that we want to use a flexbox display for the elements inside of it. We've also added some additional styling to the inner divs to better illustrate the wrapping that's occuring here.

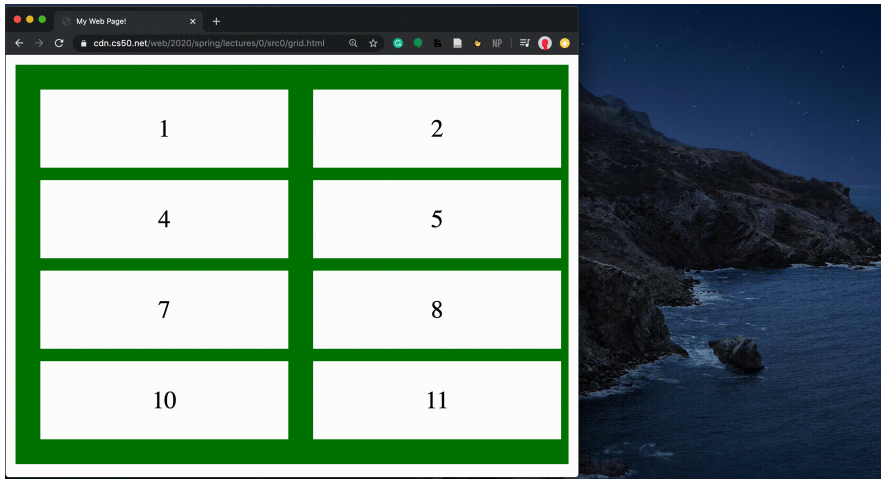
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Screen Size</title>
        <style>
            #container {
                display: flex;
                flex-wrap: wrap;
            }

            #container > div {
                background-color: green;
                font-size: 20px;
                margin: 20px;
                padding: 20px;
                width: 200px;
            }
        </style>
    </head>
    <body>
        <div id="container">
            <div>Some text 1!</div>
            <div>Some text 2!</div>
            <div>Some text 3!</div>
            <div>Some text 4!</div>
            <div>Some text 5!</div>
            <div>Some text 6!</div>
            <div>Some text 7!</div>
            <div>Some text 8!</div>
            <div>Some text 9!</div>
            <div>Some text 10!</div>
            <div>Some text 11!</div>
            <div>Some text 12!</div>
        </div>
    </body>
</html>
```

- Another popular way of styling a page is using an HTML grid (https://www.w3schools.com/css/css_grid.asp). In this grid, we can specify style attributes such as column widths and gaps between columns and rows, as demonstrated below. Note that when we specify column widths, we say the third one is `auto`, meaning it should fill the rest of the page.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>My Web Page!</title>
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <style>
            .grid {
                background-color: green;
                display: grid;
                padding: 20px;
                grid-column-gap: 20px;
                grid-row-gap: 10px;
                grid-template-columns: 200px 200px auto;
            }

            .grid-item {
                background-color: white;
```

```
                font-size: 20px;
                padding: 20px;
                text-align: center;
            }
        </style>
    </head>
    <body>
        <div class="grid">
            <div class="grid-item">1</div>
            <div class="grid-item">2</div>
            <div class="grid-item">3</div>
            <div class="grid-item">4</div>
            <div class="grid-item">5</div>
            <div class="grid-item">6</div>
            <div class="grid-item">7</div>
            <div class="grid-item">8</div>
            <div class="grid-item">9</div>
            <div class="grid-item">10</div>
            <div class="grid-item">11</div>
            <div class="grid-item">12</div>
        </div>
    </body>
</html>
```



Bootstrap

- It turns out that there are many libraries that other people have already written that can make the styling of a webpage even simpler. One popular library that we'll use throughout the course is known as bootstrap (https://getbootstrap.com/).

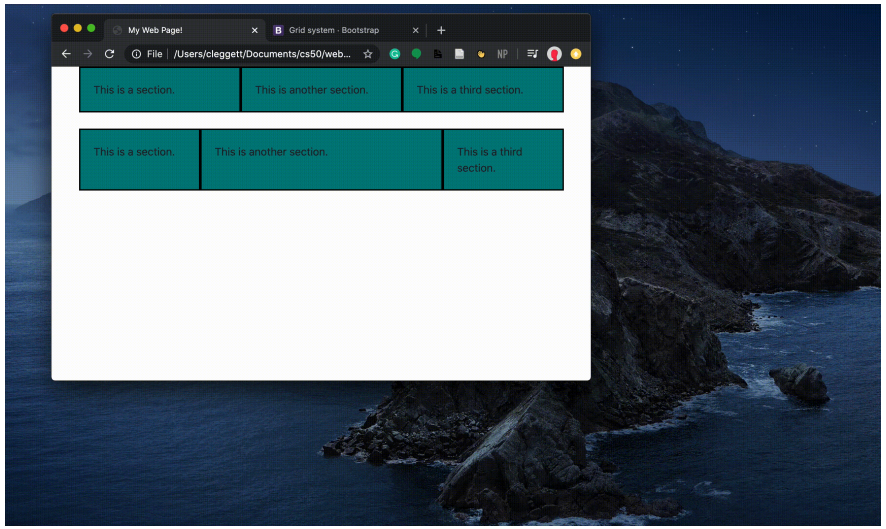
- We can include bootstrap in our code by adding a single line to the head of our HTML file:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bo
```

- Next, we can look at some of bootstrap's features by navigating to the documentation (https://getbootstrap.com/docs/4.5/components/) portion of their website. On this page, you'll find many examples of classes you can add to elements that allow them to be styled with bootstrap.
- One popular bootstrap feature is their grid system (https://getbootstrap.com/docs/4.0/layout/grid/). Bootstrap automatically splits a page into 12 columns, and we can decide how many columns an element takes up by adding the class `col-x` where `x` is a number between 1 and 12. For example, in the following page, we have a row of columns of equal width, and then a row where the center column is larger:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>My Web Page!</title>
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.
        <style>
            .row > div {
                padding: 20px;
                background-color: teal;
                border: 2px solid black;
            }
        </style>
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-4">
                    This is a section.
                </div>
                <div class="col-4">
                    This is another section.
                </div>
                <div class="col-4">
                    This is a third section.
                </div>
            </div>
        </div>
        <br/>
        <div class="container">
            <div class="row">
                <div class="col-3">
                    This is a section.
                </div>
                <div class="col-6">
                    This is another section.
```

```
                </div>
                <div class="col-3">
                    This is a third section.
                </div>
            </div>
        </div>
    </body>
</html>
```

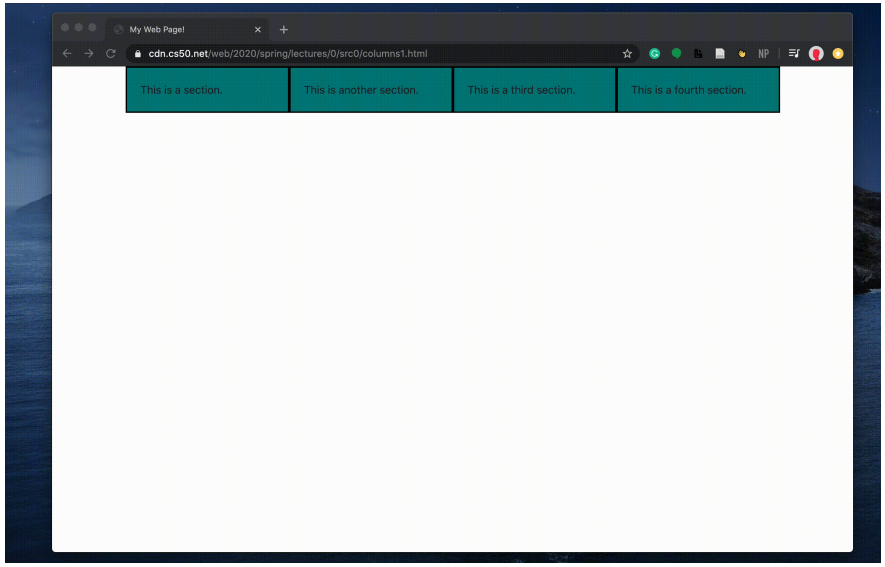


- To improve mobile-responsiveness, bootstrap also allows us to specify column sizes that differ depending on the screen size. In the following example, we use `col-lg-3` to show that an element should take up 3 columns on a large screen, and `col-sm-6` to show an element should take up 6 columns when the screen is small:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>My Web Page!</title>
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.
        <style>
            .row > div {
                padding: 20px;
                background-color: teal;
                border: 2px solid black;
            }
        </style>
    </head>
    <body>
```



```
        <div class="container">
            <div class="row">
                <div class="col-lg-3 col-sm-6">
                    This is a section.
                </div>
                <div class="col-lg-3 col-sm-6">
                    This is another section.
                </div>
                <div class="col-lg-3 col-sm-6">
                    This is a third section.
                </div>
                <div class="col-lg-3 col-sm-6">
                    This is a fourth section.
                </div>
            </div>
        </div>
    </body>
</html>
```



Sass (Syntactically Awesome Style Sheets)

- So far, we've found a few ways to eliminate redundancy in CSS such as moving it to separate files or using bootstrap, but there are still quite a few places where we can still make improvements. For example, what if we want several elements to have different styles, but

for all of them to be the same color? If we decide later we want to change the color, then we would have to change it within several different elements.

- Sass (https://sass-lang.com/) is a language that allows us to write CSS more efficiently in several ways, one of which is by allowing us to have variables, as in the following example.
- When writing in Sass, we create a new file with the extension `filename.scss`. In this file, we can create a new variable by adding a `$` before a name, then a colon, then a value. For example, we would write `$color: red` to set the variable color to the value red. We then access that variable using `$color`. Here's an example of our variables.scss file:

```
$color: red;

ul {
    font-size: 14px;
    color: $color;
}

ol {
    font-size: 18px;
    color: $color;
}
```

- Now, in order to link this styling to our HTML file, we can't just link to the `.scss` file because most web browsers only recognize `.css` files. To deal with this problem, we have to download a program called Sass (https://sass-lang.com/install) onto our computers. Then, in our terminal, we write `sass variables.scss:variables.css` This command will compile a .scss file named `variables.scss` into a .css file named `variables.css`, to which you can add a link in your HTML page.
- To speed up this process, we can use the command `sass --watch variables.scss:variables.css` which automatically changes the `.css` file every time a change is detected in the `.scss` file.
- While using Sass, we can also physically nest our styling rather than use the CSS selectors we talked about earlier. For example, if we want to apply some styling only to paragraphs and unordered lists within a div, we can write the following:

```
div {
    font-size: 18px;

    p {
        color: blue;
    }

    ul {
        color: green;
    }
}
```


Once compiled into CSS, we would get a file that looks like:

```
div {
    font-size: 18px;
}

div p {
    color: blue;
}

div ul {
    color: green;
}
```

- One more feature that Sass gives us is known as inheritance (https://sass-lang.com/guide). This allows us to create a basic set of styling that can be shared by several different elements. We do this by adding a `%` before a name of a class, adding some styling, and then later adding the line `@extend %classname` to the beginning of some styling. For example, the following code applies the styling within the `message` class to each of the different classes below, resulting in a webpage that looks like the one below.

```
%message {
    font-family: sans-serif;
    font-size: 18px;
    font-weight: bold;
    border: 1px solid black;
    padding: 20px;
    margin: 20px;
}

.success {
    @extend %message;
    background-color: green;
}

.warning {
    @extend %message;
    background-color: orange;
}

.error {
    @extend %message;
    background-color: red;
}
```

- That wraps up our content for today!

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

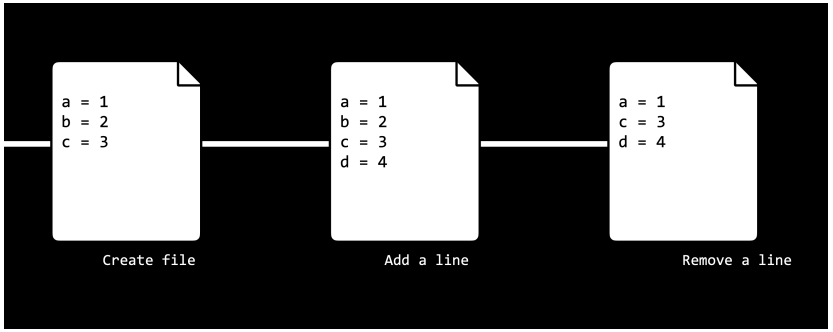
Lecture 1

Introduction

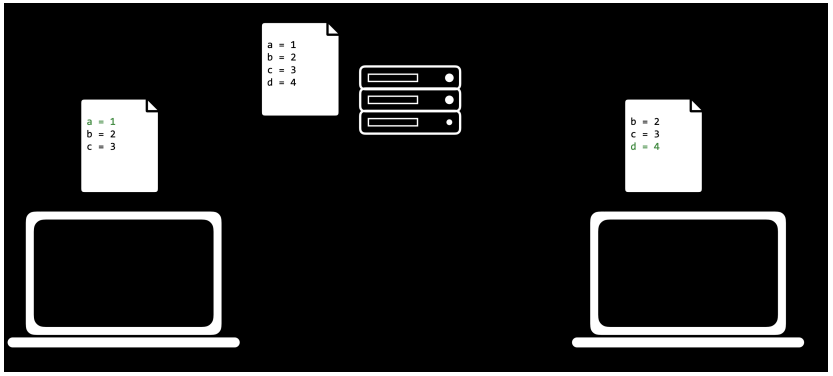
Welcome back to lecture 1! In lecture 0, we introduced HTML, CSS, and Sass as tools we can use to create some basic web pages. Today, we'll be learning about using Git and GitHub to help us in developing web programming applications.

Git

- Git (https://git-scm.com/) is a command line tool that will help us with version control in several different ways:
  - Allowing us to keep track of changes we make to our code by saving snapshots of our code at a given point in time.



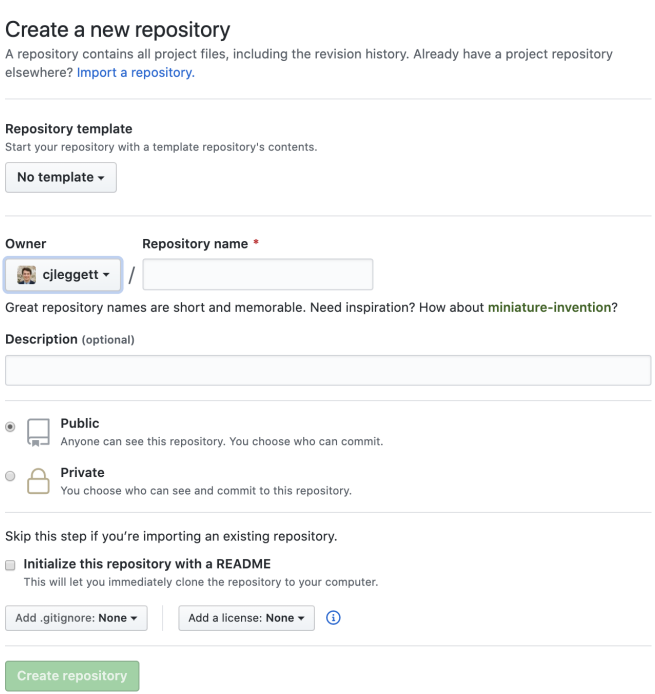
  - Allowing us to easily synchronize code between different people working on the same project by allowing multiple people to pull information from and push information to a repository stored on the web.



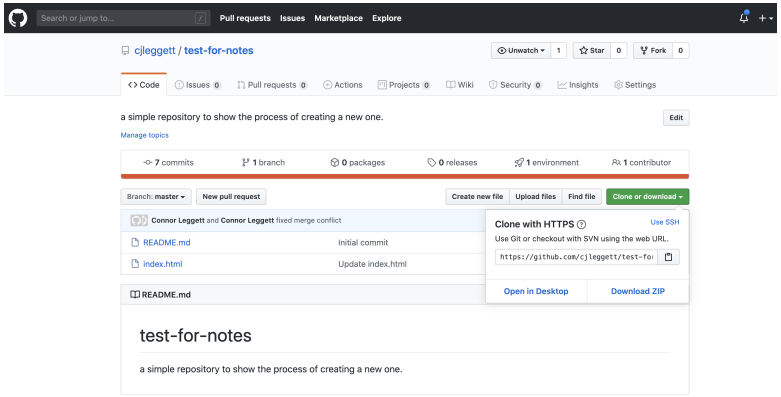
  - Allowing us to make changes to and test out code on a different *branch* without altering our main code base, and then merging the two together.
  - Allowing us to revert back to earlier versions of our code if we realize we've made a mistake.
- In the above explanations, we used the word **repository**, which we haven't explained yet. A Git repository is a file location where we'll store all of the files related to a given project. These can either be remote (stored online) or local (stored on your computer).

GitHub

- GitHub (https://www.github.com) is a website that allows us to store Git repositories remotely on the web.
- Let's get started by creating a new repository online
  1. Make sure that you have a GitHub account set up. If you don't have one yet, you can make one here (https://github.com/join?ref_cta=Sign+up&ref_loc=header+logged+out&ref_page=%2F&source=header-home).
  2. Click the **+** in the top-right corner, and then click "New repository"
  3. Create a repository name that describes your project
  4. (Optional) Provide a description for your repository
  5. Choose whether the repository should be public (visible to anyone on the web) or private (visible just to you and others you specifically grant access)
  6. (Optional) Decide whether you want to add a README, which is a file describing your new repository.



- Once we have a repository, we'll probably want to add some files to it. In order to do this, we'll take our newly created *remote* repository and create a copy, or clone, of it as a *local* repository on our computer.
    1. Make sure you have git installed on your computer by typing `git` into your terminal. If it is not installed, you can download it here (https://git-scm.com/downloads).
    2. Click the green "Clone or Download" button on your repository's page, and copy the url that pops down. If you didn't create a README, this link will appear near the top of the page in the "Quick Setup" section.



3. In your terminal, run `git clone <repository url>`. This will download the repository to your computer. If you didn't create a README, you will get the warning: `You appear to have cloned into an empty repository.` This is normal, and there's no need to worry about it.

```
(base) cleggett@Connors-MacBook-Pro Downloads % git clone https://github.com/cjlegget
t/test-for-notes.git
Cloning into 'test-for-notes'...
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 19 (delta 4), reused 6 (delta 1), pack-reused 0
Unpacking objects: 100% (19/19), done.
```

4. Run `ls`, which is a command that lists all files and folders in your current directory. You should see the name of the repository you've just cloned.
5. Run `cd <repository name>` to change directory into that folder.
6. Run `touch <new file name>` to create a new file in that folder. You can now make edits to that file. Alternatively, you can open the folder in your text editor and manually add new files.
7. Now, to let Git know that it should be keeping track of the new file you've made, Run `git add <new file name>` to track that specific file, or `git add .` to track all files within that directory.

```
(base) cleggett@Connors-MacBook-Pro test-for-notes % touch hello.html
(base) cleggett@Connors-MacBook-Pro test-for-notes % ls
README.md       hello.html      index.html
```

Commits

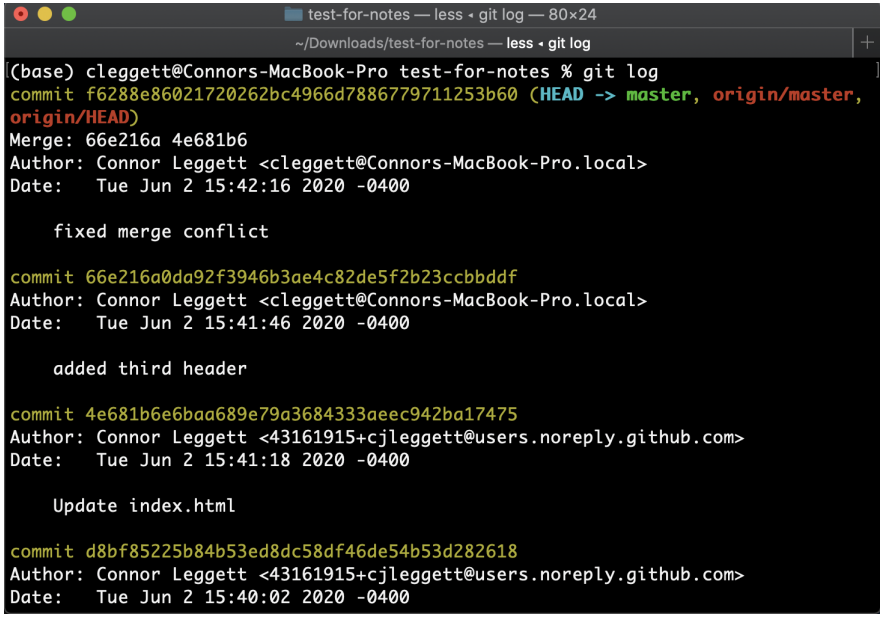
- Now, we'll start to get into what Git can be really useful for. After making some changes to a file, we can *commit* those changes, taking a snapshot of the current state of our code. To do this, we run: `git commit -m "some message"` where the message describes the changes you just made.
- After this change, we can run `git status` to see how our code compares to the code on the remote repository
- When we're ready to publish our local commits to Github, we can run `git push`. Now, when we go to GitHub in our web browser, our changes will be reflected.
- If you've only changed existing files and not created new ones, instead of using `git add .` and then `git commit...`, we can condense this into one command: `git commit -am "some message"`. This command will commit all the changes that you made.
- Sometimes, the remote repository on GitHub will be more up to date than the local version. In this case, you want to first commit any changes, and then run `git pull` to pull any remote changes to your repository.

Merge Conflicts

- One problem that can emerge when working with Git, especially when you're collaborating with other people, is something called a **merge conflict**. A merge conflict occurs when two people attempt to change a file in ways that conflict with each other.
- This will typically occur when you either `git push` or `git pull`. When this happens, Git will automatically change the file into a format that clearly outlines what the conflict is. Here's an example where the same line was added in two different ways:

```
a = 1
<<<<< HEAD
b = 2
=====
b = 3
>>>>> 56782736387980937883
c = 3
d = 4
e = 5
```

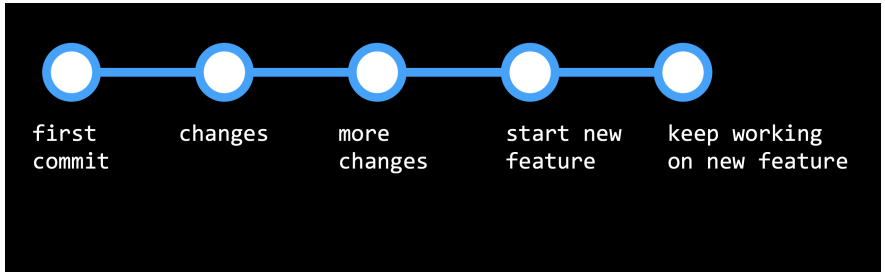
- In the above example, you added the line `b = 2` and another person wrote `b = 3`, and now we must choose one of those to keep. The long number is a *hash* that represents the commit that is conflicting with your edits. Many text editors will also provide highlighting and simple options such as "accept current" or "accept incoming" that save you the time of deleting the added lines above.
- Another potentially useful git command is `git log`, which gives you a history of all of your commits on that repository.



- Potentially even more helpful, if you realize that you've made a mistake, you can revert back to a previous commit using the command `git reset` in one of two ways:
  - `git reset --hard <commit>` reverts your code to exactly how it was after the specified commit. To specify the commit, use the commit hash associated with a commit which can be found using `git log` as shown above.
  - `git reset --hard origin/master` reverts your code to the version currently stored online on Github.

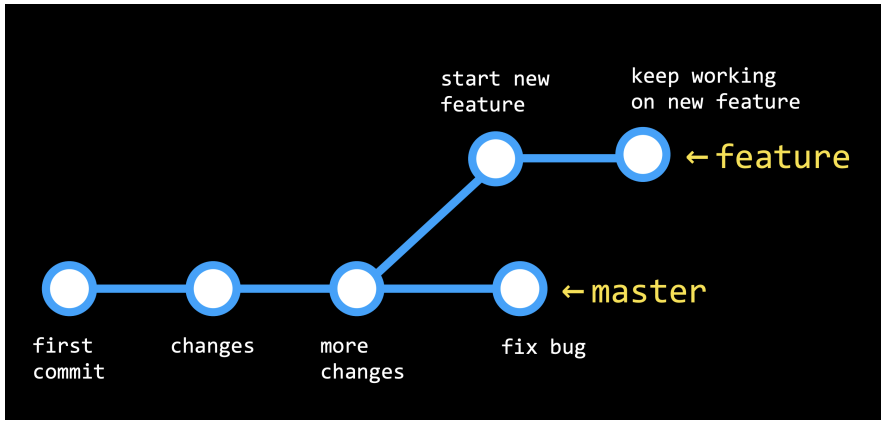
Branching

After you've been working on a project for some time, you may decide that you want to add an additional feature. At the moment, we may just commit changes to this new feature as shown in the graphic below

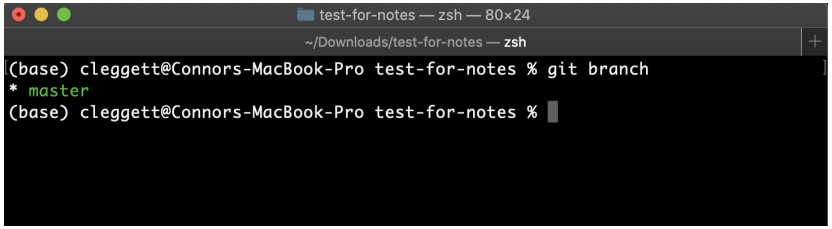


But this could become problematic if we then discover a bug in our original code, and want to revert back without changing the new feature. This is where branching can become really useful.

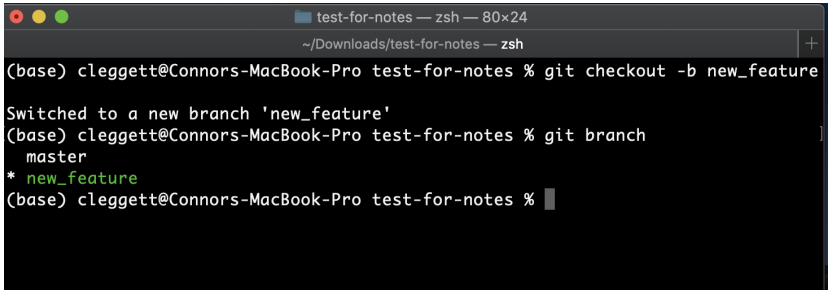
- Branching is a method of moving into a new direction when creating a new feature, and only combining this new feature with the main part of your code, or the main branch, once you're finished. This workflow will look more like the below graphic:



- The branch you are currently looking at is determined by the `HEAD`, which points to one of the two branches. By default, the HEAD is pointed at the master branch, but we can check out other branches as well.
- Now, let's get into how we actually implement branching in our git repositories:
    1. Run `git branch` to see which branch you're currently working on, which will have an asterisk to the left of its name.



1. To make a new branch, we'll run `git checkout -b <new branch name>`



1. Switch between branches using the command `git checkout <branch name>` and commit any changes to each branch.
2. When we're ready to merge our two branches together, we'll check out the branch we wish to keep (almost always the master branch) and then run the command `git merge <other branch name>`. This will be treated similarly to a push or pull, and merge conflicts may appear.

More GitHub Features

There are some useful features specific to GitHub that can help when you're working on a project:

- **Forking**: As a GitHub user, you have the ability to *fork* any repository that you have access to, which creates a copy of the repository that you are the owner of. We do this by clicking the "Fork" button in the top-right.
- **Pull Requests**: Once you've forked a repository and made some changes to your version, you may want to request that those changes be added to the main version of the repository. For example, if you wanted to add a new feature to Bootstrap, you could fork the repository, make some changes, and then submit a pull request. This pull request could then be evaluated and possibly accepted by the people who run the Bootsrap repository. This process of people making a few edits and then requesting that they be merged into a main repository is vital for what is known as *open source software*, or software that created by contributions from a number of developers.

- **GitHub Pages**: GitHub Pages is a simple way to publish a static site to the web. (We'll learn later about static vs dynamic sites.) In order to do this:
    1. Create a new GitHub repository.
    2. Clone the repository and make changes locally, making sure to include an `index.html` file which will be the landing page for your website.
    3. Push those changes to GitHub.
    4. Navigate to the Settings page of your repository, scroll down to GitHub Pages, and choose the master branch in the dropdown menu.
    5. Scroll back down to the GitHub Pages part of the settings page, and after a few minutes, you should see a notification that "Your site is published at: ..." including a URL where you can find your site!

That's all for this lecture! Next time, we'll be looking at Python!

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan) (https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/) (https://twitter.com/davidjmalan)

Lecture 2

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others.
- Today, we'll dive into Python, one of the two main programming languages we'll use throughout this course.

Python

- Python is a very powerful and widely-used language that will allow us to quickly build fairly complicated web applications. In this course, we'll be using Python 3, although Python 2 is still in use in some places. When looking at outside resources, be careful to make sure they're using the same version.
- Let's start where we start with many programming languages: Hello, world. This program, written in Python, would look like this:

```
print("Hello, world!")
```

- To break down what's going on in that line, there is a `print` **function** built in to the python language, that takes an **argument** in parentheses, and displays that argument on the command line.
- To actually write and run this program on your computers, you'll first type this line into your text editor of choice, and then save the file as `something.py`. Next, you'll head over to your terminal, navigate to the directory containing your file, and type `python something.py`. In the case of the above program, the words "Hello, world!" will then be displayed in the terminal.

- Depending on how your computer is set up, you may have to type `python3` instead of `python` before the file name, and you may even have to download Python (https://www.python.org/downloads/) if you haven't already. After installing Python, we recommend that you also download Pip (https://pip.pypa.io/en/stable/installing/), as you'll need that later in the course.
- When you type `python file.py` in your terminal, a program called an **interpreter**, which you downloaded together with Python, reads through your file line by line, and executes each line of the code. This is different than languages like **C** or **Java**, which need to be **compiled** into machine code before they can be run.

Variables

A key part of any programming language is the ability to create and manipulate variables. In order to assign a value to a variable in Python, the syntax looks like this:

```
a = 28
b = 1.5
c = "Hello!"
d = True
e = None
```

Each of these lines is taking the value to the right of the `=`, and storing it in the variable name to the left.

Unlike in some other programming languages, Python variable types are inferred, meaning that while each variable does have a type, we do not have to explicitly state which type it is when we create the variable. Some of the most common variable types are:

- **int**: An integer
- **float**: A decimal number
- **chr**: A single character
- **str**: A string, or sequence of characters
- **bool**: A value that is either `True` or `False`
- **NoneType**: A special value (`None`) indicating the absence of a value.

Now, we'll work on writing a more interesting program that can take input from the user and say hello to that user. To do this, we'll use another built in function called `input` which displays a prompt to the user, and returns whatever the user provides as input. For example, we can write the following in a file called `name.py`:

```
name = input("Name: ")
print("Hello, " + name)
```

When run on the terminal, this is what the program looks like:

```
(base) cleggett@Connors-MacBook-Pro web_notes_files % python hello.py
Name: Connor
Hello, Connor
(base) cleggett@Connors-MacBook-Pro web_notes_files %
```

A couple of things to point out here:

- In the first line, instead of assigning the variable name to an explicit value, we're assigning it to whatever the `input` function returns.
- In the second line, we're using the `+` operator to combine, or **concatenate**, two strings. In python, the `+` operator can be used to add numbers or concatenate strings and lists.

Formatting Strings

- While we can use the `+` operator to combine strings as we did above, in the latest versions of python, there are even easier ways to work with strings, known as formatted strings (https://realpython.com/python-f-strings/), or **f-strings** for short.
- To indicate that we're using formatted strings, we simply add an `f` before the quotation marks. For example, instead of using `"Hello, " + name` as we did above, we could write `f"Hello, {name}"` for the same result. We can even plug a function into this string if we want, and turn our program above into the single line:

```
print(f"Hello, {input("Name: ")}")
```

Conditions

- Just like in other programming languages, Python gives us the ability to run different segments of code based on different conditions (https://realpython.com/python-conditional-statements/). For example, in the program below, we'll change our output depending on the number a user types in:

```
num = input("Number: ")
if num > 0:
    print("Number is positive")
elif num < 0:
    print("Number is negative")
else:
    print("Number is 0")
```


- Getting into how the above program works, conditionals in python contain a keyword (`if`, `elif`, or `else`) and then (except in the `else` case) a boolean expression, or an expression that evaluates to either `True` or `False`. Then, all of the code we want to run if a certain expression is true is **indented** directly below the statement. Indentation is required as part of the Python syntax.
- However, when we run this program, we run into an exception (https://docs.python.org/3/tutorial/errors.html) that looks like this:

```
Number: 5
Traceback (most recent call last):
  File "cond.py", line 2, in <module>
    if num > 0:
TypeError: '>' not supported between instances of 'str' and 'int'
(base) cleggett@Connors-MacBook-Pro web_notes_files %
```

- An exception is what happens when an error occurs while we're running our python code, and over time you'll get better and better at interpreting these errors, which is a very valuable skill to have.
- Let's look a bit more closely at this specific exception: If we look at the bottom, we'll see that we ran into a `TypeError`, which generally means Python expected a certain variable to be of one type, but found it to be of another type. In this case, the exception tells us that we cannot use the `>` symbol to compare a `str` and `int`, and then above we can see that this comparison occurs in line 2.
- In this case, it's obvious that `0` is an integer, so it must be the case that our `num` variable is a string. This is happening because it turns out that the `input` function always returns a string, and we have to specify that it should be turned into (or **cast** into) an integer using the `int` function. This means our first line would now look like:

```
num = int(input("Number: "))
```

- Now, the program will work just as we intended!

Sequences

One of the most powerful parts of the Python language is its ability to work with **sequences** of data in addition to individual variables.

There are several types of sequences that are similar in some ways, but different in others. When explaining those differences, we'll use the terms **mutable/immutable** and **ordered/unordered**. **Mutable** means that once a sequence has been defined, we can change individual elements of that sequence, and **ordered** means that the order of the objects matters.

Strings

Ordered: Yes

Mutable: No

We've already looked at strings a little bit, but instead of just variables, we can think of a string as a sequence of characters. This means we can access individual elements within the string! For example:

```
name = "Harry"
print(name[0])
print(name[1])
```

prints out the first (or index-0) character in the string, which in this case happens to be `H`, and then prints out the second (or index-1) character, which is `a`.

Lists

Ordered: Yes

Mutable: Yes

A Python list (https://www.w3schools.com/python/python_lists.asp) allows you to store any variable types. We create a list using square brackets and commas, as shown below. Similarly to strings, we can print an entire list, or some individual elements. We can also add elements to a list using `append`, and sort a list using `sort`

```
# This is a Python comment
names = ["Harry", "Ron", "Hermione"]
# Print the entire list:
print(names)
# Print the second element of the list:
print(names[1])
# Add a new name to the list:
names.append("Draco")
# Sort the list:
```



```
names.sort()
# Print the new list:
print(names)
```

```
['Harry', 'Ron', 'Hermione']
Ron
['Draco', 'Harry', 'Hermione', 'Ron']
```

Tuples

Ordered: Yes

Mutable: No

Tuples (https://www.w3schools.com/python/python_tuples.asp) are generally used when you need to store just two or three values together, such as the x and y values for a point. In Python code, we use parentheses:

```
point = (12.5, 10.6)
```

Sets

Ordered: No

Mutable: N/A

Sets (https://www.w3schools.com/python/python_sets.asp) are different from lists and tuples in that they are **unordered**. They are also different because while you can have two or more of the same elements within a list/tuple, a set will only store each value once. We can define an empty set using the `set` function. We can then use `add` and `remove` to add and remove elements from that set, and the `len` function to find the set's size. Note that the `len` function works on all sequences in python. Also note that despite adding `4` and `3` to the set twice, each item can only appear once in a set.

```
# Create an empty set:
s = set()

# Add some elements:
s.add(1)
s.add(2)
s.add(3)
s.add(4)
s.add(3)
s.add(1)
```

```
# Remove 2 from the set
s.remove(2)

# Print the set:
print(s)

# Find the size of the set:
print(f"The set has {len(s)} elements.")

""" This is a python multi-line comment:
Output:
{1, 3, 4}
The set has 3 elements.
"""
```

Dictionaries

Ordered: No

Mutable: Yes

Python Dictionaries (https://www.w3schools.com/python/python_dictionaries.asp) or `dict`s, will be especially useful in this course. A dictionary is a set of **key-value pairs**, where each key has a corresponding value, just like in a dictionary, each word (the key) has a corresponding definition (the value). In Python, we use curly brackets to contain a dictionary, and colons to indicate keys and values. For example:

```
# Define a dictionary
houses = {"Harry": "Gryffindor", "Draco": "Slytherin"}
# Print out Harry's house
print(houses["Harry"])
# Adding values to a dictionary:
houses["Hermione"] = "Gryffindor"
# Print out Hermione's House:
print(houses["Hermione"])

""" Output:
Gryffindor
Gryffindor
"""
```

Loops

Loops are an incredibly important part of any programming language, and in Python, they come in two main forms: for loops (https://www.w3schools.com/python/python_for_loops.asp) and while loops (https://www.w3schools.com/python/python_while_loops.asp). For now, we'll focus on For Loops.

- For loops are used to iterate over a sequence of elements, performing some block of code (indented below) for each element in a sequence. For example, the following code will print out the numbers from 0 to 5:

```
for i in [0, 1, 2, 3, 4, 5]:
    print(i)

""" Output:
0
1
2
3
4
5
"""
```

- We can condense this code using the python `range` function, which allows us to easily get a sequence of numbers. The following code gives the exact same result as our code from above:

```
for i in range(6):
    print(i)

""" Output:
0
1
2
3
4
5
"""
```

- This type of loop can work for any sequence! For example, if we wish to print each name in a list, we could write the code below:

```
# Create a list:
names = ["Harry", "Ron", "Hermione"]

# Print each name:
for name in names:
    print(name)

""" Output:
Harry
Ron
Hermione
"""
```

- We can get even more specific if we want, and loop through each character in a single name!

```
name = "Harry"
for char in name:
    print(char)

""" Output:
H
a
r
r
y
"""
```

Functions

We've already seen a few python functions such as `print` and `input`, but now we're going to dive into writing our own functions. To get started, we'll write a function that takes in a number and squares it:

```
def square(x):
    return x * x
```

Notice how we use the `def` keyword to indicate we're defining a function, that we're taking in a single input called `x` and that we use the `return` keyword to indicate what the function's output should be.

We can then "call" this function just as we've called other ones: using parentheses:

```
for i in range(10):
    print(f"The square of {i} is {square(i)}")

""" Output:
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
"""
```

Modules

As our projects get larger and larger, it will become useful to be able to write functions in one file and run them in another. In the case above, we could create create one file called `functions.py` with the code:

```
def square(x):
    return x * x
```

And another file called `square.py` with the code:

```
for i in range(10):
    print(f"The square of {i} is {square(i)}")
```

However, when we try to run `square.py`, we run into the following error:

```
Traceback (most recent call last):
  File "square.py", line 4, in <module>
    print(f"The square of {i} is {square(i)}")
NameError: name 'square' is not defined
(base) cleggett@Connors-MacBook-Pro web_notes_files %
```

We run into this problem because by default, Python files don't know about each other, so we have to explicitly `import` the square function from the `functions` **module** we just wrote. Now, when `square.py` looks like this:

```
from functions import square

for i in range(10):
    print(f"The square of {i} is {square(i)}")
```

Alternatively, we can choose to import the entire `functions` module and then use dot notation to access the `square` function:

```
import functions

for i in range(10):
    print(f"The square of {i} is {functions.square(i)}")
```

There are many built-in Python modules we can import such as `math` or `csv` that give us access to even more functions. Additionally, we can download even more Modules to access even more functionality! We'll spend a lot of time using the `Django` Module, which we'll discuss in the next lecture.

Object-Oriented Programming

Object Oriented Programming (https://en.wikipedia.org/wiki/Object-oriented_programming) is a programming paradigm, or a way of thinking about programming, that is centered around objects that can store information and perform actions.

- **Classes**: We've already seen a few different types of variables in python, but what if we want to create our own type? A Python Class (https://www.w3schools.com/python/python_classes.asp) is essentially a template for a new type of object that can store information and perform actions. Here's a class that defines a two-dimensional point:

```
class Point():
    # A method defining how to create a point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- Note that in the above code, we use the keyword `self` to represent the object we are currently working with. `self` should be the first argument for any method within a Python class.

Now, let's see how we can actually use the class from above to create an object:

```
p = Point(2, 8)
print(p.x)
print(p.y)

""" Output:
2
8
"""
```

Now, let's look at a more interesting example where instead of storing just the coordinates of a Point, we create a class that represents an airline flight:

```
class Flight():
    # Method to create new flight with given capacity
    def __init__(self, capacity):
        self.capacity = capacity
        self.passengers = []

    # Method to add a passenger to the flight:
    def add_passenger(self, name):
        self.passengers.append(name)
```


However, this class is flawed because while we set a capacity, we could still add too many passengers. Let's augment it so that before adding a passenger, we check to see if there is room on the flight:

```
class Flight():
    # Method to create new flight with given capacity
    def __init__(self, capacity):
        self.capacity = capacity
        self.passengers = []

    # Method to add a passenger to the flight:
    def add_passenger(self, name):
        if not self.open_seats():
            return False
        self.passengers.append(name)
        return True

    # Method to return number of open seats
    def open_seats(self):
        return self.capacity - len(self.passengers)
```

Note that above, we use the line `if not self.open_seats()` to determine whether or not there are open seats. This works because in Python, the number 0 can be interpretted as meaning `False`, and we can also use the keyword `not` to signify the opposite of the following statement so `not True` is `False` and `not False` is `True`. Therefore, if `open_seats` returns 0, the entire expression will evaluate to `True`

Now, let's try out the class we've created by instantiating some objects:

```
# Create a new flight with o=up to 3 passengers
flight = Flight(3)

# Create a list of people
people = ["Harry", "Ron", "Hermione", "Ginny"]

# Attempt to add each person in the list to a flight
for person in people:
    if flight.add_passenger(person):
        print(f"Added {person} to flight successfully")
    else:
        print(f"No available seats for {person}")

""" Output:
Added Harry to flight successfully
Added Ron to flight successfully
Added Hermione to flight successfully
No available seats for Ginny
"""
```

Functional Programming

In addition to supporting Object-Oriented Programming, Python also supports the Functional Programming Paradigm (https://en.wikipedia.org/wiki/Functional_programming), in which functions are treated as values just like any other variable.

Decorators

One thing made possible by functional programming is the idea of a decorator, which is a higher-order function that can modify another function. For example, let's write a decorator that announces when a function is about to begin, and when it ends. We can then apply this decorator using an `@` symbol.

```
def announce(f):
    def wrapper():
        print("About to run the function")
        f()
        print("Done with the function")
    return wrapper

@announce
def hello():
    print("Hello, world!")

hello()

""" Output:
About to run the function
Hello, world!
Done with the function
"""
```

Lambda Functions

Lambda functions provide another way to create functions in python. For example, if we want to define the same `square` function we did earlier, we can write:

```
square = lambda x: x * x
```

Where the input is to the left of the `:` and the output is on the right.

This can be useful when we don't want to write a whole separate function for a single, small use. For example, if we want to sort some objects where it's not clear at first how to sort them. Imagine we have a list of people, but with names and houses instead of just names that we wish to sort:


```
people = [
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Cho", "house": "Ravenclaw"},
    {"name": "Draco", "house": "Slytherin"}
]

people.sort()

print(people)
```

This, however, leaves us with the error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'dict' and 'dict'
```

Which occurs because Python doesn't know how to compare two Dictionaries to check if one is less than the other.

We can solve this problem by including a `key` argument to the sort function, which specifies which part of the dictionary we wish to use to sort:

```
people = [
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Cho", "house": "Ravenclaw"},
    {"name": "Draco", "house": "Slytherin"}
]

def f(person):
    return person["name"]

people.sort(key=f)

print(people)

""" Output:
[{'name': 'Cho', 'house': 'Ravenclaw'}, {'name': 'Draco', 'house': 'Slytherin'}, {'nam
"""
```

While this does work, we've had to write an entire function that we're only using once, we can make our code more readable by using a lambda function:

```
people = [
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Cho", "house": "Ravenclaw"},
    {"name": "Draco", "house": "Slytherin"}
]

people.sort(key=lambda person: person["name"])
```

```
print(people)

""" Output:
[{'name': 'Cho', 'house': 'Ravenclaw'}, {'name': 'Draco', 'house': 'Slytherin'}, {'nam
"""
```

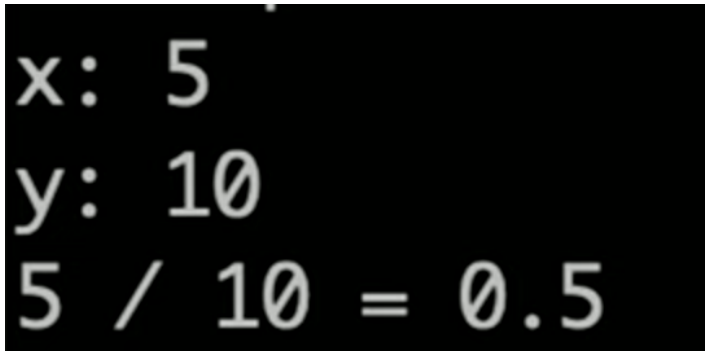
Exceptions

During this lecture, we've run into a few different exceptions, so now we'll look into some new ways of dealing with them.

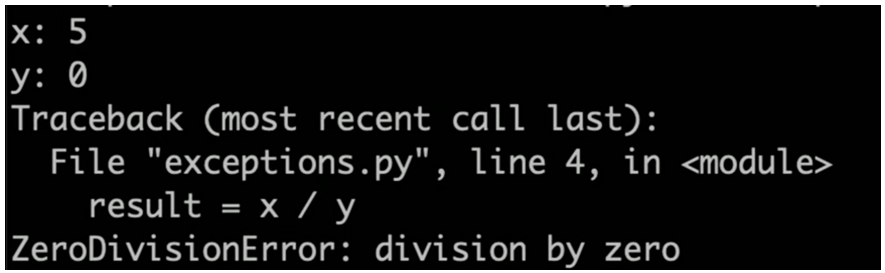
In the following chunk of code, we'll take two integers from the user, and attempt to divide them:

```
x = int(input("x: "))
y = int(input("y: "))

result = x / y

print(f"{x} / {y} = {result}")
```

In many cases, this program works well:



However, we'll run into problems when we attempt to divide by 0:

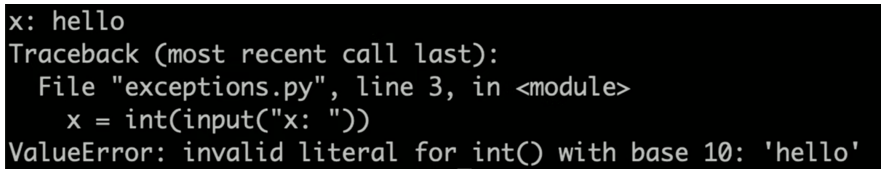


We can deal with this messy error using Exception Handling (https://www.w3schools.com/python/python_try_except.asp). In the following block of code, we will `try` to divide the two numbers, `except` when we get a `ZeroDivisionError`:

```
import sys

x = int(input("x: "))
y = int(input("y: "))

try:
    result = x / y
except ZeroDivisionError:
    print("Error: Cannot divide by 0.")
    # Exit the program
    sys.exit(1)

print(f"{x} / {y} = {result}")
```

In this case, when we try it again:

However, we still run into an error when the user enters non-numbers for x and y:



We can solve this problem in a similar manner!

```
import sys

try:
    x = int(input("x: "))
    y = int(input("y: "))
except ValueError:
    print("Error: Invalid input")
    sys.exit(1)

try:
    result = x / y
except ZeroDivisionError:
    print("Error: Cannot divide by 0.")
    # Exit the program
    sys.exit(1)

print(f"{x} / {y} = {result}")
```

That's all for this lecture! Next time, we'll use Python's `Django` Module to build some applications!

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 3

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language.
- Today, we'll work on using Python's `Django` framework in order to create dynamic applications.

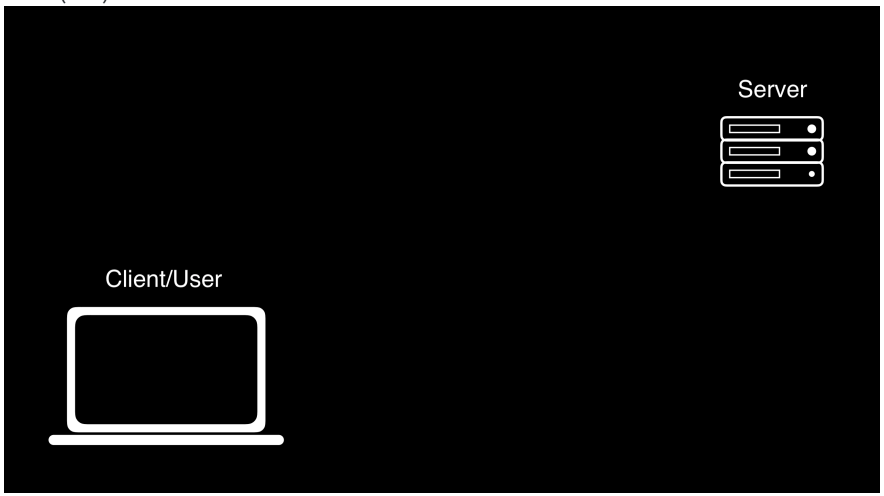
Web Applications

So far, all of the web applications we've written have been **static**. This means that every single time we open that web page, it looks exactly the same. Many websites we visit every day, however, change every time we visit them. If you visit the websites of the New York Times (https://www.nytimes.com/) or Facebook (https://www.facebook.com/), for example, you'll most likely see different things today than you will tomorrow. For large sites like those, it would be unreasonable for employees to have to manually edit a large HTML file every time a change is made, which is where **dynamic** websites can be extremely useful. A dynamic website is one that takes advantage of a programming language (such as Python) to dynamically generate HTML and CSS files. During this lecture, we'll learn how to create our first dynamic applications.

HTTP

HTTP, or HyperText Transfer Protocol, is a widely-accepted protocol for how messages are transfered back and forth across the internet. Typically, information online is passed between a

client (user) and a server.



In this protocol, the client will send a **request** to the server, that might look something like the one below. In the example below, `GET` is simply a type of request, one of three we'll discuss in this course. The `/` typically indicates that we're looking for the website's home page, and the three dots indicate that we could be passing in more information as well.

```
GET / HTTP/1.1
Host: www.example.com
...
```

After receiving a request, a server will then send back an HTTP response, which might look something like the one below. Such a response will include the HTTP version, a status code (200

means OK), a description of the content, and then some additional information.

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

200 is just one of many status codes, some of which you may have seen in the past:

Status Code	Description
200	OK
301	Moved Permanently
403	Forbidden
404	Not Found
500	Internal Server Error

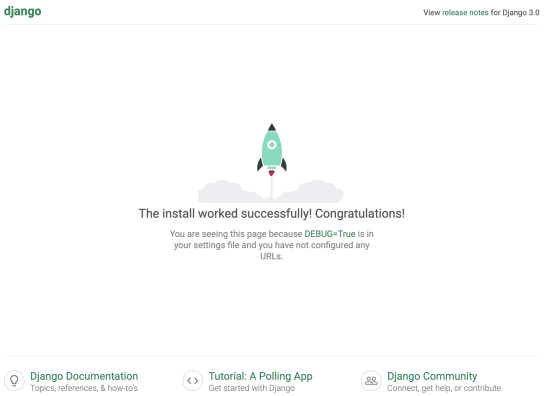
Django

Django (https://www.djangoproject.com/) is a Python-based web framework that will allow us to write Python code that dynamically generates HTML and CSS. The advantage to using a framework like Django is that a lot of code is already written for us that we can take advantage of.

- To get started, we'll have to install Django, which means you'll also have to install pip (https://pip.pypa.io/en/stable/installing/) if you haven't already done so.
- Once you have Pip installed, you can run `pip3 install Django` in your terminal to install Django.

After installing Django, we can go through the steps of creating a new Django project:

1. Run `django-admin startproject PROJECT_NAME` to create a number of starter files for our project.
2. Run `cd PROJECT_NAME` to navigate into your new project's directory.
3. Open the directory in your text editor of choice. You'll notice that some files have been created for you. We won't need to look at most of these for now, but there are three that will be very important from the start:
   - `manage.py` is what we use to execute commands on our terminal. We won't have to edit it, but we'll use it often.
   - `settings.py` contains some important configuration settings for our new project. There are some default settings, but we may wish to change some of them from time to time.
   - `urls.py` contains directions for where users should be routed after navigating to a certain URL.
4. Start the project by running `python manage.py runserver`. This will open a development server, which you can access by visiting the URL provided. This development server is being run locally on your machine, meaning other people cannot access your website. This should bring you to a default landing page:



5. Next, we'll have to create an application. Django projects are split into one or more **applications**. Most of our projects will only require one application, but larger sites can make use of this ability to split a site into multiple apps. To create an application, we run `python manage.py startapp APP_NAME`. This will create some additional directories and files that will be useful shortly, including `views.py`.
6. Now, we have to install our new app. To do this, we go to `settings.py`, scroll down to the list of `INSTALLED_APPS`, and add the name of our new application to this list.

```
# Application definition

INSTALLED_APPS = [
    'myapp',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Routes

Now, in order to get started with our application:

1. Next, we'll navigate to `views.py`. This file will contain a number of different views, and we can think of a view for now as one page the user might like to see. To create our first view, we'll write a function that takes in a `request`. For now, we'll simply return an `HttpResponse` (A very simple response that includes a response code of 200 and a string of text that can be displayed in a web browser) of "Hello, World". In order to do this, we have include `from django.http import HttpResponse`. Our file now looks like:

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def index(request):
    return HttpResponse("Hello, world!")
```

2. Now, we need to somehow associate this view we have just created with a specific URL. To do this, we'll create another file called `urls.py` in the same directory as `views.py`. We already have a `urls.py` file for the whole project, but it is best to have a separate one for each individual app.

3. Inside our new `urls.py`, we'll create a list of url patterns that a user might visit while using our website. In order to do this:
    1. We have to make some imports: `from django.urls import path` will give us the ability to reroute URLSs, and `from . import views` will import any functions we've created in `views.py`.
    2. Create a list called `urlpatterns`
    3. For each desired URL, add an item to the `urlpatterns` list that contains a call to the `path` function with two or three arguments: A string representing the URL path, a function from `views.py` that we wish to call when that URL is visited, and (optionally) a name for that path, in the format `name="something"`. For example, here's what our simple app looks like now:

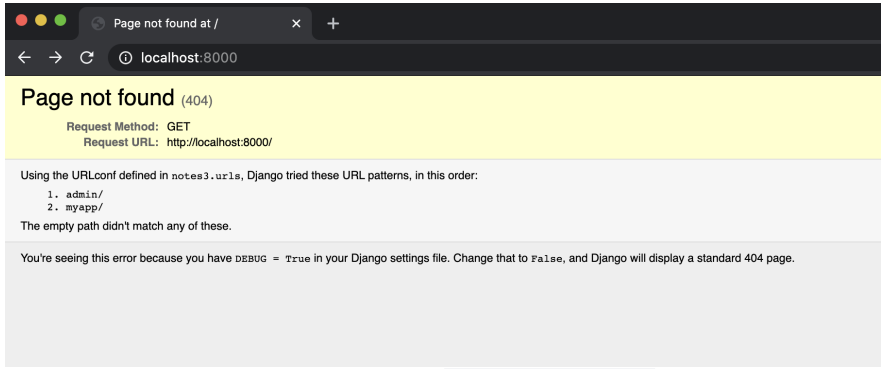
```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index")
]
```

4. Now, we've created a `urls.py` for this specific application, and it's time to edit the `urls.py` created for us for the entire project. When you open this file, you should see that there's already a path called `admin` which we'll go over in later lectures. We want to add another path for our new app, so we'll add an item to the `urlpatterns` list. This follows the same pattern as our earlier paths, except instead of adding a function from `views.py` as our second argument, we want to be able to include *all* of the paths from the `urls.py` file within our application. To do this, we write: `include("APP_NAME.urls")`, where `include` is a function we gain access to by also importing `include` from `django.urls` as shown in the `urls.py` below:

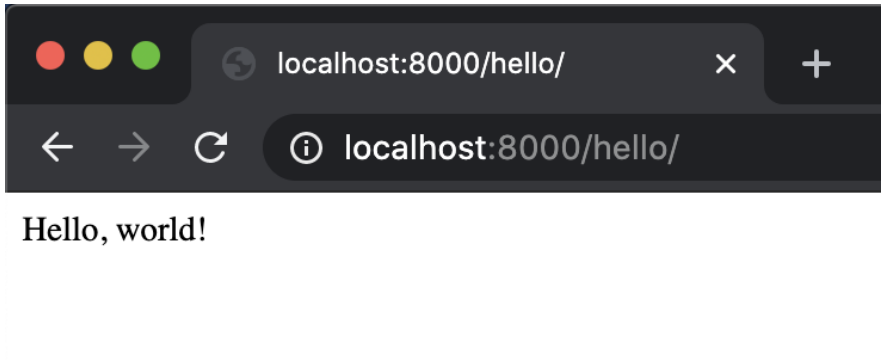
```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', include("hello.urls"))
]
```

5. By doing this, we've specified that when a user visits our site, and then in the search bar adds `/hello` to the URL, they'll be redirected to the paths inside of our new application.

Now, when I start my application using `python manage.py runserver` and visit the url provided, I'm met with this screen:



But this is because we have only defined the URL `localhost:8000/hello`, but we haven't defined the URL `localhost:8000` with nothing added to the end. So, when I add `/hello` to the URL in my search bar:



Now that we've had some success, let's go over what just happened to get us to that point:

1. When we accessed the URL `localhost:8000/hello/`, Django looked at what came after the base URL (`localhost:8000/`) and went to our project's `urls.py` file and searched for a pattern that matched `hello`.
2. It found that extension because we defined it, and saw that when met with that extension, it should `include` our `urls.py` file from within our application.
3. Then, Django ignored the parts of the URL it has already used in rerouting (`localhost:8000/hello/`, or all of it) and looked inside our other `urls.py` file for a pattern that matches the remaining part of the URL.
4. It found that our only path so far (`""`) matched what was left of the URL, and so it directed us to the function from `views.py` associated with that path.
5. Finally, Django ran that function within `views.py`, and returned the result (`HttpResponse("Hello, world!")`) to our web browser.

Now, if we want to, we can change the `index` function within `views.py` to return anything we want it to! We could even keep track of variables and do calculations within the function before eventually returning something.

Now, let's take a look at how we can add more than one view to our application. We can follow many of the same steps within our application to create pages that say hello to Brian and David.

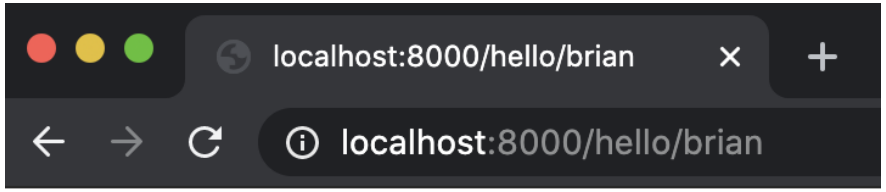
Inside `views.py`:

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def index(request):
    return HttpResponse("Hello, world!")

def brian(request):
    return HttpResponse("Hello, Brian!")

def david(request):
    return HttpResponse("Hello, David!")
```

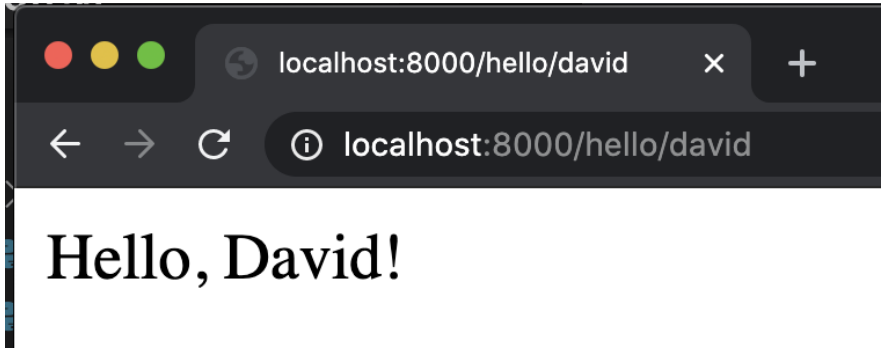
Inside `urls.py` (within our application)

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"),
    path("brian", views.brian, name="brian"),
    path("david", views.david, name="david")
]
```

Now, our site remains unchanged when we visit `localhost:8000/hello`, but we get different pages when we add `brian` or `david` to the URL:



Hello, Brian!



Hello, David!

Many sites are parameterized by items included in the URL. For example, going to www.twitter.com/cs50 (https://twitter.com/cs50) will show you all of CS50's tweets, and going to www.github.com/cs50 (https://github.com/cs50) will bring you to CS50's GitHub page. You can even find your own public GitHub repositories by navigating to `www.github.com/YOUR_USERNAME`!

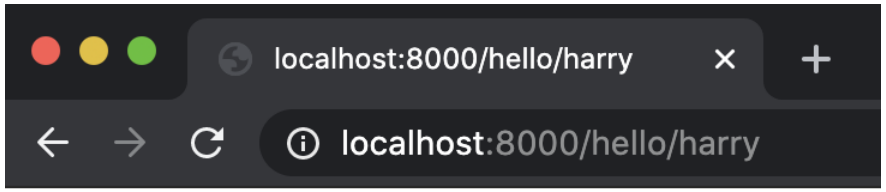
In thinking about how this is implemented, it seems impossible that sites like GitHub and Twitter would have an individual URL path for each of its users, so let's look into how we could make a path that's a bit more flexible. We'll start by adding a more general function, called `greet`, to `views.py`:

```
def greet(request, name):
    return HttpResponse(f"Hello, {name}!")
```

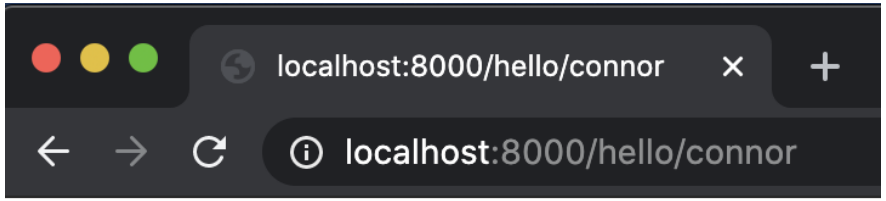
This function takes in not only a request, but also an additional argument of a user's name, and then returns a custom HTTP Response based on that name. Next, we have to create a more flexible path in `urls.py`, which could look something like this:


```
path("<str:name>", views.greet, name="greet")
```

This is some new syntax, but essentially what's going on here is we're no longer looking for a specific word or name in the URL, but any string that a user might enter. Now, we can try the site out with a few other URLs:



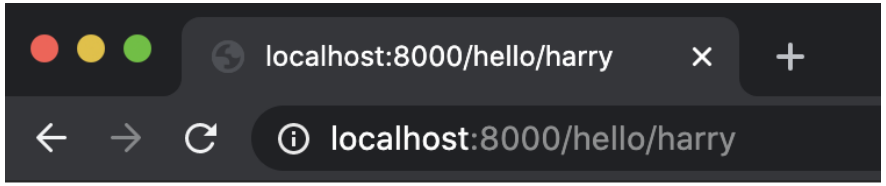
Hello, harry!



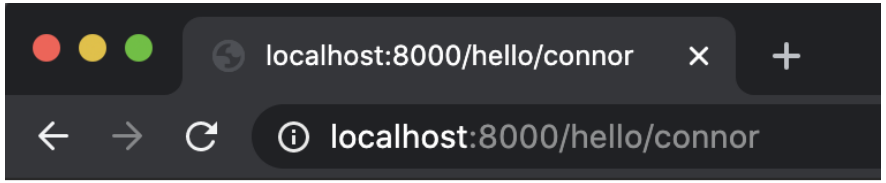
Hello, connor!

I can even make these look a little bit nicer, by augmenting the `greet` function to utilize Python's `capitalize` function that capitalizes a string:

```
def greet(request, name):
    return HttpResponse(f"Hello, {name.capitalize()}!")
```



Hello, Harry!



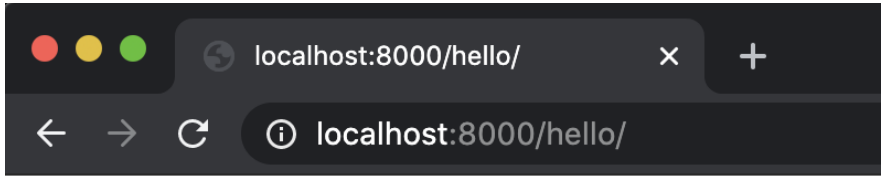
Hello, Connor!

This is a great illustration of how any functionality we have in Python can be used in Django before being returned.

Templates

So far, our HTTP Responses, have been only text, but we can include any HTML elements we want to! For example, I could decide to return a blue header instead of just the text in our `index` function:

```
def index(request):
    return HttpResponse("<h1 style=\"color:blue\">Hello, world!</h1>")
```

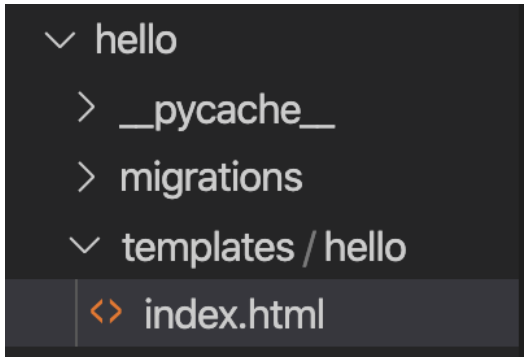
Hello, world!

It would get very tedious to write an entire HTML page within `views.py`. It would also constitute bad design, as we want to keep separate parts of our project in separate files whenever possible.

This is why we'll now introduce Django's templates (https://docs.djangoproject.com/en/4.0/topics/templates/), which will allow us to write HTML and CSS in separate files and render those files using Django. The syntax we'll use for rendering a template looks like this:

```
def index(request):
    return render(request, "hello/index.html")
```

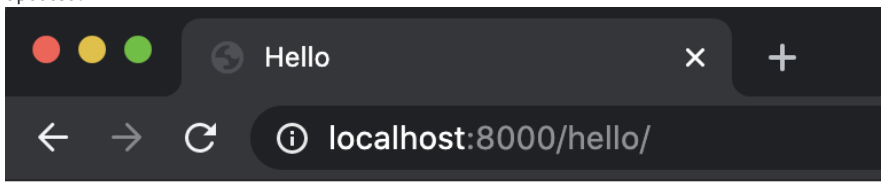
Now, we'll need to create that template. To do this, we'll create a folder called `templates` inside our app, then create a folder called `hello` (or whatever our app's name is) within that, and then add a file called `index.html`.



Next, we'll add whatever we want to that new file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Hello</title>
    </head>
    <body>
        <h1>Hello, World!</h1>
    </body>
</html>
```

Now, when we visit the main page of our application, we can see the header and title have been updated:



Hello, World!

In addition to writing some static HTML pages, we can also use Django's templating language (https://docs.djangoproject.com/en/4.0/ref/templates/language/) to change the content of our HTML files based on the URL visited. Let's try it out by changing our `greet` function from earlier:

```
def greet(request, name):
    return render(request, "hello/greet.html", {
        "name": name.capitalize()
    })
```

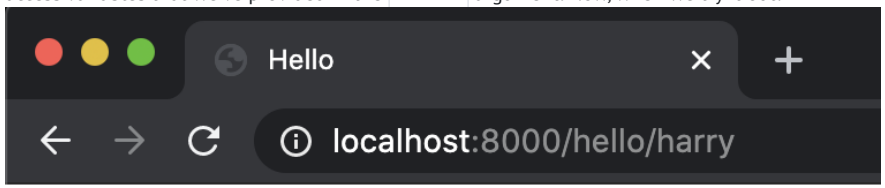
Notice that we passed a third argument into the `render` function here, one that is known as the **context**. In this context, we can provide information that we would like to have available within our HTML files. This context takes the form of a Python dictionary. Now, we can create a `greet.html` file:

```
<!DOCTYPE html>
<html lang="en">
```

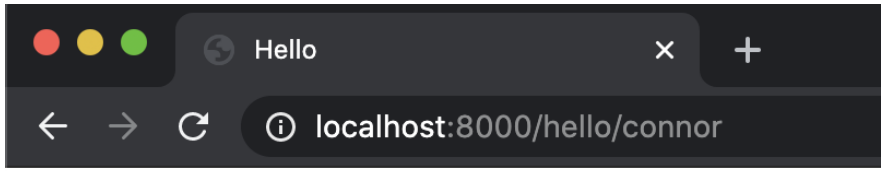


```
    <head>
        <title>Hello</title>
    </head>
    <body>
        <h1>Hello, {{ name }}!</h1>
    </body>
</html>
```

You'll noticed that we used some new syntax: double curly brackets. This syntax allows us to access variables that we've provided in the `context` argument. Now, when we try it out:



Hello, Harry!

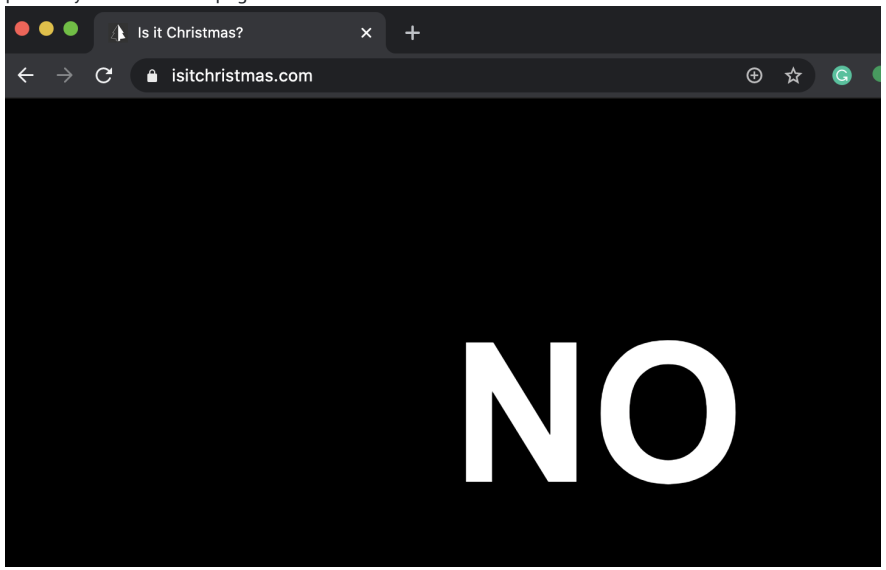


Hello, Connor!

Now, we've seen how we can modify our HTML templates based on the context we provide. However, the Django templating language is even more powerful than that, so let's take a look at a few other ways it can be helpful:

Conditionals:

We may want to change what is displayed on our website depending on some conditions. For example, if you visit the site www.isitchristmas.com (https://www.isitchristmas.com), you'll probably be met with a page that looks like this:



But this website will change on Christmas day, when the website will say **YES**. To make something like this for ourselves, let's try creating a similar application, where we check whether or not it is New Year's Day. Let's create a new app to do so, recalling our process for creating a new app:

1. run `python manage.py startapp newyear` in the terminal.
2. Edit `settings.py`, adding "newyear" as one of our `INSTALLED_APPS`
3. Edit our project's `urls.py` file, and include a path similar to the one we created for the `hello` app:

```
path('newyear/', include("newyear.urls"))
```

1. Create another `urls.py` file within our new app's directory, and update it to include a path similar to the index path in `hello`:

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"),
]
```


1. Create an index function in `views.py`.

Now that we're set up with our new app, let's figure out how to check whether or not it's New Year's Day. To do this, we can import Python's datetime (https://docs.python.org/3/library/datetime.html) module. To get a sense for how this module works, we can look at the documentation (https://docs.python.org/3/library/datetime.html), and then test it outside of Django using the Python interpreter.

- The **Python interpreter** is a tool we can use to test out small chunks of Python code. To use this, run `python` in your terminal, and then you'll be able to type and run Python code within your terminal. When you're done using the interpreter, run `exit()` to leave.

```
(base) cleggett@Connors-MacBook-Pro notes3 % python
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import datetime
>>> now = datetime.datetime.now()
>>> now.day
4
>>> now.month
6
>>> now.year
2020
>>> exit()
(base) cleggett@Connors-MacBook-Pro notes3 %
```

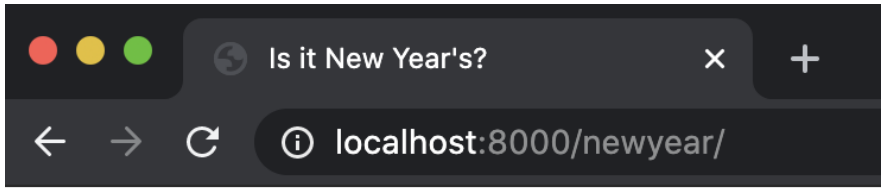
- We can use this knowledge to construct a boolean expression that will evaluate to True if and only if today is New Year's Day: `now.day == 1 and now.month == 1`
- Now that we have an expression we can use to evaluate whether or not it's New Year's Day, we can update our index function in `views.py`:

```
def index(request):
    now = datetime.datetime.now()
    return render(request, "newyear/index.html", {
        "newyear": now.month == 1 and now.day == 1
    })
```

Now, let's create our `index.html` template. We'll have to again create a new folder called `templates`, a folder within that called `newyear`, and a file within that called `index.html`. Inside that file, we'll write something like this:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Is it New Year's?</title>
    </head>
    <body>
        {% if newyear %}
            <h1>YES</h1>
        {% else %}
            <h1>NO</h1>
        {% endif %}
    </body>
</html>
```

In the code above, notice that when we wish to include logic in our HTML files, we use `{%` and `%}` as opening and closing tags around logical statements. Also note that Django's formatting language requires you to include an ending tag indicating that we are done with our `if-else` block. Now, we can open up to our page to see:



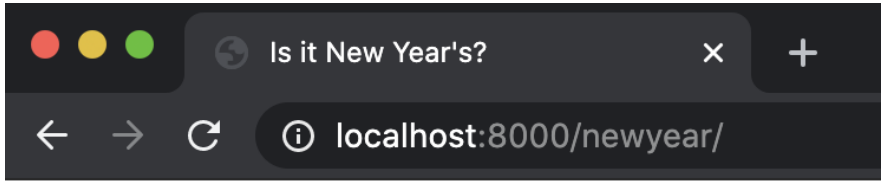
NO

Now, to get a better idea of what's going on behind the scenes, let's inspect the element of this page:


```
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <title>Is it New Year's?</title>
5      </head>
6      <body>
7
8              <h1>NO</h1>
9
10     </body>
11 </html>
```

Notice that the HTML that is actually being sent to your web browser includes only the NO header, meaning that Django is using the HTML template we wrote to create a new HTML file, and then sending it to our web browser. If we cheat a little bit and make sure that our condition is always true, we see that the opposite case is filled:

```
def index(request):
    now = datetime.datetime.now()
    return render(request, "newyear/index.html", {
        "newyear": True
    })
```



YES

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Is it New Year's?</title>
    </head>
    <body>

            <h1>YES</h1>

    </body>
</html>
```

Styling

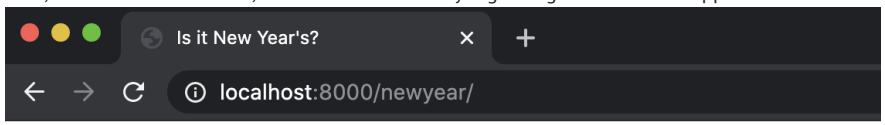
If we want to add a CSS file, which is a *static* file because it doesn't change, we'll first create a folder called `static`, then create a `newyear` folder within that, and then a `styles.css` file within that. In this file, we can add any styling we wish just as we did in the first lecture:

```
h1 {
    font-family: sans-serif;
    font-size: 90px;
    text-align: center;
}
```


Now, to include this styling in our HTML file, we add the line `{% load static %}` to the top of our HTML template, which signals to Django that we wish to have access to the files in our `static` folder. Then, rather than hard-coding the link to a stylesheet as we did before, we'll use some Django-specific syntax:

```
<link rel="stylesheet" href="{% static 'newyear/styles.css' %}">
```

Now, if we restart the server, we can see that the styling changes were in fact applied:



NO

Tasks

Now, let's take what we've learned so far and apply it to a mini-project: creating a TODO list. Let's start by, once again, creating a new app:

1. run `python manage.py startapp tasks` in the terminal.
2. Edit `settings.py`, adding "tasks" as one of our `INSTALLED_APPS`

3. Edit our project's `urls.py` file, and include a path similar to the one we created for the `hello` app:

```
path('tasks/', include("tasks.urls"))
```

4. Create another `urls.py` file within our new app's directory, and update it to include a path similar to the index path in `hello`:

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"),
]
```

5. Create an index function in `views.py`.

Now, let's begin by attempting to simply create a list of tasks and then display them to a page. Let's create a Python list at the top of `views.py` where we'll store our tasks. Then, we can update our `index` function to render a template, and provide our newly-created list as context.

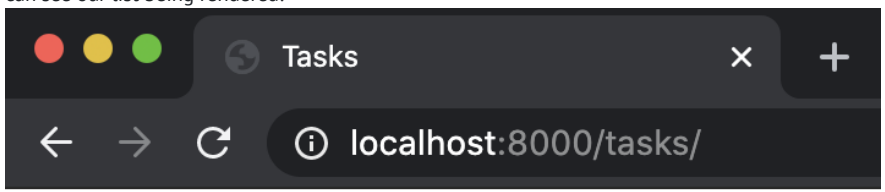
```
from django.shortcuts import render

tasks = ["foo", "bar", "baz"]

# Create your views here.
def index(request):
    return render(request, "tasks/index.html", {
        "tasks": tasks
    })
```

Now, let's work on creating our template HTML file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Tasks</title>
    </head>
    <body>
        <ul>
            {% for task in tasks %}
                <li>{{ task }}</li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Notice here that we are able to loop over our tasks using syntax similar to our conditionals from earlier, and also similar to a Python loop from Lecture 2. When we go to the tasks page now, we

can see our list being rendered:



- foo
- bar
- baz

Forms

Now that we can see all of our current tasks as a list, we may want to be able to add some new tasks. To do this we'll start taking a look at using forms to update a web page. Let's begin by adding another function to `views.py` that will render a page with a form for adding a new task:

```
# Add a new task:
def add(request):
    return render(request, "tasks/add.html")
```

Next, make sure to add another path to `urls.py`:

```
path("add", views.add, name="add")
```

Now, we'll create our `add.html` file, which is fairly similar to `index.html`, except that in the body we'll include a form rather than a list:

```
<!DOCTYPE html>
<html lang="en">
```

```
    <head>
        <title>Tasks</title>
    </head>
    <body>
        <h1>Add Task:</h1>
        <form action="">
            <input type="text" name="task">
            <input type="submit">
        </form>
    </body>
</html>
```

However, what we've just done isn't necessarily the best design, as we've just repeated the bulk of that HTML in two different files. Django's templating language gives us a way to eliminate this poor design: template inheritance (https://tutorial.djangogirls.org/en/template_extending/). This allows us to create a `layout.html` file that will contain the general structure of our page:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Tasks</title>
    </head>
    <body>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

Notice that we've again used `{%...%}` to denote some sort of non-HTML logic, and in this case, we're telling Django to fill this "block" with some text from another file. Now, we can alter our other two HTML files to look like:

`index.html`:

```
{% extends "tasks/layout.html" %}

{% block body %}
    <h1>Tasks:</h1>
    <ul>
        {% for task in tasks %}
            <li>{{ task }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

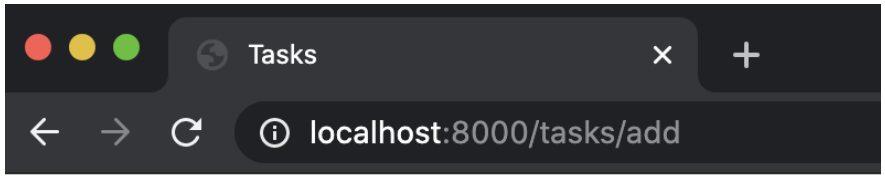
`add.html`:

```
{% extends "tasks/layout.html" %}

{% block body %}
```



```
    <h1>Add Task:</h1>
    <form action="">
        <input type="text" name="task">
        <input type="submit">
    </form>
{% endblock %}
```

Notice how we can now get rid of much of the repeated code by *extending* our layout file. Now, our index page remains the same, and we now have an add page as well:



Next, it's not ideal to have to type "/add" in the URL any time we want to add a new task, so we'll probably want to add some links between pages. Instead of hard-coding links though, we can now use the `name` variable we assigned to each path in `urls.py`, and create a link that looks like this:

```
<a href="{% url 'add' %}">Add a New Task</a>
```

where 'add' is the name of that path. We can do a similar thing in our `add.html`:

```
<a href="{% url 'index' %}">View Tasks</a>
```

This could potentially create a problem though, as we have a few routes named `index` throughout our different apps. We can solve this by going into each of our app's `urls.py` file, and adding an `app_name` variable, so that the files now look something like this:

```
from django.urls import path
from . import views

app_name = "tasks"
urlpatterns = [
    path("", views.index, name="index"),
    path("add", views.add, name="add")
]
```

We can then change our links from simply `index` and `add` to `tasks:index` and `tasks:add`

```
<a href="{% url 'tasks:index' %}">View Tasks</a>

<a href="{% url 'tasks:add' %}">Add a New Task</a>
```

Now, let's work on making sure the form actually does something when the user submits it. We can do this by adding an `action` to the form we have created in `add.html`:

```
<form action="{% url 'tasks:add' %}" method="post">
```

This means that once the form is submitted, we will be routed back to the `add` URL. Here we've specified that we'll be using a *post* method rather than a *get* method, which is typically what we'll use any time a form could alter the state of that web page.

We need to add a bit more to this form now, because Django requires a token to prevent Cross-Site Request Forgery (CSRF) Attack (https://portswigger.net/web-security/csrf). This is an attack where a malicious user attempts to send a request to your server from somewhere other than your site. This could be a really big problem for some websites. Say, for example, that a banking website has a form for one user to transfer money to another one. It would be catastrophic if someone could submit a transfer from outside of the bank's website!

To solve this problem, when Django sends a response rendering a template, it also provides a **CSRF token** that is unique with each new session on the site. Then, when a request is submitted, Django checks to make sure the CSRF token associated with the request matches one that it has recently provided. Therefore, if a malicious user on another site attempted to submit a request, they would be blocked due to an invalid CSRF token. This CSRF validation is built into the Django Middleware (https://docs.djangoproject.com/en/4.0/topics/http/middleware/) framework, which can intervene in the request-response processing of a Django app. We won't go into any more detail about Middleware in this course, but do look at the documentation (https://docs.djangoproject.com/en/4.0/topics/http/middleware/) if interested!

To incorporate this technology into our code, we must add a line to our form in `add.html`.

```
<form action="{% url 'tasks:add' %}" method="post">
    {% csrf_token %}
```



```
    <input type="text" name="task">
    <input type="submit">
</form>
```

This line adds a hidden input field with the CSRF token provided by Django, such that when we reload the page, it looks as though nothing has changed. However, if we inspect element, we'll notice that a new input field has been added:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Tasks</title>
    </head>
    <body>

    <h1>Add Task:</h1>
    <form action="/tasks/add" method="post">
        <input type="hidden" name="csrfmiddlewaretoken"
value="srphkRos17lLoMvxybqcsTQRoev4uCOySqGdwUZU2SAXhfU1SuMXYNWjTy9Jl3Z3">
        <input type="text", name="task">
        <input type="submit">
    </form>
    <a href="/tasks/">View Tasks</a>

    </body>
</html>
```

Django Forms

While we can create forms by writing raw HTML as we've just done, Django provides an even easier way to collect information from a user: Django Forms (https://docs.djangoproject.com/en/4.0/ref/forms/api/). In order to use this method, we'll add the following to the top of `views.py` to import the `forms` module:

```
from django import forms
```

Now, we can create a new form within `views.py` by creating a Python class called `NewTaskForm`:

```
class NewTaskForm(forms.Form):
    task = forms.CharField(label="New Task")
```

Now, let's go through what's going on in that class:

- Inside the parentheses after `NewTaskForm`, we see that we have `forms.Form`. This is because our new form inherits (https://www.w3schools.com/python/python_inheritance.asp) from a class called `Form` that is included in the `forms` module. We've already seen how inheritance can be used in Django's templating language and for styling using Sass. This is another

example of how inheritance is used to take a more general description (the `forms.Form` class) and narrow it down to what we want (our new Form). Inheritance is a key part of Object Oriented Programming that we won't discuss in detail during this course, but there are many online resources (https://www.w3schools.com/python/python_inheritance.asp) available to learn about the topic!

- Inside this class, we can specify what information we would like to collect from the user, in this case the name of a task.
- We specify that this should be a textual input by writing `forms.CharField`, but there are many other input fields (https://docs.djangoproject.com/en/4.0/ref/forms/fields/#built-in-field-classes) included in Django's form module that we can choose from.
- Within this `CharField`, we specify a `label`, which will appear to the user when they load the page. A `label` is just one of many arguments (https://docs.djangoproject.com/en/4.0/ref/forms/fields/#core-field-arguments) we can pass into a form field.

Now that we've created a `NewTaskForm` class, we can include it in the context while rendering the `add` page:

```
# Add a new task:
def add(request):
    return render(request, "tasks/add.html", {
        "form": NewTaskForm()
    })
```

Now, within `add.html`, we can replace our input field with the form we just created:

```
{% extends "tasks/layout.html" %}

{% block body %}
    <h1>Add Task:</h1>
    <form action="{% url 'tasks:add' %}" method="post">
        {% csrf_token %}
        {{ form }}
        <input type="submit">
    </form>
    <a href="{% url 'tasks:index' %}">View Tasks</a>
{% endblock %}
```

There are several advantages to using the `forms` module rather than manually writing an HTML form:

- If we want to add new fields to the form, we can simply add them in `views.py` without typing additional HTML.
- Django automatically performs client-side validation (https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation), or validation local to the user's machine. meaning it

will not allow a user to submit their form if it is incomplete.

- Django provides simple server-side validation (https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation), or validation that occurs once form data has reached the server.
- In the next lecture, we'll begin using **models** to store information, and Django makes it very simple to create a form based on a model.

Now that we have a form set up, let's work on what happens when a user clicks the submit button. When a user navigates to the add page by clicking a link or typing in the URL, they submit a `GET` request to the server, which we've already handled in our `add` function. When a user submits a form though, they send a `POST` request to the server, which at the moment is not handled in the `add` function. We can handle a `POST` method by adding a condition based on the `request` argument our function takes in. The comments in the code below explain the purpose of each line:

```
# Add a new task:
def add(request):

    # Check if method is POST
    if request.method == "POST":

        # Take in the data the user submitted and save it as form
        form = NewTaskForm(request.POST)

        # Check if form data is valid (server-side)
        if form.is_valid():

            # Isolate the task from the 'cleaned' version of form data
            task = form.cleaned_data["task"]

            # Add the new task to our list of tasks
            tasks.append(task)

            # Redirect user to list of tasks
            return HttpResponseRedirect(reverse("tasks:index"))

        else:

            # If the form is invalid, re-render the page with existing information.
            return render(request, "tasks/add.html", {
                "form": form
            })

    return render(request, "tasks/add.html", {
        "form": NewTaskForm()
    })
```

A quick note: in order to redirect the user after a successful submission, we need a few more imports:

```
from django.urls import reverse
from django.http import HttpResponseRedirect
```

Sessions

At this point, we've successfully built an application that allows us to add tasks to a growing list. However, it may be a problem that we store these tasks as a global variable, as it means that all of the users who visit the page see the exact same list. In order to solve this problem we're going to employ a tool known as sessions. (https://docs.djangoproject.com/en/4.0/topics/http/sessions/)

Sessions are a way to store unique data on the server side for each new visit to a website.

To use sessions in our application, we'll first delete our global `tasks` variable, then alter our `index` function, and finally make sure that anywhere else we had used the variable `tasks`, we replace it with `request.session["tasks"]`

```
def index(request):

    # Check if there already exists a "tasks" key in our session

    if "tasks" not in request.session:

        # If not, create a new list
        request.session["tasks"] = []

    return render(request, "tasks/index.html", {
        "tasks": request.session["tasks"]
    })

# Add a new task:
def add(request):
    if request.method == "POST":

        # Take in the data the user submitted and save it as form
        form = NewTaskForm(request.POST)

        # Check if form data is valid (server-side)
        if form.is_valid():

            # Isolate the task from the 'cleaned' version of form data
            task = form.cleaned_data["task"]

            # Add the new task to our list of tasks
            request.session["tasks"] += [task]

            # Redirect user to list of tasks
            return HttpResponseRedirect(reverse("tasks:index"))
        else:
```



```
            # If the form is invalid, re-render the page with existing information.
            return render(request, "tasks/add.html", {
                "form": form
            })

    return render(request, "tasks/add.html", {
        "form": NewTaskForm()
    })
```

Finally, before Django will be able to store this data, we must run `python manage.py migrate` in the terminal. Next week we'll talk more about what a migration is, but for now just know that the above command allows us to store sessions.

That's all for this lecture! Next time we'll be working on using Django to store, access, and manipulate data.

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan)
(https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 4

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, and started using Django to create web applications.
- Today, we'll learn about using SQL and Django models to efficiently store and access data.

SQL

SQL (https://www.w3schools.com/sql/), or Structured Query Language, is a programming language that allows us to update and query databases.

Databases

Before we get into how to use the SQL language, we should discuss how our data is stored. When using SQL, we'll work with a relational database (https://www.oracle.com/database/what-is-a-relational-database/#:~:text=A%20relational%20database%20is%20a,of%20representing%20data%20in%20tables.) where we can find all of our data stored in a number of tables (https://www.essentialsql.com/what-is-a-database-table/). Each of these tables is made up of a set number of columns and a flexible number of rows.

To illustrate how to work with SQL, we'll use the example of a website for an airline used to keep track of flights and passengers. In the following table, we see that we're keeping track of a number of flights, each of which has an `origin`, a `destination`, and a `duration`.

origin	destination	duration
New York	London	415
Shanghai	Paris	760
Istanbul	Tokyo	700
New York	Paris	435
Moscow	Paris	245
Lima	New York	455

There are several different relational database management systems that are commonly used to store information, and that can easily interact with SQL commands:

- MySQL (https://www.mysql.com/)
- PostgreSQL (https://www.postgresql.org/)
- SQLite (https://www.sqlite.org/index.html)
- ...

The first two, MySQL and PostgreSQL, are heavier-duty database management systems that are typically run on servers separate from those running a website. SQLite, on the other hand, is a lighter-weight system that can store all of its data in a single file. We'll be using SQLite throughout this course, as it is the default system used by Django.

Column Types

Just as we worked with several different variable types in Python, SQLite has types (https://www.sqlite.org/datatype3.html) that represent different forms of information. Other management systems may have different data types, but all are fairly similar to those of SQLite:

- `TEXT`: For strings of text (Ex. a person's name)
- `NUMERIC`: A more general form of numeric data (Ex. A date or boolean value)
- `INTEGER`: Any non-decimal number (Ex. a person's age)

- `REAL`: Any real number (Ex. a person's weight)
- `BLOB` (Binary Large Object): Any other binary data that we may want to store in our database (Ex. an image)

Tables

Now, to actually get started with using SQL to interact with a database, let's begin by creating a new table. The command to create a new table (https://www.w3schools.com/sql/sql_create_table.asp) looks something like this:

```
CREATE TABLE flights(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    origin TEXT NOT NULL,
    destination TEXT NOT NULL,
    duration INTEGER NOT NULL
);
```

In the above command, we're creating a new table that we've decided to call `flights`, and we've added four columns to this table:

1. `id`: It is often helpful to have an number that allows us to uniquely identify each row in a table. Here we have specified that `id` is an integer, and also that it is our primary key (https://www.w3schools.com/sql/sql_primarykey.ASP), meaning it is our unique identifier. We have additionally specified that it will `AUTOINCREMENT`, which means we will not have to provide an id every time we add to the table because it will be done automatically.
2. `origin`: Here we've specified that this will be a text field, and by writing `NOT NULL` we have required that it have a value.
3. `destination`: Again we've specified that this will be a text field and prevented it from being null.
4. `duration`: Again this value cannot be null, but this time it is represented by an integer rather than as text.

We just saw the `NOT NULL` and `PRIMARY KEY` constraint when making a column, but there are several other constraints (https://www.tutorialspoint.com/sqlite/sqlite_constraints.htm) available to us:

- `CHECK`: Makes sure certain constraints are met before allowing a row to be added/modified
- `DEFAULT`: Provides a default value if no value is given
- `NOT NULL`: Makes sure a value is provided
- `PRIMARY KEY`: Indicates this is the primary way of searching for a row in the database
- `UNIQUE`: Ensures that no two rows have the same value in that column.

- ...

Now that we've seen how to create a table, let's look at how we can add rows to it. In SQL, we do this using the `INSERT` command:

```
INSERT INTO flights
    (origin, destination, duration)
    VALUES ("New York", "London", 415);
```

In the above command, we've specified the table name we wish to insert into, then provided a list of the column names we will be providing information on, and then specified the `VALUES` we would like to fill that row in the table, making sure the `VALUES` come in the same order as our corresponding list of columns. Note that we don't need to provide a value for `id` because it is automatically incrementing.

SELECT

Once a table has been populated with some rows, we'll probably want a way to access data within that table. We do this using SQL's SELECT (https://www.w3schools.com/sql/sql_select.asp) query. The simplest `SELECT` query into our flights table might look something like this:

```
SELECT * FROM flights;
```

The above command (*) retrieves all of the data from our flights table

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

It may be the case though that we don't really need all of the columns from the database, just origin and destination. To access just these columns, we can replace the * with the column names we would like access to. The following query returns all of the origins and destinations.


```
SELECT origin, destination FROM flights;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

As our tables get larger and larger, we will also want to narrow down which rows our query returns. We do this by adding a WHERE (https://www.w3schools.com/sql/sql_where.asp) followed by some condition. For example, the following command selects only row with an `id` of `3`:

```
SELECT * FROM flights WHERE id = 3;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

e can filter by any column, not just `id`!

```
SELECT * FROM flights WHERE origin = "New York";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

Working with SQL in the Terminal

Now that we know some basic SQL commands, let's test them out in the terminal! In order to work with SQLite on your computer, you must first download SQLite (https://www.sqlite.org/download.html). (We won't use it in lecture, but you can also download DB Browser (https://sqlitebrowser.org/dl/) for a more user-friendly way to run SQL queries.)

We can start by creating a file for our database either by manually creating a new file, or running `touch flights.sql` in the terminal. Now, if we run `sqlite3 flights.sql` in the terminal, we'll be brought to a SQLite prompt where we can run SQL commands:

```
# Entering into the SQLite Prompt
(base) % sqlite3 flights.sql
SQLite version 3.26.0 2018-12-01 12:34:55
Enter ".help" for usage hints.

# Creating a new Table
sqlite> CREATE TABLE flights(
   ...>     id INTEGER PRIMARY KEY AUTOINCREMENT,
   ...>     origin TEXT NOT NULL,
   ...>     destination TEXT NOT NULL,
   ...>     duration INTEGER NOT NULL
   ...> );

# Listing all current tables (Just flights for now)
sqlite> .tables
flights

# Querying for everything within flights (Which is now empty)
sqlite> SELECT * FROM flights;

# Adding one flight
```



```
sqlite> INSERT INTO flights
   ...>     (origin, destination, duration)
   ...>     VALUES ("New York", "London", 415);

# Checking for new information, which we can now see
sqlite> SELECT * FROM flights;
1|New York|London|415

# Adding some more flights
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Shanghai", "Paris
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Istanbul", "Tokyo
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("New York", "Paris
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Moscow", "Paris",
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Lima", "New York"

# Querying this new information
sqlite> SELECT * FROM flights;
1|New York|London|415
2|Shanghai|Paris|760
3|Istanbul|Tokyo|700
4|New York|Paris|435
5|Moscow|Paris|245
6|Lima|New York|455

# Changing the settings to make output more readable
sqlite> .mode columns
sqlite> .headers yes

# Querying all information again
sqlite> SELECT * FROM flights;
id          origin      destination  duration
----------  ----------  -----------  ----------
1           New York    London       415
2           Shanghai    Paris        760
3           Istanbul    Tokyo        700
4           New York    Paris        435
5           Moscow      Paris        245
6           Lima        New York     455

# Searching for just those flights originating in New York
sqlite> SELECT * FROM flights WHERE origin = "New York";
id          origin      destination  duration
----------  ----------  -----------  ----------
1           New York    London       415
4           New York    Paris        435
```

We can also use more than just equality to filter out our flights. For integer and real values, we can use greater than or less than:

```
SELECT * FROM flights WHERE duration > 500;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

And we can also use other logic (AND, OR (https://www.w3schools.com/sql/sql_and_or.asp)) like in Python:

```
SELECT * FROM flights WHERE duration > 500 AND destination = "Paris";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

```
SELECT * FROM flights WHERE duration > 500 OR destination = "Paris";
```


id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

We can also use the keyword IN (https://www.w3schools.com/sql/sql_in.asp) to see if a bit of data is one of several options:

```
SELECT * FROM flights WHERE origin IN ("New York", "Lima");
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

We can even use regular expressions to search words more broadly using the LIKE (https://www.w3schools.com/sql/sql_like.asp) keyword. The below query finds all results with an `a` in the origin, by using `%` as a wildcard character.

```
SELECT * FROM flights WHERE origin LIKE "%a%";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

Functions

There are also a number of SQL functions we can apply to the results of a query. These can be useful if we don't need all of the data returned by a query, but just some summary statistics of the data.

- AVERAGE (https://www.w3schools.com/sql/sql_count_avg_sum.asp)
- COUNT (https://www.w3schools.com/sql/sql_count_avg_sum.asp)
- MAX (https://www.w3schools.com/sql/sql_min_max.asp)
- MIN (https://www.w3schools.com/sql/sql_min_max.asp)
- SUM (https://www.w3schools.com/sql/sql_count_avg_sum.asp)
- ...

UPDATE

We've now seen how to add to and search tables, but we may also want to be able update rows of a table that already exist. We do this using the UPDATE (https://www.w3schools.com/sql/sql_update.asp) command as shown below. As you may have guessed by reading this out loud, the command finds any flights that go from New York to London, and then sets their durations to 430.

```
UPDATE flights
    SET duration = 430
    WHERE origin = "New York"
    AND destination = "London";
```


DELETE

We also might want the ability to delete rows from our database, and we can do this using the DELETE (https://www.w3schools.com/sql/sql_delete.asp) command. The following code will remove all flights that land in Tokyo:

```
DELETE FROM flights WHERE destination = "Tokyo";
```

Other Clauses

There are a number of additional clauses we can use to control queries coming back to us

- LIMIT (https://www.w3schools.com/sql/sql_top.asp): Limits the number of results returned by a query
- ORDER BY (https://www.w3schools.com/sql/sql_orderby.asp): Orders the results based on a specified column
- GROUP BY (https://www.w3schools.com/sql/sql_groupby.asp): Groups results by a specified column
- HAVING (https://www.w3schools.com/sql/sql_having.asp): Allows for additional constraints based on the number of results

Joining Tables

So far, we've only been working with one table at a time, but many databases in practice are populated by a number of tables that all relate to each other in some way. In our flights example, let's imagine we also want to add an airport code to go with the city. The way our table is currently set up, we would have to add two more columns to go with each row. We would also be repeating information, as we would have to write in multiple places that city X is associated with code Y.

One way we can solve this problem is by deciding to have one table that keeps track of flights, and then another table keeping track of airports. The second table might look something like this

id	code	city
1	JFK	New York
2	PVG	Shanghai
3	IST	Istanbul
4	LHR	London
5	SVO	Moscow
6	LIM	Lima
7	CDG	Paris
8	NRT	Tokyo

Now we have a table relating codes and cities, rather than storing an entire city name in our flights table, it will save storage space if we're able to just save the `id` of that airport. Therefore, we should rewrite the flights table accordingly. Since we're using the `id` column of the airports table to populate `origin_id` and `destination_id`, we call those values Foreign Keys (https://www.w3schools.com/sql/sql_foreignkey.asp)

id	origin_id	destination_id	duration
1	1	4	415
2	2	7	760
3	3	8	700
4	1	7	435
5	5	7	245
6	6	1	455

In addition to flights and airports, an airline might also want to store data about its passengers, like which flight each passenger will be on. Using the power of relational databases, we can add another table that stores first and last names, and a foreign key representing the flight they are on

`passengers`

id	first	last	flight_id
1	Harry	Potter	1
2	Ron	Weasley	1
3	Hermione	Granger	2
4	Draco	Malfoy	4
5	Luna	Lovegood	6
6	Ginny	Weasley	6

We can do even better than this though, as the same person may be on more than one flight. To account for this, we can create a `people` table that stores first and last names, and a `passengers` table that pairs people with flights

`people`

id	first	last
1	Harry	Potter
2	Ron	Weasley
3	Hermione	Granger
4	Draco	Malfoy
5	Luna	Lovegood
6	Ginny	Weasley

passengers

person_id	flight_id
1	1
2	1
2	4
3	2
4	4
5	6
6	6

ecause in this case a single person can be on many flights and a single flight can have many people, we call the relationship between `flights` and `people` a **Many to Many** relationship. The `passengers` table that connects the two is known as an **association table**.

JOIN Query

Although our data is now more efficiently stored, it seems like it may be harder to query our data. Thankfully, SQL has a JOIN (https://www.w3schools.com/sql/sql_join.asp) query where we can combine two tables for the purposes of another query.

For example, let's say we want to find the origin, destination, and first name of every trip a passenger is taking. Also for simplicity in this table, we're going to be using the unoptimized `passengers` table that includes the flight id, first name, and last name. The first part of this query looks fairly familiar:

```
SELECT first, origin, destination
FROM ...
```

But we run into a problem here because `first` is stored in the `passengers` table, while `origin` and `destination` are stored in the `flights` table. We solve this by joining the two tables using the fact that `flight_id` in the `passengers` table corresponds to `id` in the `flights` table:

```
SELECT first, origin, destination
FROM flights JOIN passengers
ON passengers.flight_id = flights.id;
```


first	origin	destination
Harry	New York	London
Ron	New York	London
Hermione	Shanghai	Paris
Draco	New York	Paris
Luna	Lima	New York
Ginny	Lima	New York

We've just used something called an INNER JOIN (https://www.w3schools.com/sql/sql_join_inner.asp), which means we are ignoring rows that have no matches between the tables, but there are other types of joins, including **LEFT JOIN**s (https://www.w3schools.com/sql/sql_join_left.asp), **RIGHT JOIN**s (https://www.w3schools.com/sql/sql_join_right.asp), and **FULL OUTER JOIN**s (https://www.w3schools.com/sql/sql_join_full.asp), which we won't discuss here in detail.

Indexing

One way we can make our queries more efficient when dealing with large tables is to create an index similar to the index you might see in the back of a textbook. For example, if we know that we'll often look up passengers by their last name, we could create an index from last name to id using the command:

```
CREATE INDEX name_index ON passengers (last);
```

SQL Vulnerabilities

Now that we know the basics of using SQL to work with data, it's important to point out the main vulnerabilities associated with using SQL. We'll start with SQL Injection (https://www.w3schools.com/sql/sql_injection.asp).

A SQL injection attack is when a malicious user enters SQL code as input on a site in order to bypass the sites security measures. For example, let's say we have a table storing usernames and passwords, and then a login form on the home site of a page. We may search for the user using a query such as:

```
SELECT * FROM users
WHERE username = username AND password = password;
```

A user named Harry might go to this site and type `harry` as a username and `12345` as a password, in which case the query would look like this:

```
SELECT * FROM users
WHERE username = "harry" AND password = "12345";
```

A hacker, on the other hand, might type `harry" --` as a username and nothing as a password. It turns out that `--` stands for a comment in SQL, meaning the query would look like:

```
SELECT * FROM users
WHERE username = "harry"--" AND password = "12345";
```

Because in this query the password checking has been commented out, the hacker can log into Harry's account without knowing their password. To solve this problem, we can use:

- Escape characters to make sure SQL treats the input as plain text and not as SQL code.
- An abstraction layer on top of SQL which includes its own escape sequence, so we don't have to write SQL queries ourselves.

The other main vulnerability when it comes to SQL is known as a Race Condition (https://searchstorage.techtarget.com/definition/race-condition#:~:text=A%20race%20condition%20is%20an,sequence%20to%20be%20done%20correctly.).

A race condition is a situation that occurs when multiple queries to a database occur simultaneously. When these are not adequately handled, problems can arise in the precise times that databases are updated. For example, let's say I have $150 in my bank account. A race condition could occur if I log into my bank account on both my phone and my laptop, and attempt to withdraw $100 on each device. If the bank's software developers did not deal with race conditions correctly, then I may be able to withdraw $200 from an account with only $150 in it. One potential solution for this problem would be locking the database. We could not allow any other interaction with the database until one transaction has been completed. In the bank example, after clicking navigating to the "Make a Withdrawl" page on my computer, the bank might not allow me to navigate to that page on my phone.

Django Models

Django Models (https://docs.djangoproject.com/en/4.0/topics/db/models/) are a level of abstraction (https://techterms.com/definition/abstraction) on top of SQL that allow us to work with

databases using Python classes and objects rather than direct SQL queries.

Let's get started on using models by creating a django project for our airline, and creating an app within that project.

```
django-admin startproject airline
cd airline
python manage.py startapp flights
```

Now we'll have to go through the process of adding an app as usual:

1. Add `flights` to the `INSTALLED_APPS` list in `settings.py`
2. Add a route for `flights` in `urls.py`:

   ```
   path("flights/", include("flights.urls")),
   ```

3. Create a `urls.py` file within the `flights` application. And fill it with standard `urls.py` imports and lists.

Now, rather than creating actual paths and getting started on `views.py`, we'll create some models in the `models.py` file. In this file, we'll outline what data we want to store in our application. Then, Django will determine the SQL syntax necessary to store information on each of our models. Let's take a look at what a model for a single flight might look like:

```
class Flight(models.Model):
    origin = models.CharField(max_length=64)
    destination = models.CharField(max_length=64)
    duration = models.IntegerField()
```

Let's take a look at what's going on in this model definition:

- In the first line, we create a new model that **extends** Django's model class.
- Below, we add fields for origin, destination, and duration. The first two are Char Fields (https://docs.djangoproject.com/en/4.0/ref/forms/fields/#charfield), meaning they store strings, and the third is an Integer Field (https://docs.djangoproject.com/en/4.0/ref/forms/fields/#integerfield). These are just two of many built-in Django Field classes (https://docs.djangoproject.com/en/4.0/ref/forms/fields/#built-in-field-classes)
- We specify maximum lengths of 64 for the two Character Fields. you can check the specifications available for a given field by checking the documentation (https://docs.djangoproject.com/en/4.0/ref/forms/fields/#built-in-field-classes).

Migrations

Now, even though we've created a model, we do not yet have a database to store this information. to create a database from our models, we navigate to the main directory of our project and run the command.

```
python manage.py makemigrations
```

This command creates some Python files that will create or edit our database to be able to store what we have in our models. You should get an output that looks something like the one below, and if you navigate to your `migrations` directory, you'll notice a new file was created for us

```
Migrations for 'flights':
  flights/migrations/0001_initial.py
    - Create model Flight
```

Next, to apply these migrations to our database, we run the command

```
python manage.py migrate
```

Now, you'll see some default migrations have been applied along with our own, and you'll also notice that we now have a file called `db.sqlite3` in our project's directory


```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, flights, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying flights.0001_initial... OK
  Applying sessions.0001_initial... OK
(base) cleggett@Connors-MacBook-Pro airline %
```

Shell

Now, to begin working adding information to and manipulating this database, we can enter Django's shell where we can run Python commands within our project.

```
python manage.py shell
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
# Import our flight model
In [1]: from flights.models import Flight

# Create a new flight
In [2]: f = Flight(origin="New York", destination="London", duration=415)

# Instert that flight into our database
In [3]: f.save()

# Query for all flights stored in the database
In [4]: Flight.objects.all()
Out[4]: <QuerySet [<Flight: Flight object (1)>]>
```

When we query our database, we see that we get just one flight called `Flight object (1)`. This isn't a very informative name, but we can fix that. Inside `models.py`, we'll define a `__str__` function that provides instructions for how to turn a Flight object into a string:

```
class Flight(models.Model):
    origin = models.CharField(max_length=64)
    destination = models.CharField(max_length=64)
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origin} to {self.destination}"
```

Now, when we go back to the shell, our output is a bit more readable:

```
# Create a variable called flights to store the results of a query
In [7]: flights = Flight.objects.all()

# Displaying all flights
In [8]: flights
Out[8]: <QuerySet [<Flight: 1: New York to London>]>

# Isolating just the first flight
In [9]: flight = flights.first()

# Printing flight information
In [10]: flight
Out[10]: <Flight: 1: New York to London>

# Display flight id
In [11]: flight.id
Out[11]: 1

# Display flight origin
In [12]: flight.origin
Out[12]: 'New York'

# Display flight destination
In [13]: flight.destination
Out[13]: 'London'

# Display flight duration
In [14]: flight.duration
Out[14]: 415
```

This is a good start, but thinking back to earlier, we don't want to have to store the city name as an origin and destination for every flight, so we probably want another model for an airport that is somehow related to the flight model:

```
class Airport(models.Model):
    code = models.CharField(max_length=3)
    city = models.CharField(max_length=64)
```



```
    def __str__(self):
        return f"{self.city} ({self.code})"

class Flight(models.Model):
    origin = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="depart
    destination = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="a
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origin} to {self.destination}"
```

We've seen everything in our new `Airport` class before, but the changes to the `origin` and `destination` fields within the `Flight` class are new to us:

- We specify that the `origin` and `destination` fields are each Foreign Keys (https://docs.djangoproject.com/en/4.0/topics/db/examples/many_to_one/), which means they refer to another object.
- By entering `Airport` as our first argument, we are specifying the type of object this field refers to.
- The next argument, `on_delete=models.CASCADE` gives instructions for what should happen if an airport is deleted. In this case, we specify that when an airport is deleted, all flights associated with it should also be deleted. There are several other options (https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.ForeignKey.on_delete) in addition to `CASCADE`.
- We provide a related name (https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.ForeignKey.related_name), which gives us a way to search for all flights with a given airport as their origin or destination.

Every time we make changes in `models.py`, we have to make migrations and then migrate. Note that you may have to delete your existing flight from New York to London, as it doesn't fit in with the new database structure.

```
# Create New Migrations
python manage.py makemigration

# Migrate
python manage.py migrate
```

Now, let's try these new models out in the Django shell:

```
# Import all models
In [1]: from flights.models import *
```

```
# Create some new airports
In [2]: jfk = Airport(code="JFK", city="New York")
In [4]: lhr = Airport(code="LHR", city="London")
In [6]: cdg = Airport(code="CDG", city="Paris")
In [9]: nrt = Airport(code="NRT", city="Tokyo")

# Save the airports to the database
In [3]: jfk.save()
In [5]: lhr.save()
In [8]: cdg.save()
In [10]: nrt.save()

# Add a flight and save it to the database
f = Flight(origin=jfk, destination=lhr, duration=414)
f.save()

# Display some info about the flight
In [14]: f
Out[14]: <Flight: 1: New York (JFK) to London (LHR)>
In [15]: f.origin
Out[15]: <Airport: New York (JFK)>

# Using the related name to query by airport of arrival:
In [17]: lhr.arrivals.all()
Out[17]: <QuerySet [<Flight: 1: New York (JFK) to London (LHR)>]>
```

Starting our application

We can now begin to build an application around this process of using models to interact with a database. Let's begin by creating an index route for our airline. Inside `urls.py`:

```
urlpatterns = [
    path('', views.index, name="index"),
]
```

Inside `views.py`:

```
from django.shortcuts import render
from .models import Flight, Airport

# Create your views here.

def index(request):
    return render(request, "flights/index.html", {
        "flights": Flight.objects.all()
    })
```

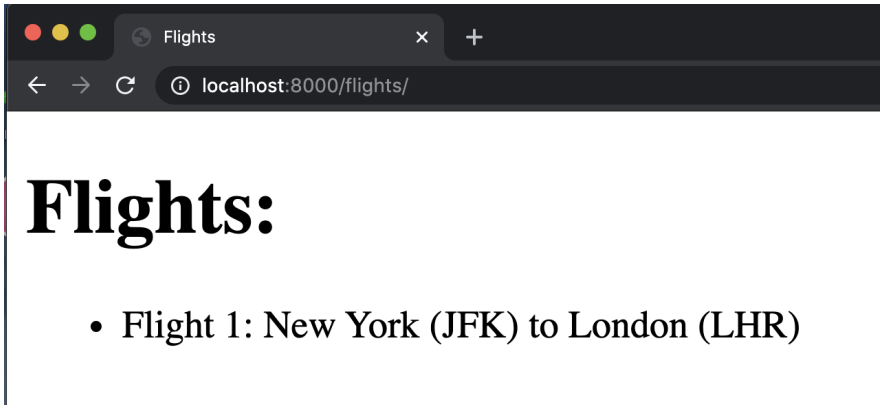
Inside our new `layout.html` file:


```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Flights</title>
    </head>
    <body>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

Inside a new `index.html` file:

```
{% extends "flights/layout.html" %}

{% block body %}
    <h1>Flights:</h1>
    <ul>
        {% for flight in flights %}
            <li>Flight {{ flight.id }}: {{ flight.origin }} to {{ flight.destination }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

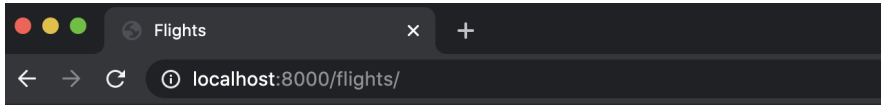
What we've done here is created a default page where we have a list of all flights we've created so far. When we open up the page now, it looks like this



Now, let's add some more flights to our application by returning to the Django shell:

```
# Using the filter command to find all airports based in New York
In [3]: Airport.objects.filter(city="New York")
Out[3]: <QuerySet [<Airport: New York (JFK)>]>

# Using the get command to get only one airport in New York
In [5]: Airport.objects.get(city="New York")
Out[5]: <Airport: New York (JFK)>

# Assigning some airports to variable names:
In [6]: jfk = Airport.objects.get(city="New York")
In [7]: cdg = Airport.objects.get(city="Paris")

# Creating and saving a new flight:
In [8]: f = Flight(origin=jfk, destination=cdg, duration=435)
In [9]: f.save()
```

Now, when we visit our site again



Flights:

- Flight 1: New York (JFK) to London (LHR)
- Flight 2: New York (JFK) to Paris (CDG)

Django Admin

Since it is so common for developers to have to create new objects like we've been doing in the shell, Django comes with a default admin interface (https://docs.djangoproject.com/en/4.0/ref/contrib/admin/) that allows us to do this more easily. To begin using this tool, we must first create an administrative user:

```
(base) cleggett@Connors-MacBook-Pro airline % python manage.py createsuperuser
Username: user_a
Email address: a@a.com
Password:
```

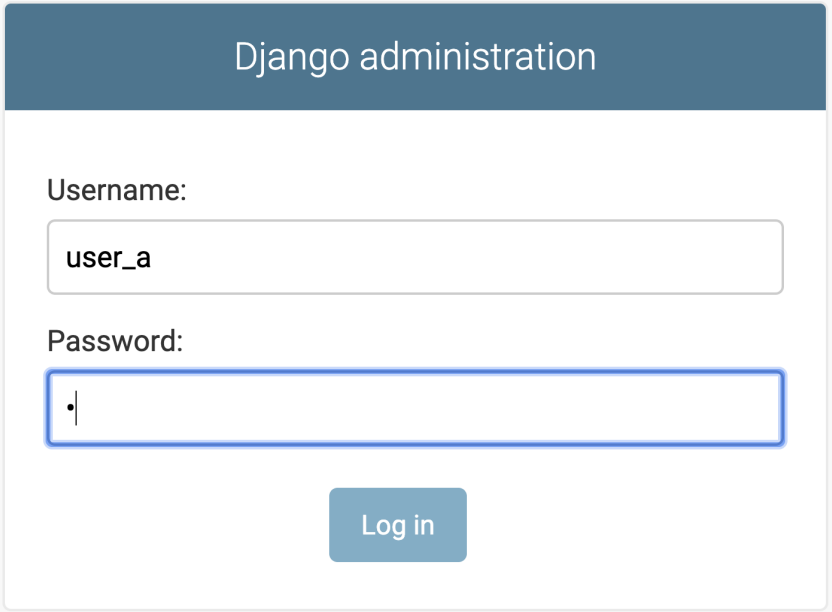


```
Password (again):
Superuser created successfully.
```

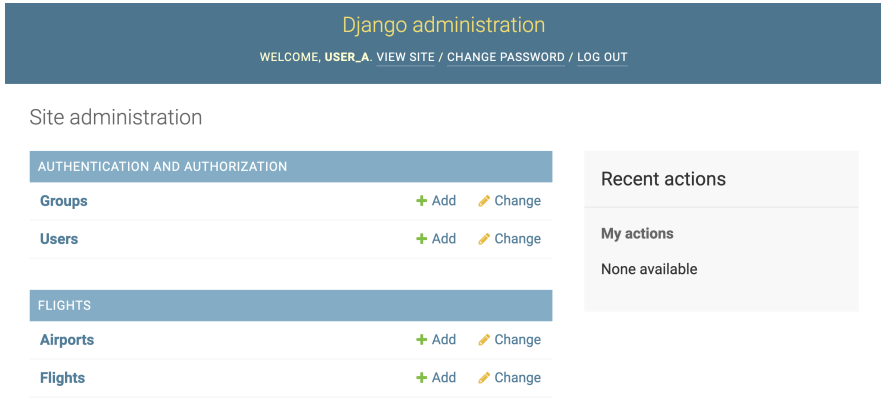
Now, we must add our models to the admin application by entering the `admin.py` file within our app, and importing and registering our models. This tells Django which models we would like to have access to in the admin app.

```
from django.contrib import admin
from .models import Flight, Airport

# Register your models here.
admin.site.register(Flight)
admin.site.register(Airport)
```

Now, when we visit our site and add `/admin` to the url, we can log into a page that looks like this



After loggin in, you'll be brought to a page like the one below where you can create, edit, and delete objects stored in the database



Now, let's add a few more pages to our site. We'll begin by adding the ability to click on a flight to get more information about it. To do this, let's create a URL path that includes the `id` of a flight:

```
path("<int:flight_id>", views.flight, name="flight")
```

Then, in `views.py` we will create a `flight` function that takes in a flight id and renders a new html page:

```
def flight(request, flight_id):
    flight = Flight.objects.get(id=flight_id)
    return render(request, "flights/flight.html", {
        "flight": flight
    })
```

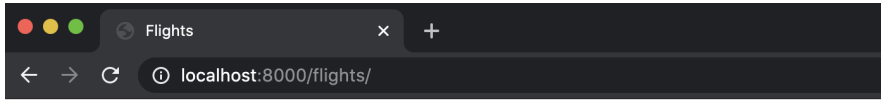
Now we'll create a template to display this flight information with a link back to the home page

```
{% extends "flights/layout.html" %}

{% block body %}
    <h1>Flight {{ flight.id }}</h1>
    <ul>
        <li>Origin: {{ flight.origin }}</li>
        <li>Destination: {{ flight.destination }}</li>
        <li>Duration: {{ flight.duration }} minutes</li>
    </ul>
    <a href="{% url 'index' %}">All Flights</a>
{% endblock %}
```


Finally, we need to add the ability to link from one page to another, so we'll modify our index page to include links:

```
{% extends "flights/layout.html" %}

{% block body %}
    <h1>Flights:</h1>
    <ul>
        {% for flight in flights %}
            <li><a href="{% url 'flight' flight.id %}">Flight {{ flight.id }}</a>: {{
        {% endfor %}
    </ul>
{% endblock %}
```

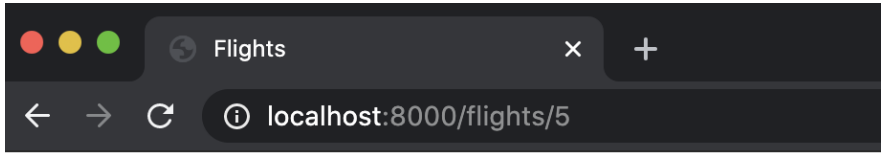
Now our homepage looks like this



Flights:

- Flight 1: New York (JFK) to London (LHR)
- Flight 2: New York (JFK) to Paris (CDG)
- Flight 3: New York (JFK) to Istanbul (IST)
- Flight 4: Tokyo (NRT) to Lima (LIM)
- Flight 5: London (LHR) to Istanbul (IST)
- Flight 6: Paris (CDG) to Lima (LIM)

And when we click on flight 5, for example, we're brought to this page



Flight 5

- Origin: London (LHR)
- Destination: Istanbul (IST)
- Duration: 85 minutes

All Flights

Many-to-Many Relationships

Now, let's work on integrating passengers into our models. We'll create a passenger model to start:

```
class Passenger(models.Model):
    first = models.CharField(max_length=64)
    last = models.CharField(max_length=64)
    flights = models.ManyToManyField(Flight, blank=True, related_name="passengers")

    def __str__(self):
        return f"{self.first} {self.last}"
```

- As we discussed, passengers have a **Many to Many** relationship with flights, which we describe in Django using the ManyToManyField.
- The first argument in this field is the class of objects that this one is related to.

- We have provided the argument `blank=True` which means a passenger can have no flights
- We have added a `related_name` that serves the same purpose as it did earlier: it will allow us to find all passengers on a given flight.

To actually make these changes, we must make migrations and migrate. We can then register the Passenger model in `admin.py` and visit the admin page to create some passengers!

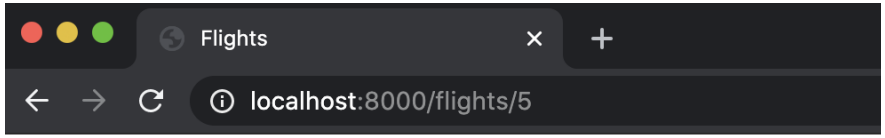
Now that we've added some passengers, let's update our flight page so that it displays all passengers on a flight. We'll first visit `views.py` and update our flight view to provide a list of passengers as context. We access the list using the related name we defined earlier.

```
def flight(request, flight_id):
    flight = Flight.objects.get(id=flight_id)
    passengers = flight.passengers.all()
    return render(request, "flights/flight.html", {
        "flight": flight,
        "passengers": passengers
    })
```

Now, add a list of passengers to `flight.html`:

```
<h2>Passengers:</h2>
<ul>
    {% for passenger in passengers %}
        <li>{{ passenger }}</li>
    {% empty %}
        <li>No Passengers.</li>
    {% endfor %}
</ul>
```

At this point, when we click on flight 5, we see



Flight 5

- Origin: London (LHR)
- Destination: Istanbul (IST)
- Duration: 85 minutes

Passengers:

- Ron Weasley
- Hermione Granger

All Flights

Now, let's work on giving visitors to our site the ability to book a flight. We'll do this by adding a booking route in `urls.py`:

```
path("<int:flight_id>/book", views.book, name="book")
```

Now, we'll add a book function to `views.py` that adds a passenger to a flight:


```
def book(request, flight_id):

    # For a post request, add a new flight
    if request.method == "POST":

        # Accessing the flight
        flight = Flight.objects.get(pk=flight_id)

        # Finding the passenger id from the submitted form data
        passenger_id = int(request.POST["passenger"])

        # Finding the passenger based on the id
        passenger = Passenger.objects.get(pk=passenger_id)

        # Add passenger to the flight
        passenger.flights.add(flight)

        # Redirect user to flight page
        return HttpResponseRedirect(reverse("flight", args=(flight.id,)))
```

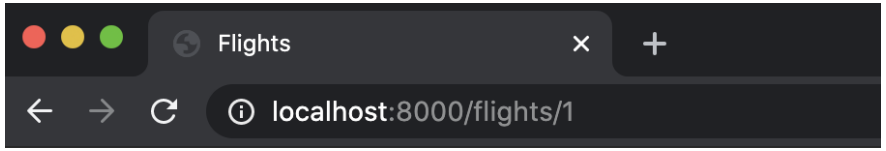
Next, we'll add some context to our flight template so that the page has access to everyone who is not currently a passenger on the flight using Django's ability to exclude (https://docs.djangoproject.com/en/4.0/topics/db/queries/#retrieving-specific-objects-with-filters) certain objects from a query:

```
def flight(request, flight_id):
    flight = Flight.objects.get(id=flight_id)
    passengers = flight.passengers.all()
    non_passengers = Passenger.objects.exclude(flights=flight).all()
    return render(request, "flights/flight.html", {
        "flight": flight,
        "passengers": passengers,
        "non_passengers": non_passengers
    })
```

Now, we'll add a form to our flight page's HTML using a select input field:

```
<form action="{% url 'book' flight.id %}" method="post">
    {% csrf_token %}
    <select name="passenger" id="">
        {% for passenger in non_passengers %}
            <option value="{{ passenger.id }}">{{ passenger }}</option>
        {% endfor %}
    </select>
    <input type="submit">
</form>
```

Now, let's see what the site looks like when I go to a flight page and then add a passenger



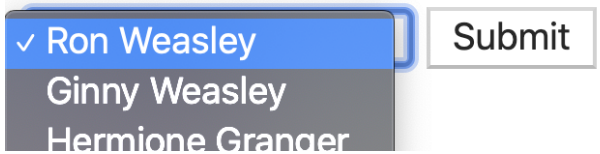
Flight 1

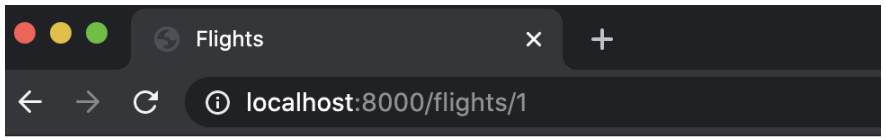
- Origin: New York (JFK)
- Destination: London (LHR)
- Duration: 414 minutes

Passengers:

- Harry Potter

Add Passenger





Flight 1

- Origin: New York (JFK)
- Destination: London (LHR)
- Duration: 414 minutes

Passengers:

- Harry Potter
- Ron Weasley

Add Passenger

Ginny Weasley Submit

All Flights

Another advantage of using the Django admin app is that it is customizable. For example, if we wish to see all aspects of a flight in the admin interface, we can create a new class within `admin.py` and add it as an argument when registering the `Flight` model:

```
class FlightAdmin(admin.ModelAdmin):
    list_display = ("id", "origin", "destination", "duration")

# Register your models here.
admin.site.register(Flight, FlightAdmin)
```

Now, when we visit the admin page for flights, we can see the `id` as well

	ID	ORIGIN	DESTINATION	DURATION
	6	Paris (CDG)	Lima (LIM)	405
	5	London (LHR)	Istanbul (IST)	85
	4	Tokyo (NRT)	Lima (LIM)	655
	3	New York (JFK)	Istanbul (IST)	455

Check out Django's admin documentation (https://docs.djangoproject.com/en/4.0/ref/contrib/admin/) to find more ways to customize the admin app.

Users

The last thing we'll discuss in lecture today is the idea of authentication, or allowing users to log in and out of a website. Fortunately, Django makes this very easy for us, so let's go through an example of how we would do this. We'll start by creating a new app called `users`. Here we'll go through all the normal steps of creating a new app, but in our new `urls.py` file, we'll add a few more routes:

```
urlpatterns = [
    path('', views.index, name="index"),
    path("login", views.login_view, name="login"),
    path("logout", views.logout_view, name="logout")
]
```

Let's begin by creating a form where a user can log in. We'll create a `layout.html` file as always, and then create a `login.html` file which contains a form, and that displays a message if one exists.

```
{% extends "users/layout.html" %}

{% block body %}
    {% if message %}
        <div>{{ message }}</div>
    {% endif %}

    <form action="{% url 'login' %}" method="post">
        {% csrf_token %}
        <input type="text", name="username", placeholder="Username">
        <input type="password", name="password", placeholder="Password">
        <input type="submit", value="Login">
    </form>
{% endblock %}
```

Now, in `views.py`, we'll add three functions:

```
def index(request):
    # If no user is signed in, return to login page:
    if not request.user.is_authenticated:
        return HttpResponseRedirect(reverse("login"))
    return render(request, "users/user.html")

def login_view(request):
    return render(request, "users/login.html")

def logout_view(request):
    # Pass is a simple way to tell python to do nothing.
    pass
```

Next, we can head to the admin site and add some users. After doing that, we'll go back to `views.py` and update our `login_view` function to handle a `POST` request with a username and password:

```
# Additional imports we'll need:
from django.contrib.auth import authenticate, login, logout

def login_view(request):
    if request.method == "POST":
        # Accessing username and password from form data
        username = request.POST["username"]
        password = request.POST["password"]

        # Check if username and password are correct, returning User object if so
        user = authenticate(request, username=username, password=password)

        # If user object is returned, log in and route to index page:
        if user:
            login(request, user)
            return HttpResponseRedirect(reverse("index"))
```



```
        # Otherwise, return login page again with new context
        else:
            return render(request, "users/login.html", {
                "message": "Invalid Credentials"
            })
    return render(request, "users/login.html")
```

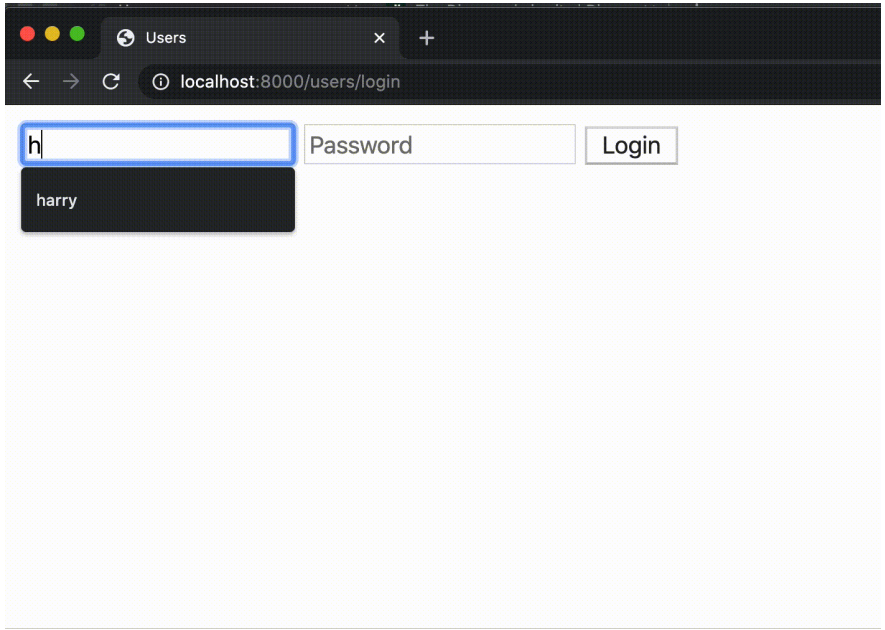
Now, we'll create the `user.html` file that the `index` function renders when a user is authenticated:

```
{% extends "users/layout.html" %}

{% block body %}
    <h1>Welcome, {{ request.user.first_name }}</h1>
    <ul>
        <li>Username: {{ request.user.username }}</li>
        <li>Email: {{ request.user.email }}</li>
    </ul>

    <a href="{% url 'logout' %}">Log Out</a>
{% endblock %}
```

Finally, to allow the user to log out, we'll update the `logout_view` function so that it uses Django's built-in `logout` function:

```
def logout_view(request):
    logout(request)
    return render(request, "users/login.html", {
                "message": "Logged Out"
            })
```

Now that we're finished, here's a demonstration of the site



That's all for this lecture! Next time, we'll learn our second programming language of the course: JavaScript.

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

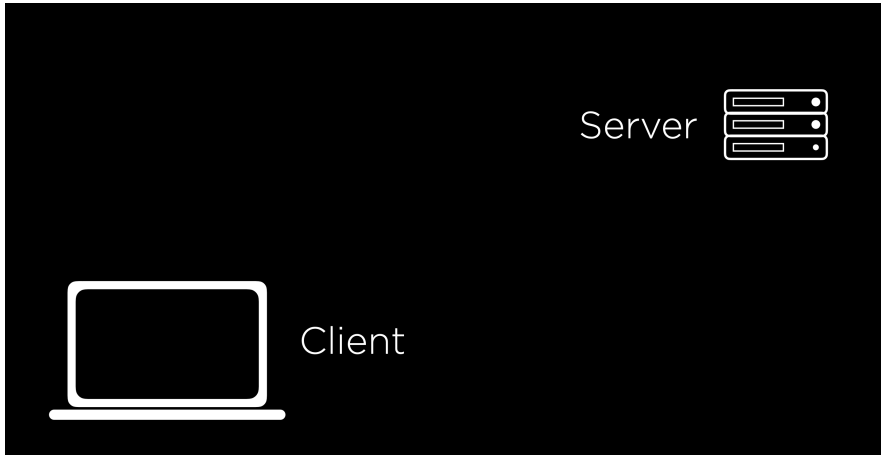
Lecture 5

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, started using Django to create web applications, and learned how to use Django models to store information in our sites.
- Today, we'll introduce a new programming language: JavaScript.

JavaScript

Let's begin by revisiting a diagram from a couple of lectures ago:



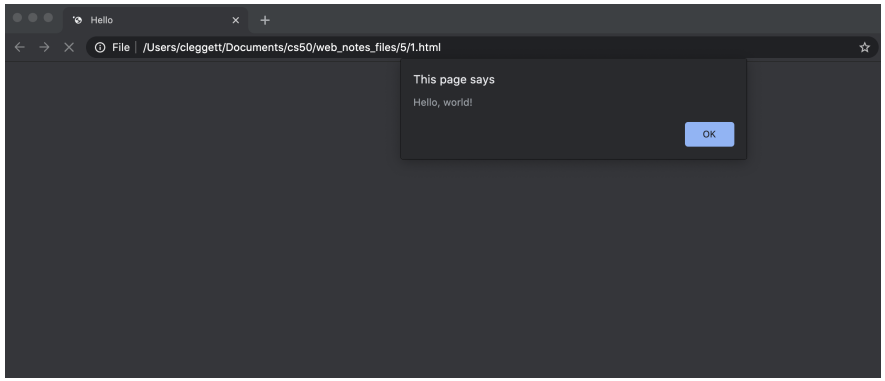
Recall that in most online interactions, we have a client/user that sends an HTTP Request to a server, which sends back an HTTP Response. All of the Python code we've written so far using Django has been running on a server. JavaScript will allow us to run code on the client side, meaning no interaction with the server is necessary while it's running, allowing our websites to become much more interactive.

In order to add some JavaScript to our page, we can add a pair of `<script>` tags somewhere in our HTML page. We use `<script>` tags to signal to the browser that anything we write in between the two tags is JavaScript code we wish to execute when a user visits our site. Our first program might look something like this:


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

The `alert` function in JavaScript displays a message to the user which they can then dismiss. To show where this would fit into an actual HTML document, here's an example of a simple page with some JavaScript:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Hello</title>
        <script>
            alert('Hello, world!');
        </script>
    </head>
    <body>
        <h1>Hello!</h1>
    </body>
</html>
```



Events

One feature of JavaScript that makes it helpful for web programming is that it supports Event-Driven Programming (https://medium.com/@vsvaibhav2016/introduction-to-event-driven-programming-28161b79c223).

Event-Driven Programming is a programming paradigm that centers around the detection of events, and actions that should be taken when an event is detected.

An event can be almost anything including a button being clicked, the cursor being moved, a response being typed, or a page being loaded. Just about everything a user does to interact with a

web page can be thought of as an event. In JavaScript, we use Event Listeners (https://www.w3schools.com/js/js_htmldom_eventlistener.asp) that wait for certain events to occur, and then execute some code.

Let's begin by turning our JavaScript from above into a function (https://www.w3schools.com/js/js_functions.asp) called `hello`:

```
function hello() {
    alert('Hello, world!')
}
```

Now, let's work on running this function whenever a button is clicked. To do this, we'll create an HTML button in our page with an `onclick` attribute, which gives the browser instructions for what should happen when the button is clicked:

```
<button onclick="hello()">Click Here</button>
```

These changes allow us to wait to run parts of our JavaScript code until a certain event occurs.

Variables

JavaScript is a programming language just like Python, C, or any other language you've worked with before, meaning it has many of the same features as other languages including variables. There are three keywords we can use to assign values in JavaScript:

- `var`: used to define a variable globally

```
var age = 20;
```

- `let`: used to define a variable that is limited in scope to the current block such as a function or loop

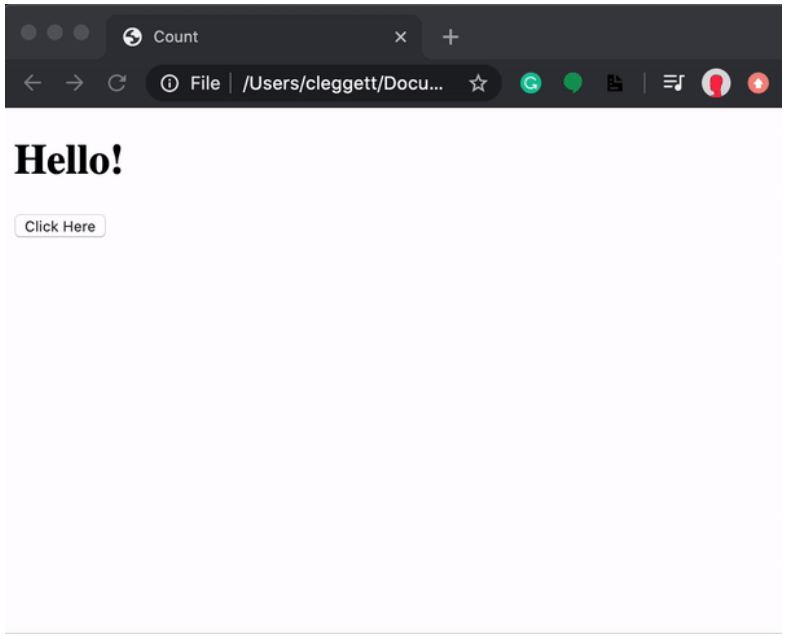
```
let counter = 1;
```

- `const`: used to define a value that will not change

```
const PI = 3.14;
```

For an example of how we can use a variable, let's take a look at a page that keeps track of a counter:


```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Count</title>
        <script>
            let counter = 0;
            function count() {
                counter++;
                alert(counter);
            }
        </script>
    </head>
    <body>
        <h1>Hello!</h1>
        <button onclick="count()">Count</button>
    </body>
</html>
```



`querySelector`

In addition to allowing us to display messages through alerts, JavaScript also allows us to change elements on the page. In order to do this, we must first introduce a function called

`document.querySelector`. This function searches for and returns elements of the DOM. For example, we would use:

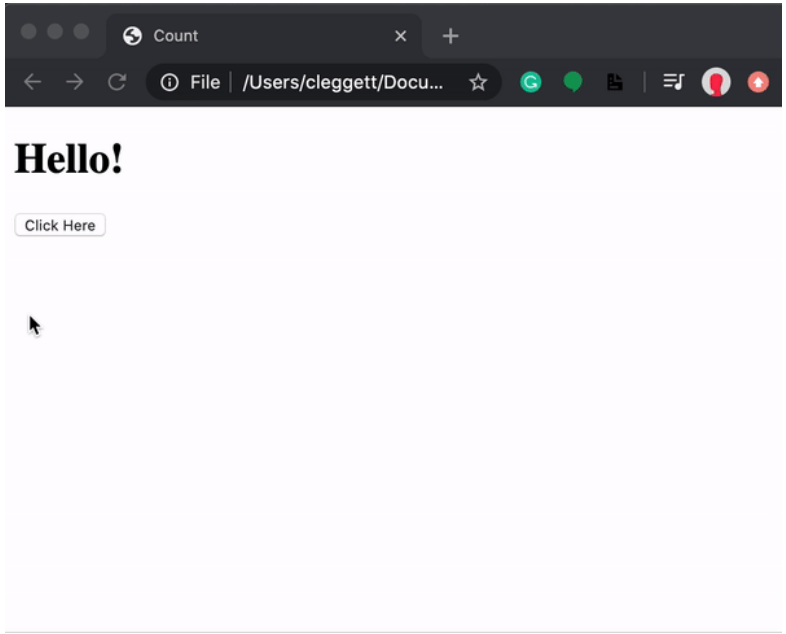
```
let heading = document.querySelector('h1');
```

to extract a heading. Then, to manipulate the element we've recently found, we can change its `innerHTML` property:

```
heading.innerHTML = `Goodbye!`;
```

Just as in Python, we can also take advantage of conditions (https://www.w3schools.com/js/js_if_else.asp) in JavaScript. For example, let's say rather than always changing our header to `Goodbye!`, we wish to toggle back and forth between `Hello!` and `Goodbye!`. Our page might then look something like the one below. Notice that in JavaScript, we use `===` as a stronger comparison between two items which also checks that the objects are of the same type. We typically want to use `===` whenever possible.

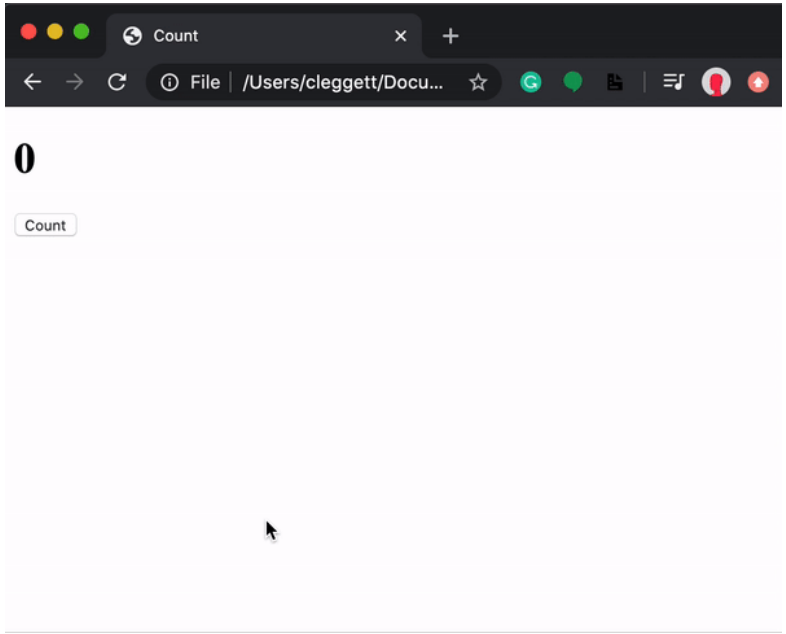
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Count</title>
        <script>
            function hello() {
                const header = document.querySelector('h1');
                if (header.innerHTML === 'Hello!') {
                    header.innerHTML = 'Goodbye!';
                }
                else {
                    header.innerHTML = 'Hello!';
                }
            }
        </script>
    </head>
    <body>
        <h1>Hello!</h1>
        <button onclick="hello()">Click Here</button>
    </body>
</html>
```

DOM Manipulation

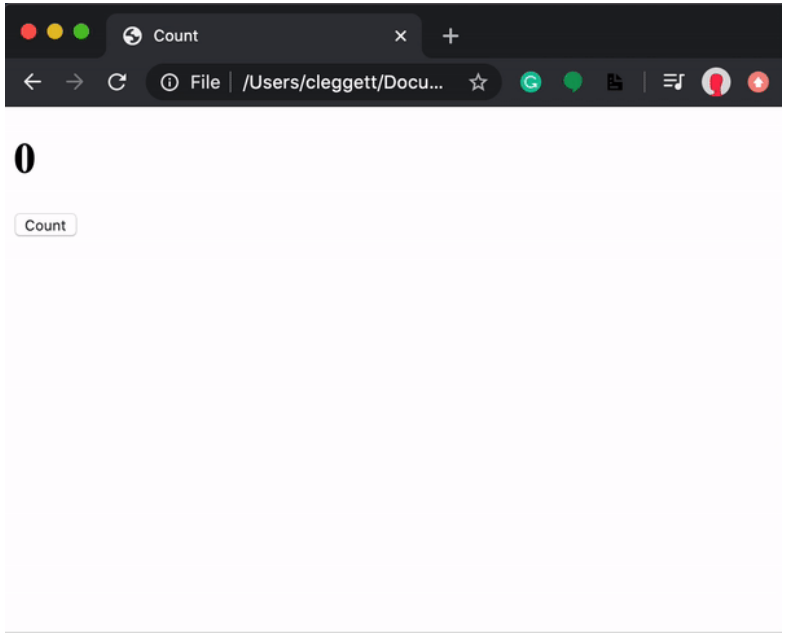
Let's use this idea of DOM manipulation to improve our counter page:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Count</title>
        <script>
            let counter = 0;
            function count() {
                counter++;
                document.querySelector('h1').innerHTML = counter;
            }
        </script>
    </head>
    <body>
        <h1>0</h1>
        <button onclick="count()">Count</button>
    </body>
</html>
```



We can make this page even more interesting by displaying an alert every time the counter gets to a multiple of ten. In this alert, we'll want to format a string to customize the message, which in JavaScript we can do using template literals (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals). Template literals require that there are backticks (`) around the entire expression and a $ and curly braces around any substitutions. For example, let's change our count function

```
function count() {
    counter++;
    document.querySelector('h1').innerHTML = counter;

    if (counter % 10 === 0) {
        alert(`Count is now ${counter}`)
    }
}
```

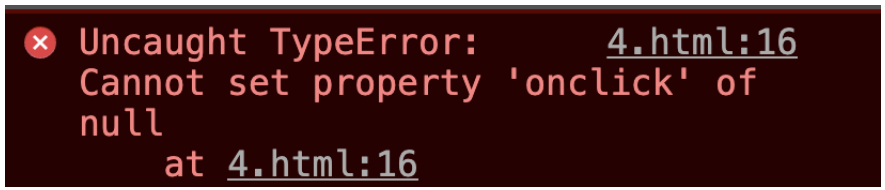



Now, let's look at some ways in which we can improve the design of this page. First, just as we try to avoid in-line styling with CSS, we want to avoid in-line JavaScript as much as possible. We can do this in our counter example by adding a line of script that changes the `onclick` attribute of a button on the page, and removing the `onclick` attribute from within the `button` tag.

```
document.querySelector('button').onclick = count;
```

One thing to notice about what we've just done is that we're not calling the `count` function by adding parentheses afterward, but instead just naming the function. This specifies that we only wish to call this function when the button is clicked. This works because, like Python, JavaScript supports functional programming, so functions can be treated as values themselves.

The above change alone is not enough though, as we can see by inspecting the page and looking at our browser's console:



This error came up because when JavaScript searched for an element using `document.querySelector('button')`, it didn't find anything. This is because it takes a small bit of time for the page to load, and our JavaScript code ran before the button had been rendered. To account for this, we can specify that code will run only after the page has loaded using the addEventListener (https://www.w3schools.com/jsref/met_document_addeventlistener.asp) function. This function takes in two arguments:

1. An event to listen for (eg: `'click'`)
2. A function to run when the event is detected (eg: `hello` from above)

We can use the function to only run the code once all content has loaded:

```
document.addEventListener('DOMContentLoaded', function() {
    // Some code here
});
```

In the example above, we've used an anonymous (https://www.w3schools.com/js/js_function_definition.asp) function, which is a function that is never given a name. Putting all of this together, our JavaScript now looks like this:

```
let counter = 0;

function count() {
    counter++;
    document.querySelector('h1').innerHTML = counter;

    if (counter % 10 === 0) {
        alert(`Count is now ${counter}`)
    }
}

document.addEventListener('DOMContentLoaded', function() {
    document.querySelector('button').onclick = count;
});
```

Another way that we can improve our design is by moving our JavaScript into a separate file. The way we do this is very similar to how we put our CSS in a separate file for styling:

1. Write all of your JavaScript code in a separate file ending in `.js`, maybe `index.js`.

2. Add a `src` attribute to the `<script>` tag that points to this new file.

For our counter page, we could have a file called `counter.html` that looks like this:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Count</title>
        <script src="counter.js"></script>
    </head>
    <body>
        <h1>0</h1>
        <button>Count</button>
    </body>
</html>
```

And a file called `counter.js` that looks like this:

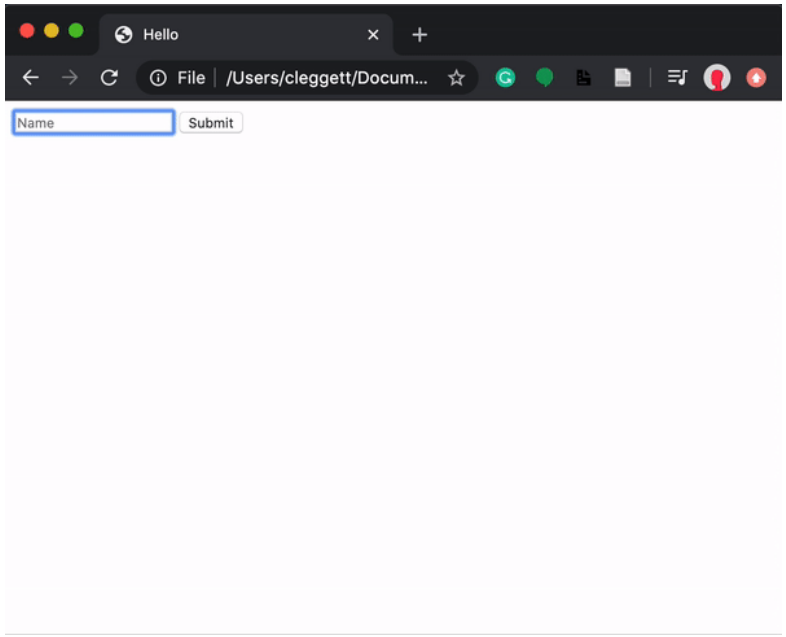
```
let counter = 0;

function count() {
    counter++;
    document.querySelector('h1').innerHTML = counter;

    if (counter % 10 === 0) {
        alert(`Count is now ${counter}`)
    }
}

document.addEventListener('DOMContentLoaded', function() {
    document.querySelector('button').onclick = count;
});
```

Having JavaScript in a separate file is useful for a number of reasons:

- Visual appeal: Our individual HTML and JavaScript files become more readable.
- Access among HTML files: Now we can have multiple HTML files that all share the same JavaScript.
- Collaboration: We can now easily have one person work on the JavaScript while another works on HTML.
- Importing: We are able to import JavaScript libraries that other people have already written. For example Bootstrap (https://getbootstrap.com/docs/4.5/getting-started/introduction/#js) has their own JavaScript library you can include to make your site more interactive.

Let's get started on another example of a page that can be a bit more interactive. Below, we'll create a page where a user can type in their name to get a custom greeting.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Hello</title>
    <script>
        document.addEventListener('DOMContentLoaded', function() {
            document.querySelector('form').onsubmit = function() {
                const name = document.querySelector('#name').value;
                alert(`Hello, ${name}`);
            };
        });
    </script>
</head>
<body>
    <form>
        <input autofocus id="name" placeholder="Name" type="text">
        <input type="submit">
    </form>
</body>
</html>
```



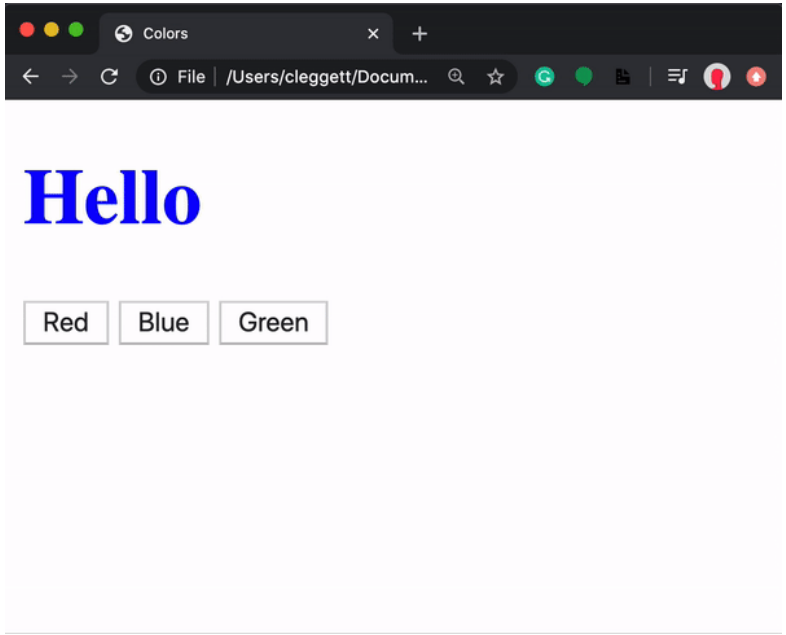
Some notes about the page above:

- We use the `autofocus` field in the `name` input to indicate that the cursor should be set inside that input as soon as the page is loaded.

- We use `#name` inside of `document.querySelector` to find an element with an `id` of `name`. We can use all the same selectors in this function as we could in CSS.
- We use the `value` attribute of an input field to find what is currently typed in.

We can do more than just add HTML to our page using JavaScript, we can also change the styling of a page! In the page below, we use buttons to change the color of our heading.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Colors</title>
    <script>
        document.addEventListener('DOMContentLoaded', function() {
            document.querySelectorAll('button').forEach(function(button) {
                button.onclick = function() {
                    document.querySelector("#hello").style.color = button.dataset.colo
                }
            });
        });
    </script>
</head>
<body>
    <h1 id="hello">Hello</h1>
    <button data-color="red">Red</button>
    <button data-color="blue">Blue</button>
    <button data-color="green">Green</button>
</body>
</html>
```



Some notes on the page above:

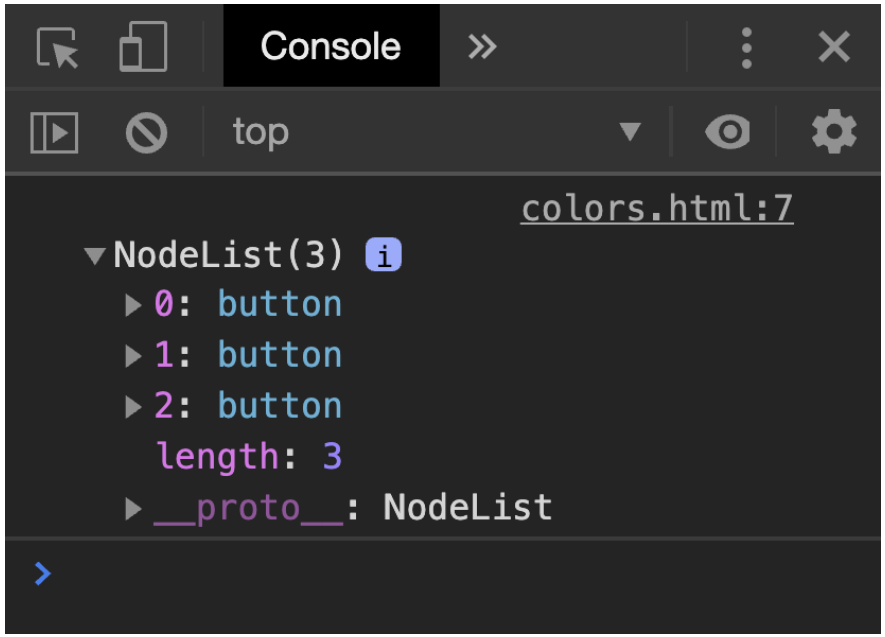
- We change the style of an element using the `style.SOMETHING` attribute.
- We use the `data-SOMETHING` attribute to assign data to an HTML element. We can later access that data in JavaScript using the element's `dataset` property.
- We use the `querySelectorAll` function to get an Node List (https://www.w3schools.com/js/js_htmldom_nodelist.asp) (similar to a Python list or a JavaScript array (https://www.w3schools.com/js/js_arrays.asp)) with all elements that match the query.
- The forEach (https://www.w3schools.com/jsref/jsref_foreach.asp) function in JavaScript takes in another function, and applies that function to each element in a list or array.

JavaScript Console

The console is a useful tool for testing out small bits of code and debugging. You can write and run JavaScript code in the console, which can be found by inspecting element in your web browser and then clicking `console`. (The exact process may change frome browser to browser.) One useful tool for debugging is printing to the console, which you can do using the `console.log` function. For example, in the `colors.html` page above, I can add the following line:


```
console.log(document.querySelectorAll('button'));
```

Which gives us this in the console:



Arrow Functions

In addition to the traditional function notation we've seen already, JavaScript now gives us the ability to use Arrow Functions (https://www.w3schools.com/js/js_arrow_function.asp) where we have an input (or parentheses when there's no input) followed by `=>` followed by some code to be run. For example, we can alter our script above to use an anonymous arrow function:

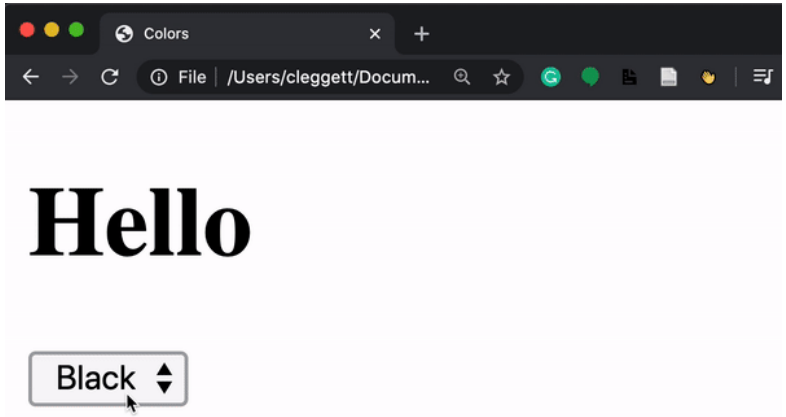
```
document.addEventListener('DOMContentLoaded', () => {
    document.querySelectorAll('button').forEach(button => {
        button.onclick = () => {
            document.querySelector("#hello").style.color = button.dataset.color;
        }
    });
});
```

We can also have named functions that use arrows, as in this rewriting of the count function:

```
count = () => {
    counter++;
    document.querySelector('h1').innerHTML = counter;

    if (counter % 10 === 0) {
        alert(`Count is now ${counter}`)
    }
}
```

To get an idea about some other events we can use, let's see how we can implement our color switcher using a dropdown menu instead of three separate buttons. We can detect changes in a `select` element using the `onchange` attribute. In JavaScript, this (https://www.w3schools.com/js/js_this.asp) is a keyword that changes based on the context in which it's used. In the case of an event handler, `this` refers to the object that triggered the event.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Colors</title>
        <script>
            document.addEventListener('DOMContentLoaded', function() {
                document.querySelector('select').onchange = function() {
                    document.querySelector('#hello').style.color = this.value;
                }
            });
        </script>
    </head>
    <body>
        <h1 id="hello">Hello</h1>
        <select>
            <option value="black">Black</option>
            <option value="red">Red</option>
            <option value="blue">Blue</option>
            <option value="green">Green</option>
        </select>

    </body>
</html>
```

There are many other events (https://www.w3schools.com/js/js_events.asp) we can detect in JavaScript including the common ones below:

- `onclick`
- `onmouseover`
- `onkeydown`
- `onkeyup`
- `onload`
- `onblur`
- ...

TODO List

To put together a few of the things we've learned in this lecture, let's work on making a TODO list entirely in JavaScript. We'll start by writing the HTML layout of the page. Notice below how we leave space for an unorderd list, but we dont yet add anything to it. Also notice that we add a link to `tasks.js` where we'll write our JavaScript.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Tasks</title>
        <script src="tasks.js"></script>
    </head>
    <body>
        <h1>Tasks</h1>
        <ul id="tasks"></ul>
        <form>
            <input id="task" placeholder = "New Task" type="text">
            <input id="submit" type="submit">
        </form>
    </body>
</html>
```

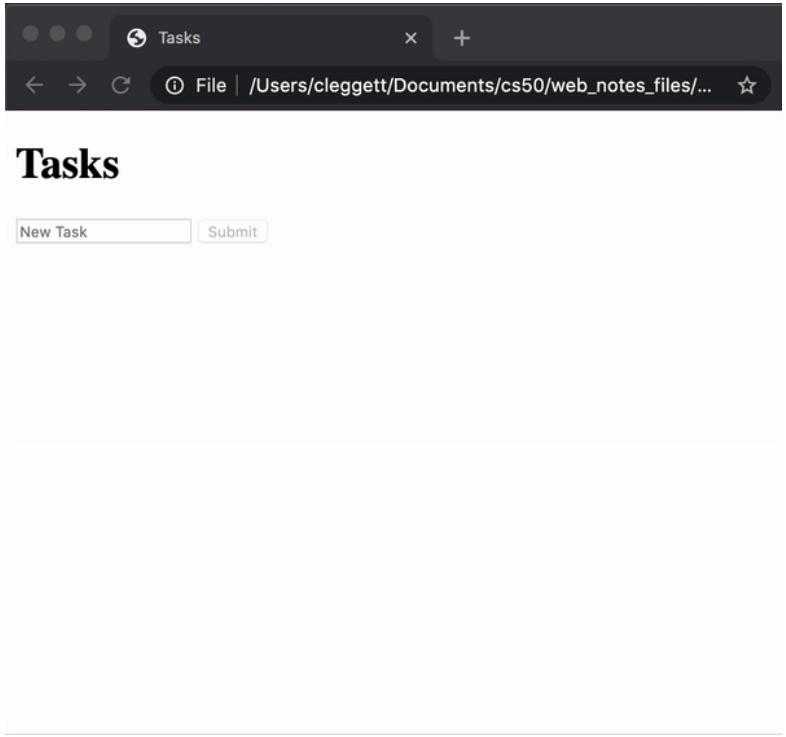
Now, here's our code which we can keep in `tasks.js`. A few notes on what you'll see below:

- This code is slightly different from the code in lecture. Here, we only query for our submit button and input task field once in the beginning and store those two values in the variables `submit` and `newTask`.
- We can enable/disable a button by setting its `disabled` attribute to `false` / `true`.
- In JavaScript, we use `.length` to find the length of objects such as strings and arrays.
- At the end of the script, we add the line `return false`. This prevents the default submission of the form which involves either reloading the current page or redirecting to a new one.
- In JavaScript, we can create HTML elements using the createElement (https://www.w3schools.com/jsref/met_document_createelement.asp) function. We can then add those elements to the DOM using the `append` function.

```
// Wait for page to load
document.addEventListener('DOMContentLoaded', function() {

    // Select the submit button and input to be used later
    const submit = document.querySelector('#submit');
    const newTask = document.querySelector('#task');

    // Disable submit button by default:
    submit.disabled = true;

    // Listen for input to be typed into the input field
    newTask.onkeyup = () => {
        if (newTask.value.length > 0) {
            submit.disabled = false;
        }
        else {
            submit.disabled = true;
        }
    }
```



```
    // Listen for submission of form
    document.querySelector('form').onsubmit = () => {

        // Find the task the user just submitted
        const task = newTask.value;

        // Create a list item for the new task and add the task to it
        const li = document.createElement('li');
        li.innerHTML = task;

        // Add new element to our unordered list:
        document.querySelector('#tasks').append(li);

        // Clear out input field:
        newTask.value = '';

        // Disable the submit button again:
        submit.disabled = true;

        // Stop form from submitting
        return false;
    }
});
```



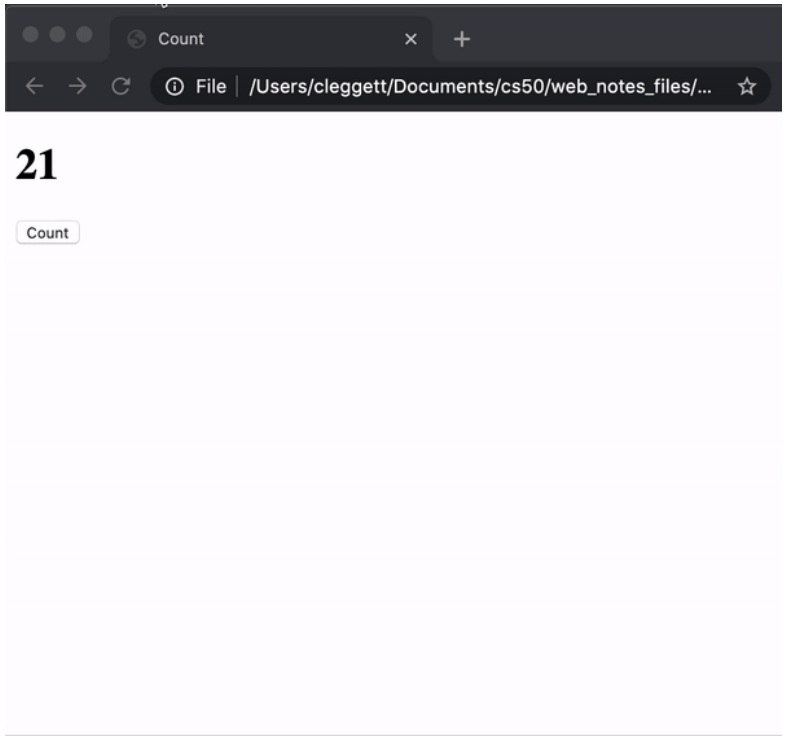
Intervals

In addition to specifying that functions run when an event is triggered, we can also set functions to run after a set amount of time. For example, let's return to our counter page's script, and add an interval so even if the user doesn't click anything, the counter increments every second. To do this, we use the setInterval (https://www.w3schools.com/jsref/met_win_setinterval.asp) function, which takes as argument a function to be run, and a time (in milliseconds) between function runs.

```
let counter = 0;

function count() {
    counter++;
    document.querySelector('h1').innerHTML = counter;
}

document.addEventListener('DOMContentLoaded', function() {
    document.querySelector('button').onclick = count;
```



```
    setInterval(count, 1000);
});
```



Local Storage

One thing to notice about all of our sites so far is that every time we reload the page, all of our information is lost. The heading color goes back to black, the counter goes back to 0, and all of the tasks are erased. Sometimes this is what we intend, but other time's we'll want to be able to store information that we can use when a user returns to the site.

One way we can do this is by using Local Storage (https://www.w3schools.com/jsref/prop_win_localstorage.asp), or storing information on the user's web browser that we can access later. This information is stored as a set of key-value pairs, almost like a Python dictionary. In order to use local storage, we'll employ two key functions:

- `localStorage.getItem(key)`: This function searches for an entry in local storage with a given key, and returns the value associated with that key.
- `localStorage.setItem(key, value)`: This function sets and entry in local storage, associating the key with a new vlaue.

Let's look at how we can use these new functions to update our counter! In the code below,

```
// Check if there is already a vlaue in local storage
if (!localStorage.getItem('counter')) {

    // If not, set the counter to 0 in local storage
    localStorage.setItem('counter', 0);
}

function count() {
    // Retrieve counter value from local storage
    let counter = localStorage.getItem('counter');

    // update counter
    counter++;
    document.querySelector('h1').innerHTML = counter;

    // Store counter in local storage
    localStorage.setItem('counter', counter);
}

document.addEventListener('DOMContentLoaded', function() {
    // Set heading to the current value inside local storage
    document.querySelector('h1').innerHTML = localStorage.getItem('counter');
    document.querySelector('button').onclick = count;
});
```

APIs

JavaScript Objects

A JavaScript Object (https://www.w3schools.com/js/js_objects.asp) is very similar to a Python dictionary, as it allows us to store key-value pairs. For example, I could create a JavaScript Object representing Harry Potter:

```
let person = {
    first: 'Harry',
    last: 'Potter'
};
```

I can then access or change parts of this object using either bracket or dot notation:


```
> person.first
< "Harry"
> person.last
< "Potter"
> person['first']
< "Harry"
> person['last']
< "Potter"
> person.first = 'Ron';
< "Ron"
> person.last = 'Weasley';
< "Weasley"
> person
< ▶ {first: "Ron", last: "Weasley"}
```

One way in which JavaScript Objects are really useful is in transferring data from one site to another, particularly when using APIs (https://www.mulesoft.com/resources/api/what-is-an-api)

An API, or Application Programming Interface, is a structured form communication between two different applications.

For example, we may want our application to get information from Google Maps, Amazon, or some weather service. We can do this by making calls to a service's API, which will return structured data to us, often in JSON (https://www.w3schools.com/js/js_json_intro.asp) (JavaScript Object Notation) form. For example, a flight in JSON form might look like this:

```
{
    "origin": "New York",
```

```
    "destination": "London",
    "duration": 415
}
```

The values within a JSON do not have to just be strings and numbers as in the example above. We can also store lists, or even other JavaScript Objects:

```
{
    "origin": {
        "city": "New York",
        "code": "JFK"
    },
    "destination": {
        "city": "London",
        "code": "LHR"
    },
    "duration": 415
}
```

Currency Exchange

To show how we can use APIs in our applications, let's work on building an application where we can find exchange rates between two currencies. Throughout the exercise, we'll be using the European Central Bank's Exchange Rate API (https://exchangeratesapi.io). By visiting their website, you'll see the API's documentation, which is generally a good place to start when you wish to use an API. We can test this api by visiting the URL: https://api.exchangeratesapi.io/latest?base=USD (https://api.exchangeratesapi.io/latest?base=USD). When you visit this page, you'll see the exchange rate between the U.S. Dollar and many other currencies, written in JSON form. You can also change the GET parameter in the URL from `USD` to any other currency code to change the rates you get.

Let's take a look at how to implement this API into an application by creating a new HTML file called `currency.html` and link it to a JavaScript file but leave the body empty:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Currency Exchange</title>
        <script src="currency.js"></script>
    </head>
    <body></body>
</html>
```

Now, we'll use something called AJAX (https://www.w3schools.com/js/js_ajax_intro.asp), or Asynchronous JavaScript And XML, which allows us to access information from external pages even after our page has loaded. In order to do this, we'll use the fetch (https://javascript.info/fetch)

function which will allow us to send an HTTP request. The `fetch` function returns a promise (https://web.dev/promises/). We won't talk about the details of what a promise is here, but we can think of it as a value that will come through at some point, but not necessarily right away. We deal with promises by giving them a `.then` attribute describing what should be done when we get a `response`. The code snippet below will log our response to the console.

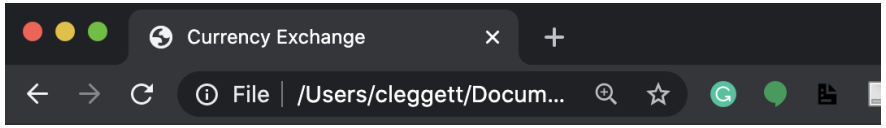
```
document.addEventListener('DOMContentLoaded', function() {
    // Send a GET request to the URL
    fetch('https://api.exchangeratesapi.io/latest?base=USD')
    // Put response into json form
    .then(response => response.json())
    .then(data => {
        // Log data to the console
        console.log(data);
    });
});
```

```
▼ Object i
    base: "USD"
    date: "2020-06-10"
  ▶ rates: {CAD: 1.3387252747, HKD: 7.7500659341, ISK: 132.…
  ▶ __proto__: Object
```

One important point about the above code is that the argument of `.then` is always a function. Although it seems we are creating the variables `response` and and `data`, those variables are just the parameters of two anonymous functions.

Rather than simply logging this data, we can use JavaScript to display a message to the screen, as in the code below:

```
document.addEventListener('DOMContentLoaded', function() {
    // Send a GET request to the URL
    fetch('https://api.exchangeratesapi.io/latest?base=USD')
    // Put response into json form
    .then(response => response.json())
    .then(data => {

        // Get rate from data
        const rate = data.rates.EUR;

        // Display message on the screen
        document.querySelector('body').innerHTML = `1 USD is equal to ${rate.toFixed(3
    });
});
```



1 USD is equal to 0.879 EUR.

Now, let's make the site a bit more interactive by allowing the user to choose which currency they would like to see. We'll start by altering our HTML to allow the user to input a currency:

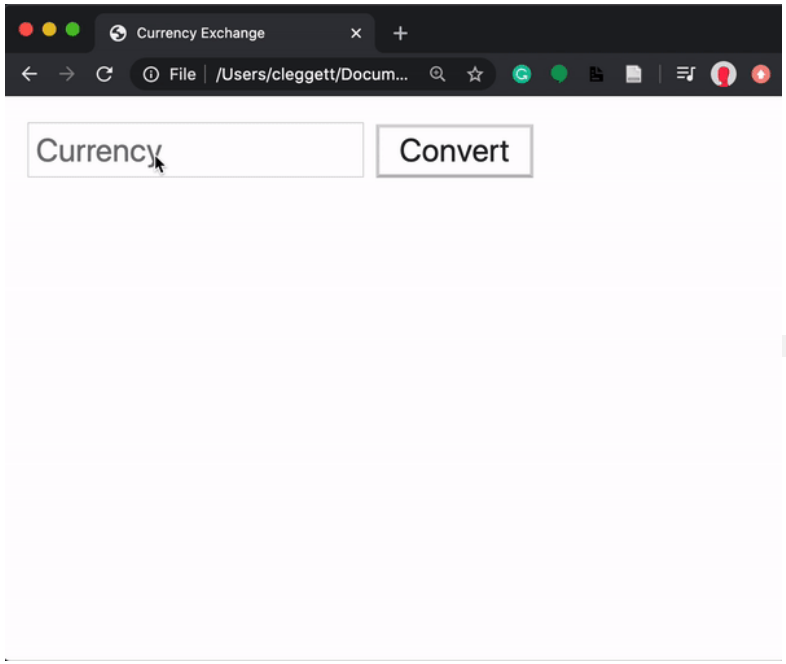
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Currency Exchange</title>
        <script src="currency.js"></script>
    </head>
    <body>
        <form>
            <input id="currency" placeholder="Currency" type="text">
            <input type="submit" value="Convert">
        </form>
        <div id="result"></div>
    </body>
</html>
```

Now, we'll make some changes to our JavaScript so it only changes when the form is submitted, and so it takes into account the user's input. We'll also add some error checking here:

```
document.addEventListener('DOMContentLoaded', function() {
    document.querySelector('form').onsubmit = function() {

        // Send a GET request to the URL
        fetch('https://api.exchangeratesapi.io/latest?base=USD')
        // Put response into json form
        .then(response => response.json())
        .then(data => {
            // Get currency from user input and convert to upper case
            const currency = document.querySelector('#currency').value.toUpperCase();

            // Get rate from data
            const rate = data.rates[currency];

            // Check if currency is valid:
```



```
            if (rate !== undefined) {
                // Display exchange on the screen
                document.querySelector('#result').innerHTML = `1 USD is equal to ${rat
            }
            else {
                // Display error on the screen
                document.querySelector('#result').innerHTML = 'Invalid Currency.';
            }
        })
        // Catch any errors and log them to the console
        .catch(error => {
            console.log('Error:', error);
        });
        // Prevent default submission
        return false;
    }
});
```



That's all for this lecture! Next time, we'll work on using JavaScript to create even more engaging user interfaces!

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 6

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, started using Django to create web applications, and learned how to use Django models to store information in our sites. We then introduced JavaScript and learned how to use it to make web pages more interactive.
- Today, we'll discuss common paradigms in User Interface design, using JavaScript and CSS to make our sites even more user friendly.

User Interfaces

A User Interface is how visitors to a web page interact with that page. Our goal as web developers is to make these interactions as pleasant as possible for the user, and there are many methods we can use to do this.

Single Page Applications

Previously, if we wanted a website with multiple pages, we would accomplish that using different routes in our Django application. Now, we have the ability to load just a single page and then use JavaScript to manipulate the DOM. One major advantage of doing this is that we only need to modify the part of the page that is actually changing. For example, if we have a Nav Bar that doesn't change based on your current page, we wouldn't want to have to re-render that Nav Bar every time we switch to a new part of the page.

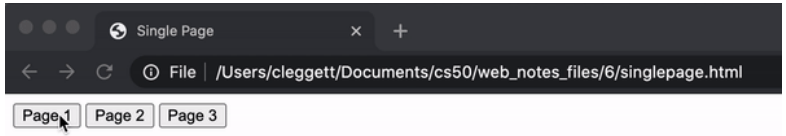
Let's look at an example of how we could simulate page switching in JavaScript:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Single Page</title>
        <style>
            div {
                display: none;
            }
        </style>
        <script src="singlepage.js"></script>
    </head>
    <body>
        <button data-page="page1">Page 1</button>
        <button data-page="page2">Page 2</button>
        <button data-page="page3">Page 3</button>
        <div id="page1">
            <h1>This is page 1</h1>
        </div>
        <div id="page2">
            <h1>This is page 2</h1>
        </div>
```

```
        <div id="page3">
            <h1>This is page 3</h1>
        </div>
    </body>
</html>
```

Notice in the HTML above that we have three buttons and three divs. At the moment, the divs contain only a small bit of text, but we could imagine each div containing the contents of one page on our site. Now, we'll add some JavaScript that allows us to use the buttons to toggle between pages.

```
// Shows one page and hides the other two
function showPage(page) {

    // Hide all of the divs:
    document.querySelectorAll('div').forEach(div => {
        div.style.display = 'none';
    });

    // Show the div provided in the argument
    document.querySelector(`#${page}`).style.display = 'block';
}

// Wait for page to loaded:
document.addEventListener('DOMContentLoaded', function() {

    // Select all buttons
    document.querySelectorAll('button').forEach(button => {

        // When a button is clicked, switch to that page
        button.onclick = function() {
            showPage(this.dataset.page);
        }
    })
});
```

In many cases, it will be inefficient to load the entire contents of every page when we first visit a site, so we will need to use a server to access new data. For example, when you visit a news site, it would take far too long for the site to load if it had to load every single article it has available when you first visit the page. We can avoid this problem using a strategy similar to the one we used while loading currency exchange rates in the previous lecture. This time, we'll take a look at using Django to send and receive information from our single page application. To show how this works, let's take a look at a simple Django application. It has two URL patterns in `urls.py`:

```
urlpatterns = [
    path("", views.index, name="index"),
    path("sections/<int:num>", views.section, name="section")
]
```

And two corresponding routes in `views.py`. Notice that the `section` route takes in an integer, and then returns a string of text based on that integer as an HTTP Response.

```
from django.http import Http404, HttpResponse
from django.shortcuts import render

# Create your views here.
def index(request):
```

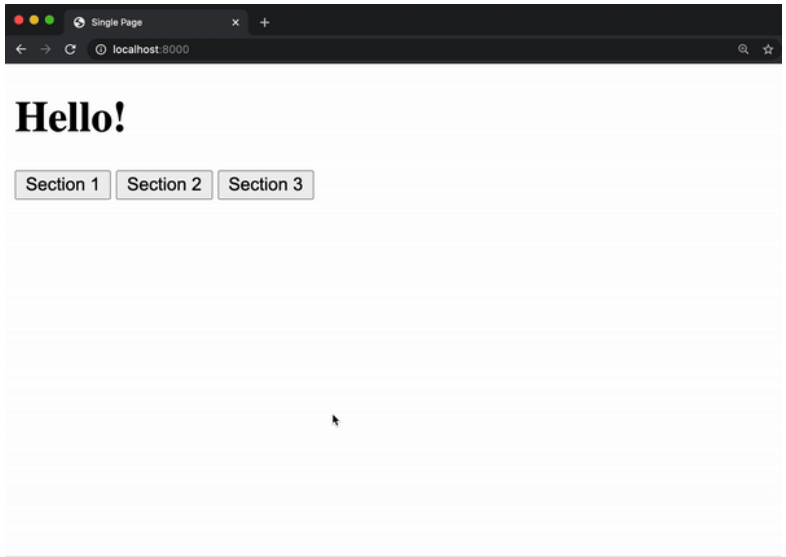
```
    return render(request, "singlepage/index.html")

# The texts are much longer in reality, but have
# been shortened here to save space
texts = ["Text 1", "Text 2", "Text 3"]

def section(request, num):
    if 1 <= num <= 3:
        return HttpResponse(texts[num - 1])
    else:
        raise Http404("No such section")
```

Now, within our `index.html` file, we'll take advantage of AJAX, which we learned about last lecture, to make a request to the server to gain the text of a particular section and display it on the screen:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Single Page</title>
        <style>
        </style>
        <script>

            // Shows given section
            function showSection(section) {

                // Find section text from server
                fetch(`/sections/${section}`)
                .then(response => response.text())
                .then(text => {
                    // Log text and display on page
                    console.log(text);
                    document.querySelector('#content').innerHTML = text;
                });
            }

            document.addEventListener('DOMContentLoaded', function() {
                // Add button functionality
                document.querySelectorAll('button').forEach(button => {
                    button.onclick = function() {
                        showSection(this.dataset.section);
                    };
                });
            });
        </script>
    </head>
    <body>
        <h1>Hello!</h1>
        <button data-section="1">Section 1</button>
        <button data-section="2">Section 2</button>
        <button data-section="3">Section 3</button>
        <div id="content">
        </div>
```



```
    </body>
</html>
```



Now, we've created a site where we can load new data from a server without reloading our entire HTML page!

One disadvantage of our site though is that the URL is now less informative. You'll notice in the video above that the URL remains the same even when we switch from section to section. We can solve this problem using the JavaScript History API (https://developer.mozilla.org/en-US/docs/Web/API/History_API). This API allows us to push information to our browser history and update the URL manually. Let's take a look at how we can use this API. Imagine we have a Django project identical to the previous one, but this time we wish to alter our script to be employ the history API:

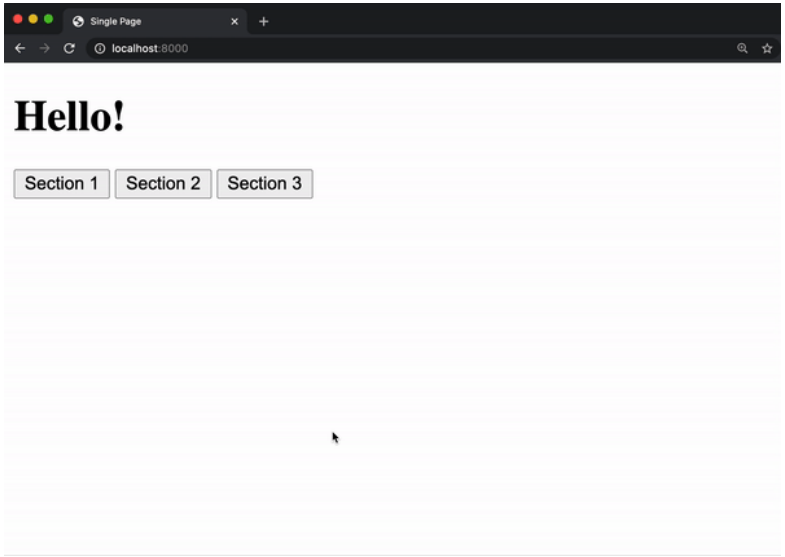
```
// When back arrow is clicked, show previous section
window.onpopstate = function(event) {
    console.log(event.state.section);
    showSection(event.state.section);
}

function showSection(section) {
    fetch(`/sections/${section}`)
    .then(response => response.text())
    .then(text => {
        console.log(text);
        document.querySelector('#content').innerHTML = text;
```

```
    });

}

document.addEventListener('DOMContentLoaded', function() {
    document.querySelectorAll('button').forEach(button => {
        button.onclick = function() {
            const section = this.dataset.section;

            // Add the current state to the history
            history.pushState({section: section}, "", `section${section}`);
            showSection(section);
        };
    });
});
```

In the `showSection` function above, we employ the `history.pushState` function. This function adds a new element to our browsing history based on three arguments:

1. Any data associated with the state.
2. A title parameter ignored by most web browsers
3. What should be displayed in the URL

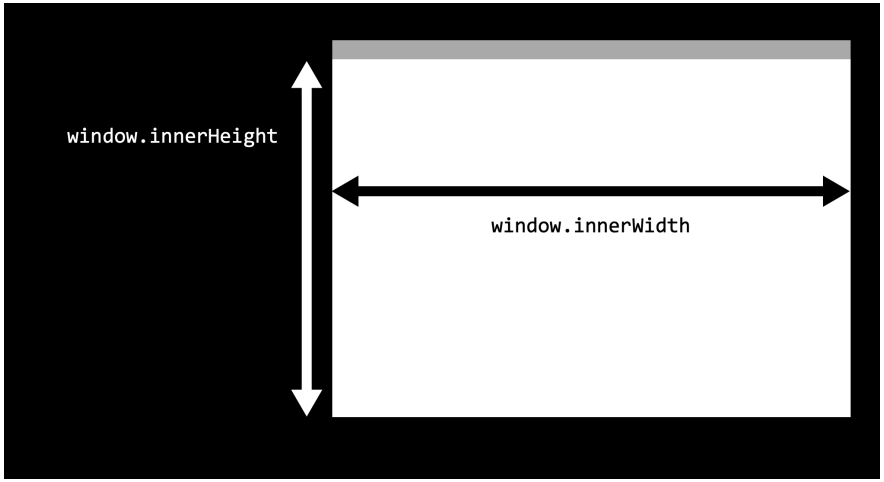
The other change we make in the above JavaScript is in setting the `onpopstate` parameter, which specifies what we should do when the user clicks the back arrow. In this case, we want to show the previous section when the button is pressed. Now, the site looks a little more user-friendly:



Scroll

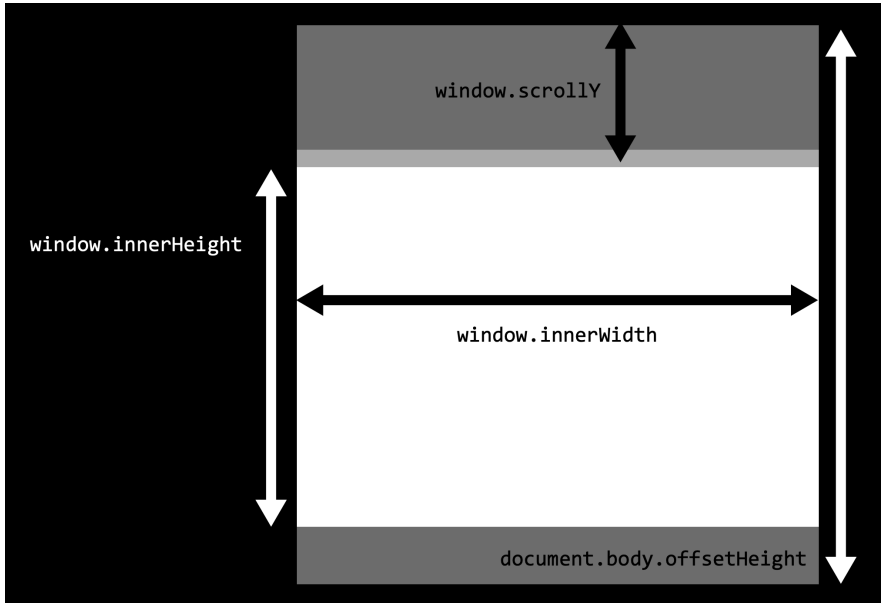
In order to update and access the browser history, we used an important JavaScript object known as the window (https://www.w3schools.com/js/js_window.asp). There are some other properties of the window that we can use to make our sites look nicer:

- `window.innerWidth`: Width of window in pixels
- `window.innerHeight`: Height of window in pixels



While the window represents what is currently visible to the user, the document (https://www.w3schools.com/js/js_htmldom_document.asp) refers to the entire web page, which is often much larger than the window, forcing the user to scroll up and down to see the page's contents. To work with our scrolling, we have access to other variables:

- `window.scrollY`: How many pixels we have scrolled from the top of the page
- `document.body.offsetHeight`: The height in pixels of the entire document.

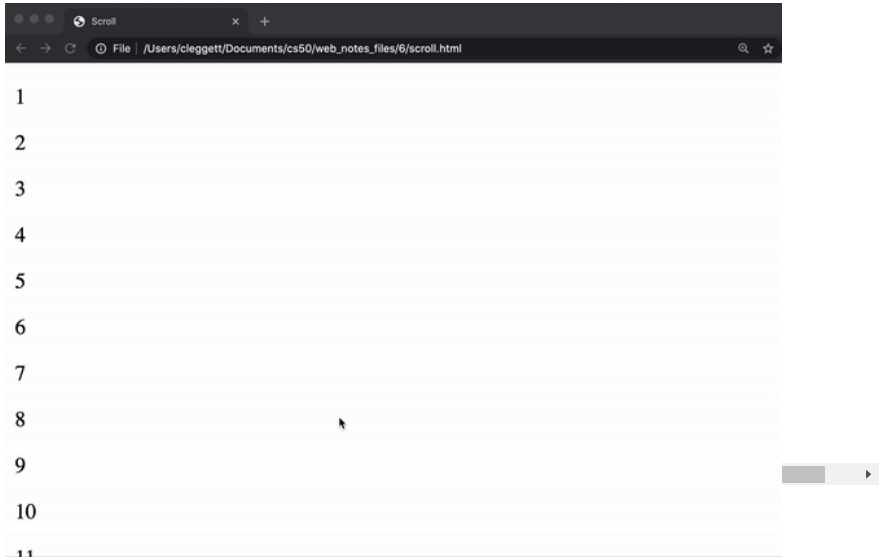


We can use these measures to determine whether or not the user has scrolled to the end of a page using the comparison `window.scrollY + window.innerHeight >= document.body.offsetHeight`. The following page, for example, will change the backgroud color to green when we reach the bottom of a page:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Scroll</title>
        <script>

            // Event listener for scrolling
            window.onscroll = () => {

                // Check if we're at the bottom
                if (window.innerHeight + window.scrollY >= document.body.offsetHeight) {

                    // Change color to green
                    document.querySelector('body').style.background = 'green';
                } else {

                    // Change color to white
                    document.querySelector('body').style.background = 'white';
                }

            };
```



```
        </script>
    </head>
    <body>
        <p>1</p>
        <p>2</p>
        <!-- More paragraphs left out to save space -->
        <p>99</p>
        <p>100</p>
    </body>
</html>
```



Infinite Scroll

Changing the background color at the end of the page probably isn't all that useful, but we may want to detect that we're at the end of the page if we want to implement **infinite scroll**. For example, if you're on a social media site, you don't want to have to load all posts at once, you might want to load the first ten, and then when the user reaches the bottom, load the next ten. Let's take a look at a Django application that could do this. This app has two paths in `urls.py`

```
urlpatterns = [
    path("", views.index, name="index"),
    path("posts", views.posts, name="posts")
]
```

And two corresponding views in `views.py`:

```
import time

from django.http import JsonResponse
from django.shortcuts import render

# Create your views here.
def index(request):
    return render(request, "posts/index.html")

def posts(request):

    # Get start and end points
    start = int(request.GET.get("start") or 0)
    end = int(request.GET.get("end") or (start + 9))

    # Generate list of posts
    data = []
    for i in range(start, end + 1):
        data.append(f"Post #{i}")

    # Artificially delay speed of response
    time.sleep(1)

    # Return list of posts
    return JsonResponse({
        "posts": data
    })
```

Notice that the `posts` view requires two arguments: a `start` point and an `end` point. In this view, we've created our own **API**, which we can test out by visiting the url `localhost:8000/posts?start=10&end=15`, which returns the following JSON:

```
{
    "posts": [
        "Post #10",
        "Post #11",
        "Post #12",
        "Post #13",
        "Post #14",
        "Post #15"
    ]
}
```

Now, in the `index.html` template that the site loads, we start out with only an empty `div` in the body and some styling. Notice that we load our static files at the beginning, and then we reference a JavaScript file within our `static` folder.

```
{% load static %}
```



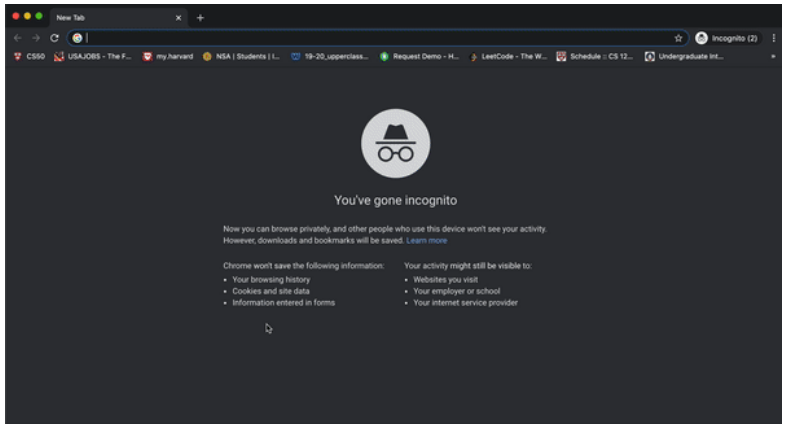
```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
        <style>
            .post {
                background-color: #77dd11;
                padding: 20px;
                margin: 10px;
            }

            body {
                padding-bottom: 50px;
            }
        </style>
        <script scr="{% static 'posts/script.js' %}"></script>
    </head>
    <body>
        <div id="posts">
        </div>
    </body>
</html>
```

Now with JavaScript, we'll wait until a user scrolls to the end of the page and then load more posts using our API:

```
// Start with first post
let counter = 1;

// Load posts 20 at a time
const quantity = 20;

// When DOM loads, render the first 20 posts
document.addEventListener('DOMContentLoaded', load);

// If scrolled to bottom, load the next 20 posts
window.onscroll = () => {
    if (window.innerHeight + window.scrollY >= document.body.offsetHeight) {
        load();
    }
};

// Load next set of posts
function load() {

    // Set start and end post numbers, and update counter
    const start = counter;
    const end = start + quantity - 1;
    counter = end + 1;

    // Get new posts and add posts
    fetch(`/posts?start=${start}&end=${end}`)
    .then(response => response.json())
```

```
    .then(data => {
        data.posts.forEach(add_post);
    })
};

// Add a new post with given contents to DOM
function add_post(contents) {

    // Create new post
    const post = document.createElement('div');
    post.className = 'post';
    post.innerHTML = contents;

    // Add post to DOM
    document.querySelector('#posts').append(post);
};
```

Now, we've created a site with infinite scroll!



Animation

Another way we can make our sites a bit more interesting is by adding some animation to them. It turns out that in addition to providing styling, CSS makes it easy for us to animate HTML elements.

To create an animation in CSS, we use the format below, where the animation specifics can include starting and ending styles (`to` and `from`) or styles at different stages in the duration (anywhere from `0%` to `100%`). For example:

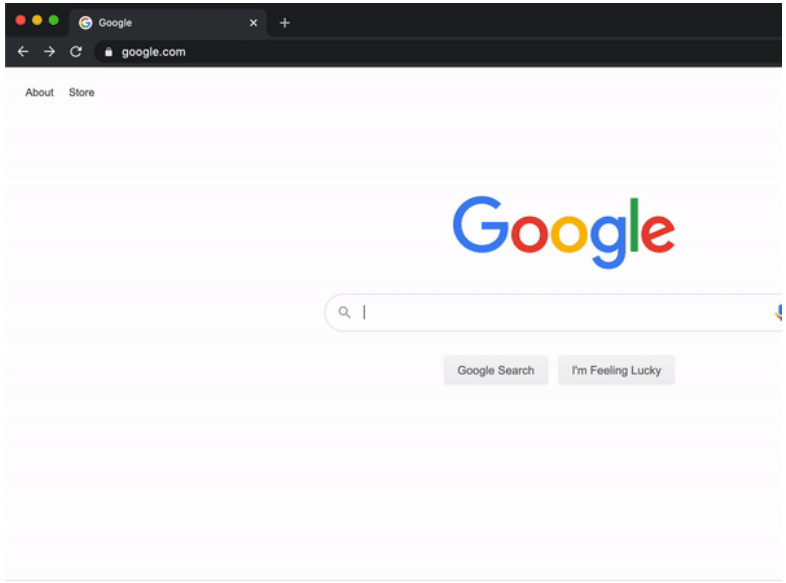
```
@keyframes animation_name {
    from {
        /* Some styling for the start */
    }

    to {
        /* Some styling for the end */
    }
}
```


or:

```
@keyframes animation_name {
    0% {
        /* Some styling for the start */
    }

    75% {
        /* Some styling after 3/4 of animation */
    }

    100% {
        /* Some styling for the end */
    }
}
```

Then, to apply an animation to an element, we include the `animation-name`, the `animation-duration` (in seconds), and the `animation-fill-mode` (typically `forwards`). For example, here's a page where a title grows when we first enter the page:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Animate</title>
        <style>
            @keyframes grow {
                from {
                    font-size: 20px;
                }
                to {
                    font-size: 100px;
                }
            }

            h1 {
                animation-name: grow;
                animation-duration: 2s;
                animation-fill-mode: forwards;
            }
        </style>
    </head>
    <body>
        <h1>Welcome!</h1>
    </body>
</html>
```

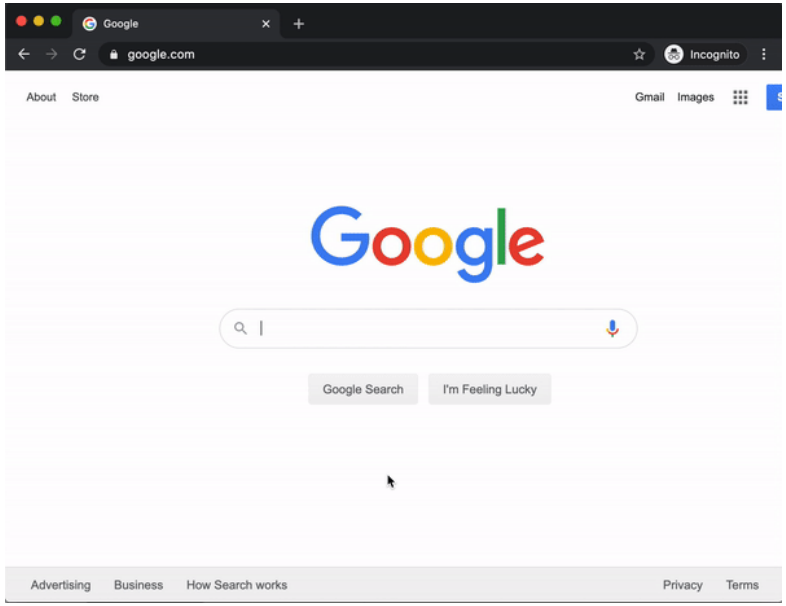


We can do more than just manipulate size: the below example shows how we can change the position of a heading just by changing a few lines:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Animate</title>
        <style>
            @keyframes move {
                from {
                    left: 0%;
                }
                to {
                    left: 50%;
                }
            }

            h1 {
                position: relative;
                animation-name: move;
                animation-duration: 2s;
                animation-fill-mode: forwards;
            }
        </style>
    </head>
    <body>
        <h1>Welcome!</h1>
```

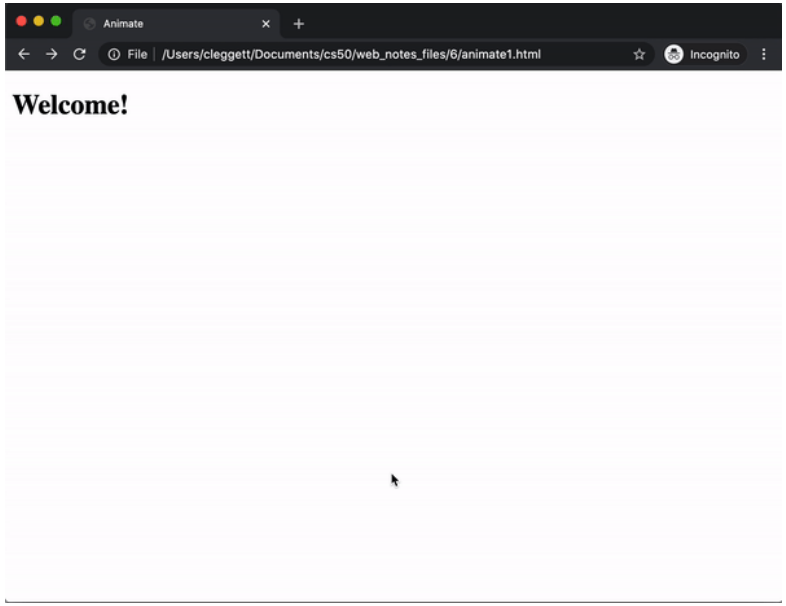


```
    </body>
</html>
```



Now, let's look at setting some intermediate CSS properties as well. We can specify the style at any percentage of the way through an animation. In the below example we'll move the title from left to right, and then back to left by altering only the animation from above

```
@keyframes move {
    0% {
        left: 0%;
    }
    50% {
        left: 50%;
    }
    100% {
        left: 0%;
    }
}
```



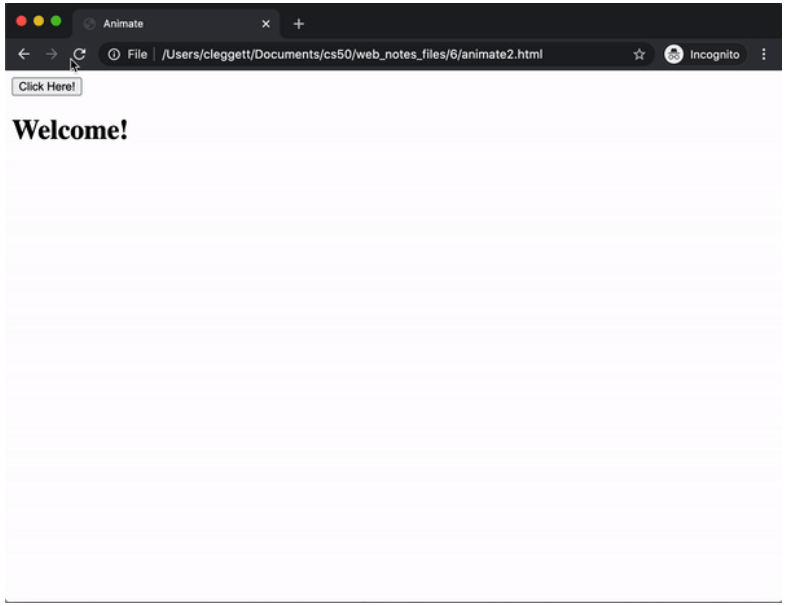
If we want to repeat an animation multiple times, we can change the `animation-iteration-count` to a number higher than one (or even `infinite` for endless animation). There are many animation properties (https://www.w3schools.com/cssref/css3_pr_animation.asp) that we can set in order to change different aspects of our animation.

In addition to CSS, we can use JavaScript to further control our animations. Let's use our moving header example (with infinite repetition) to show how we can create a button that starts and stops the animation. Assuming we already have an animation, button, and heading, we can add the following script to start and pause the animation:

```
document.addEventListener('DOMContentLoaded', function() {

    // Find heading
    const h1 = document.querySelector('h1');

    // Pause Animation by default
    h1.style.animationPlayState = 'paused';

    // Wait for button to be clicked
    document.querySelector('button').onclick = () => {

        // If animation is currently paused, begin playing it
        if (h1.style.animationPlayState == 'paused') {
            h1.style.animationPlayState = 'running';
        }
```



```
        // Otherwise, pause the animation
        else {
            h1.style.animationPlayState = 'paused';
        }
    }

})
```



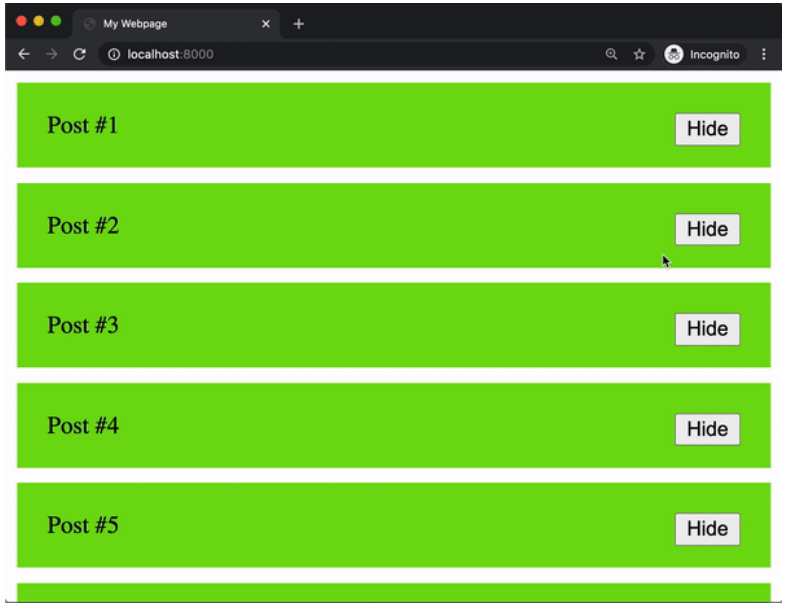
Now, let's look at how we can apply our new knowledge of animations to the posts page we made earlier. Specifically, let's say we want the ability to hide posts once we're done reading them. Let's imagine a Django project identical to the one we just created, but with some slightly different HTML and JavaScript. The first change we'll make is to the `add_post` function, this time also adding a button to the right side of the post:

```
// Add a new post with given contents to DOM
function add_post(contents) {

    // Create new post
    const post = document.createElement('div');
    post.className = 'post';
    post.innerHTML = `${contents} <button class="hide">Hide</button>`;

    // Add post to DOM
```

```
    document.querySelector('#posts').append(post);
};
```

Now, we'll work on hiding a post when the `hide` button is clicked. To do this, we'll add an event listener that is triggered whenever a user clicks anywhere on the page. We then write a function that takes in the `event` as an argument, which is useful because we can use the `event.target` attribute to access what was clicked on. We can also use the `parentElement` class to find the parent of a given element in the DOM.

```
// If hide button is clicked, delete the post
document.addEventListener('click', event => {

    // Find what was clicked on
    const element = event.target;

    // Check if the user clicked on a hide button
    if (element.className === 'hide') {
        element.parentElement.remove()
    }

});
```



We can now see that we've implemented the hide button, but it doesn't look as nice as it possible could. Maybe we want to have the post fade away and shrink before we remove it. In order to do

this, we'll first create a CSS animation. The animation below will spend 75% of its time changing the `opacity` from 1 to 0, which esentially makes the post fade out slowly. It then spends the rest of the time moving all of its `height`-related attributes to 0, effectively shrinking the post to nothing.

```
@keyframes hide {
    0% {
        opacity: 1;
        height: 100%;
        line-height: 100%;
        padding: 20px;
        margin-bottom: 10px;
    }
    75% {
        opacity: 0;
        height: 100%;
        line-height: 100%;
        padding: 20px;
        margin-bottom: 10px;
    }
    100% {
        opacity: 0;
        height: 0px;
        line-height: 0px;
        padding: 0px;
        margin-bottom: 0px;
    }
}
```

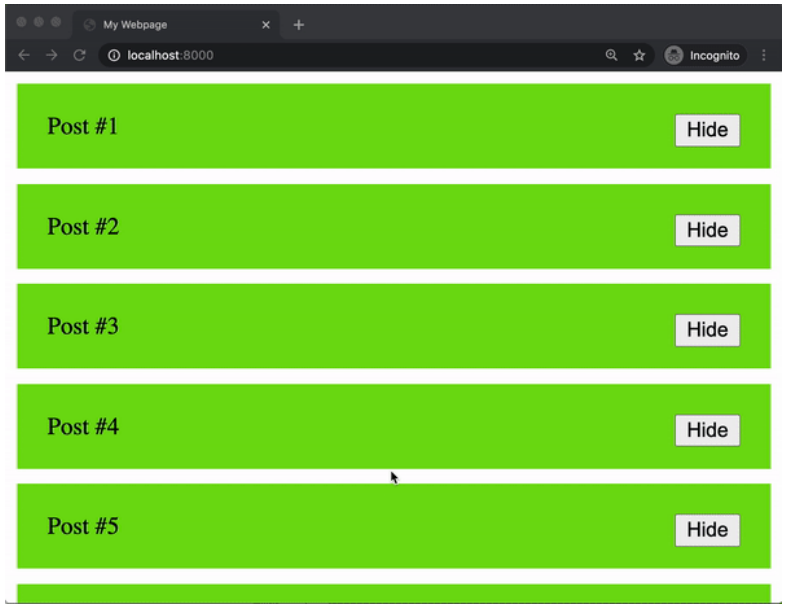
Next, we would add this animation to our post's CSS. Notice that we initially set the `animation-play-state` to `paused`, meaning the post will not be hidden by default.

```
.post {
    background-color: #77dd11;
    padding: 20px;
    margin-bottom: 10px;
    animation-name: hide;
    animation-duration: 2s;
    animation-fill-mode: forwards;
    animation-play-state: paused;
}
```

Finally, we want to be able to start the animation once the `hide` button has been clicked, and then remove the post. We can do this by editing our JavaScript from above:

```
// If hide button is clicked, delete the post
document.addEventListener('click', event => {

    // Find what was clicked on
    const element = event.target;
```

```
    // Check if the user clicked on a hide button
    if (element.className === 'hide') {
        element.parentElement.style.animationPlayState = 'running';
        element.parentElement.addEventListener('animationend', () => {
            element.parentElement.remove();
        });
    }

});
```



As you can see above, the hide functionality now looks a lot nicer!

React

At this point, you can imagine how much JavaScript code would have to go into a more complicated website. We can mitigate how much code we actually need to write by employing a JavaScript framework, just as we employed Bootstrap as a CSS framework to cut down on the amount of CSS we actually had to write. One of the most popular JavaScript frameworks is a library called React (https://reactjs.org/).

So far in this course, we've been using **imperative programming** methods, where we give the computer a set of statements to execute. For example, to update the counter in an HTML page we

might have have code that looks like this:

View:

```
<h1>0</h1>
```

Logic:

```
let num = parseInt(document.querySelector("h1").innerHTML);
num += 1;
document.querySelector("h1").innerHTML = num;
```

React allows us to use **declarative programming**, which will allow us to simply write code explaining *what* we wish to display and not worry about *how* we're displaying it. In React, a counter might look a bit more like this:

View:

```
<h1>{num}</h1>
```

Logic:

```
num += 1;
```

The React framework is built around the idea of components, each of which can have an underlying state. A component would be something you can see on a web page like a post or a navigation bar, and a state is a set of variables associated with that component. The beauty of React is that when the state changes, React will automatically change the DOM accordingly.

There are a number of ways to use React, (including the popular create-react-app (https://reactjs.org/docs/create-a-new-react-app.html) command published by Facebook) but today we'll focus on getting started directly in an HTML file. To do this, we'll have to import three JavaScript Packages:

- `React`: Defines components and their behavior
- `ReactDOM`: Takes React components and inserts them into the DOM
- `Babel`: Translates from JSX (https://reactjs.org/docs/introducing-jsx.html), the language in which we'll write in React, to plain JavaScript that our browsers can interpret. JSX is very similar to JavaScript, but with some additional features, including the ability to represent HTML inside of our code.

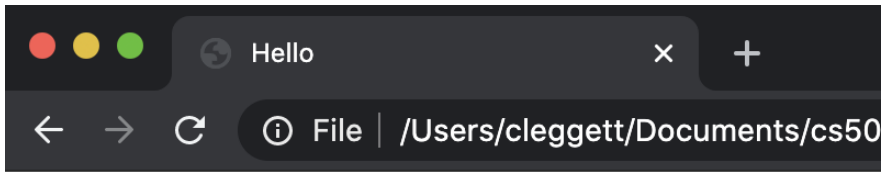
Let's dive in and create our first React application!

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <script src="https://unpkg.com/react@17/umd/react.production.min.js" crossorig
        <script src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js" c
        <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
        <title>Hello</title>
    </head>
    <body>
        <div id="app"></div>

        <script type="text/babel">
            function App() {
                return (
                    <div>
                        Hello!
                    </div>
                );
            }

            ReactDOM.render(<App />, document.querySelector("#app"));
        </script>
    </body>
</html>
```

Since this is our first React app, let's take a detailed look at what each part of this code is doing:

- In the three lines above the title, we import the latest versions of React, ReactDOM, and Babel.
- In the body, we include a single `div` with an `id` of `app`. We almost always want to leave this empty, and fill it in our react code below.
- We include a script tag where we specify that `type="text/babel"`. This signals to the browser that the following script needs to be translated using Babel.
- Next, we create a component called `App`. Components in React can be represented by JavaScript functions.
- Our component returns what we would like to render to the DOM. In this case, we simply return `<div>Hello!</div>`.
- The last line of our script employs the `ReactDOM.render` function, which takes two arguments:
    1. A component to render
    2. An element in the DOM inside of which the component should be rendered

Now that we understant what the code is doing, we can take a look at the resulting webpage:



Welcome!

Hello!

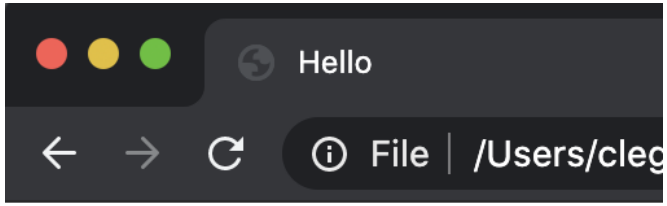
One useful feature of React is the ability to render components within other components. To demonstrate this, let's create another component called `Hello`:

```
function Hello(props) {
    return (
        <h1>Hello</h1>
    );
}
```

And now, let's render three `Hello` components inside of our `App` component:

```
function App() {
    return (
        <div>
            <Hello />
            <Hello />
            <Hello />
        </div>
    );
}
```

This gives us a page that looks like:



Hello

Hello

Hello

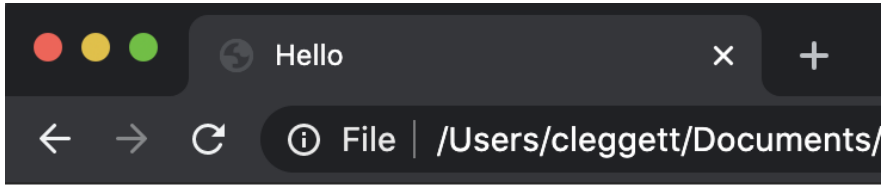
So far, the components haven't been all that interesting, as they are all exactly the same. We can make these components more flexible by adding additional properties (**props** in React terms) to them. For example, let's say we wish to say hello to three different people. We can provide those people's names in a method that looks similar to HTML attributes:

```
function App() {
    return (
        <div>
            <Hello name="Harry" />
            <Hello name="Ron" />
            <Hello name="Hermione" />
        </div>
    );
}
```

We can then access those props using `props.PROP_NAME`. We can then insert this into our JSX using curly braces:


```
function Hello(props) {
    return (
        <h1>Hello, {props.name}!</h1>
    );
}
```

Now, our page displays the three names!



Hello, Harry!

Hello, Ron!

Hello, Hermione!

Now, let's see how we can use React to re-implement the counter page we built when first working with JavaScript. Our overall structure will remain the same, but inside of our `App` component, we'll use React's `useState` hook to add state to our component. The argument to `useState` is the initial value of the state, which we'll set to `0`. The function returns both a variable representing the state and a function that allows us to update the state.

```
const [count, setCount] = React.useState(0);
```

Now, we can work on what the function will render, where we'll specify a header and a button. We'll also add an event listener for when the button is clicked, which React handles using the

`onClick` attribute:

```
return (
    <div>
        <h1>{count}</h1>
        <button onClick={updateCount}>Count</button>
    </div>
);
```

Finally, let's define the `updateCount` function. To do this, we'll use the `setCount` function, which can take as argument a new value for the state.

```
function updateCount() {
    setCount(count + 1);
}
```

Now we have a functioning counter site!

Addition

Now that we have a feel for the React framework, let's work on using what we've learned to build a game-like site where users will solve addition problems. We'll begin by creating a new file with the same setup as our other React pages. To start building this application, let's think about what we might want to keep track of in the state. We should include anything that we think might change while a user is on our page. Our state might include:

- `num1`: The first number to be added
- `num2`: The second number to be added
- `response`: What the user has typed in

- `score`: How many questions the user has answered correctly.

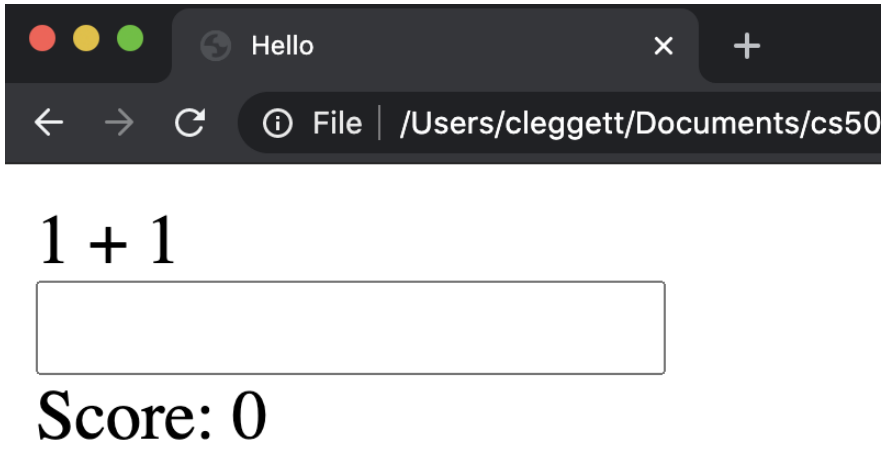
Now, our state can be a JavaScript object that includes all of this information:

```
const [state, setState] = React.useState({
    num1: 1,
    num2: 1,
    response: "",
    score: 0
});
```

Now, using the values in the state, we can render a basic user interface.

```
return (
    <div>
        <div>{state.num1} + {state.num2}</div>
        <input value={state.response} />
        <div>Score: {state.score}</div>
    </div>
);
```

Now, the basic layout of the site looks like this:



At this point, the user cannot type anything in the input box because its value is fixed as `state.response` which is currently the empty string. To fix this, let's add an `onChange` attribute to the input element, and set it equal to a function called `updateResponse`

```
onChange={updateResponse}
```

Now, we'll have to define the `updateResposne` function, which takes in the event that triggered the function, and sets the `response` to the current value of the input. This function allows the user to type, and stores whatever has been typed in the `state`.

```
function updateResponse(event) {
    setState({
        ...state,
        response: event.target.value
    });
}
```

Now, let's add the ability for a user to submit a problem. We'll first add another event listener and link it to a function we'll write next:

```
onKeyPress={inputKeyPress}
```

Now, we'll define the `inputKeyPress` function. In this function, we'll first check whether the `Enter` key was pressed, and then check to see if the answer is correct. When the user is correct, we want to increase the score by 1, choose random numbers for the next problem, and clear the response. If the answer is incorrect, we want to decrease the score by 1 and clear the response.

```
function inputKeyPress(event) {
    if (event.key === "Enter") {
        const answer = parseInt(state.response);
        if (answer === state.num1 + state.num2) {
            // User got question right
            setState({
                ...state,
                score: state.score + 1,
                response: "",
                num1: Math.ceil(Math.random() * 10),
                num2: Math.ceil(Math.random() * 10)
            });
        } else {
            // User got question wrong
            setState({
                ...state,
```

```
                score: state.score - 1,
                response: ""
            })
        }
    }
}
```

To put some finishing touches on the application, let's add some style to the page. We'll center everything in the app, and then make the problem larger by adding an `id` of `problem` to the div containing the problem, and then adding the following CSS to a style tag:

```
#app {
    text-align: center;
    font-family: sans-serif;
}

#problem {
    font-size: 72px;
}
```

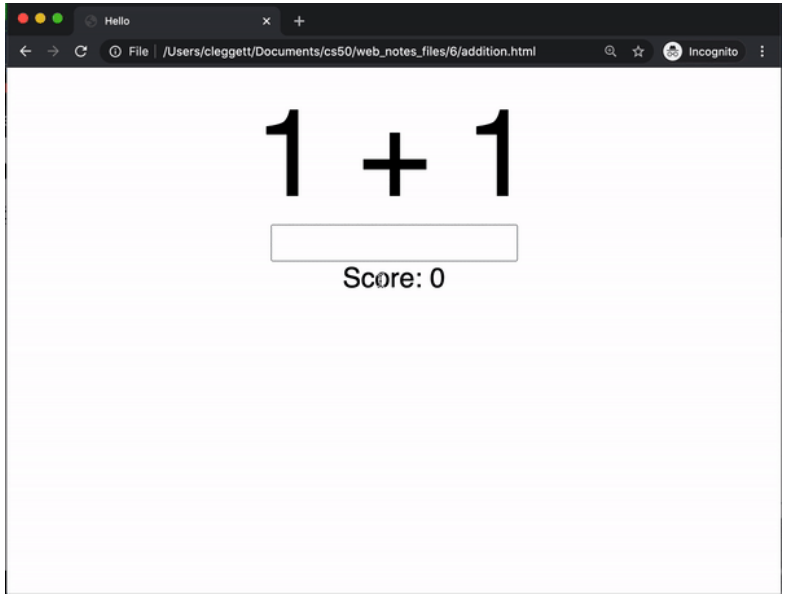
Finally, let's add the ability to win the game after gaining 10 points. To do this, we'll add a condition to the `render` funciton, returning something completely different once we have 10 points:

```
if (state.score === 10) {
    return (
        <div id="winner">You won!</div>
    );
}
```

To make the win more exciting, we'll add some style to the alternative div as well:

```
#winner {
    font-size: 72px;
    color: green;
}
```

Now, let's take a look at our application!



That's all for lecture today! Next time, we'll talk about some best practices for building larger web applications.

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 7

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, started using Django to create web applications, and learned how to use Django models to store information in our sites. We then introduced JavaScript and learned how to use it to make web pages more interactive, and talked about using animation and React to further improve our User Interfaces.
- Today, we'll learn about best practices when if comes to working on and launching larger projects.

Testing

One important part of the software development process is the act of **Testing** the code we've written to make sure everything runs as we expect it to. In this lecture, we'll discuss several ways that we can improve the way we test our code.

Assert

One of the simplest ways we can run tests in Python is by using the `assert` command. This command is followed by some expression that should be `True`. If the expression is `True`, nothing will happen, and if it is `False`, an exception will be thrown. Let's look at how we could incorporate command to test the `square` function we wrote when first learning Python. When the function is written correctly, nothing happens as the `assert` is `True`

```
def square(x):
    return x * x

assert square(10) == 100

""" Output:

"""
```

And then when it is written incorrectly, an exception is thrown.

```
def square(x):
    return x + x

assert square(10) == 100

""" Output:
Traceback (most recent call last):
  File "assert.py", line 4, in <module>
```

```
    assert square(10) == 100
AssertionError
"""
```

Test-Driven Development

As you begin building larger projects, you may want to consider using **test-driven development**, a development style where every time you fix a bug, you add a test that checks for that bug to a growing set of tests that are run every time you make changes. This will help you to make sure that additional features you add to a project don't interfere with your existing features.

Now, let's look at a slightly more complex function, and think about how writing tests can help us to find errors. We'll now write a function called `is_prime` that returns `True` if and only if its input is prime:

```
import math

def is_prime(n):

    # We know numbers less than 2 are not prime
    if n < 2:
        return False

    # Checking factors up to sqrt(n)
    for i in range(2, int(math.sqrt(n))):

        # If i is a factor, return false
        if n % i == 0:
            return False

    # If no factors were found, return true
    return True
```

Now, let's take a look at a function we've written to test our `prime` function:

```
from prime import is_prime

def test_prime(n, expected):
    if is_prime(n) != expected:
        print(f"ERROR on is_prime({n}), expected {expected}")
```

At this point, we can go into our python interpreter and test out some values:

```
>>> test_prime(5, True)
>>> test_prime(10, False)
>>> test_prime(25, False)
ERROR on is_prime(25), expected False
```


We can see from the output above that 5 and 10 were correctly identified as prime and not prime, but 25 was incorrectly identified as prime, so there must be something wrong with our function. Before we look into what is wrong with our function though, let's look at a way to automate our testing. One way we can do this is by creating a **shell script**, or some script that can be run inside our terminal. These files require a `.sh` extension, so our file will be called `tests0.sh`. Each of the lines below consists of

1. A `python3` to specify the Python version we're running
2. A `-c` to indicate that we wish to run a command
3. A command to run in string format

```
python3 -c "from tests0 import test_prime; test_prime(1, False)"
python3 -c "from tests0 import test_prime; test_prime(2, True)"
python3 -c "from tests0 import test_prime; test_prime(8, False)"
python3 -c "from tests0 import test_prime; test_prime(11, True)"
python3 -c "from tests0 import test_prime; test_prime(25, False)"
python3 -c "from tests0 import test_prime; test_prime(28, False)"
```

Now we can run these commands by running `./tests0.sh` in our terminal, giving us this result:

```
ERROR on is_prime(8), expected False
ERROR on is_prime(25), expected False
```

Unit Testing

Even though we were able to run tests automatically using the above method, we still might want to avoid having to write out each of those tests. Thankfully, we can use the Python `unittest` library to make this process a little bit easier. Let's take a look at what a testing program might look like for our `is_prime` function.

```
# Import the unittest library and our function
import unittest
from prime import is_prime

# A class containing all of our tests
class Tests(unittest.TestCase):

    def test_1(self):
        """Check that 1 is not prime."""
        self.assertFalse(is_prime(1))

    def test_2(self):
        """Check that 2 is prime."""
        self.assertTrue(is_prime(2))

    def test_8(self):
```

```
        """Check that 8 is not prime."""
        self.assertFalse(is_prime(8))

    def test_11(self):
        """Check that 11 is prime."""
        self.assertTrue(is_prime(11))

    def test_25(self):
        """Check that 25 is not prime."""
        self.assertFalse(is_prime(25))

    def test_28(self):
        """Check that 28 is not prime."""
        self.assertFalse(is_prime(28))


# Run each of the testing functions
if __name__ == "__main__":
    unittest.main()
```

Notice that each of the functions within our `Tests` class followed a pattern:

- The name of the functions begin with `test_`. This is necessary for the functions to be run automatically with the call to `unittest.main()`.
- Each test takes in the `self` argument. This is standard when writing methods within Python classes.
- The first line of each function contains a **docstring** surrounded by three quotation marks. These are not just for the code's readability. When the tests are run, the comment will be displayed as a discription of the test if it fails.
- The next line of each of the functions contained an assertion in the form `self.assertSOMETHING`. There are many different assertions you can make including `assertTrue`, `assertFalse`, `assertEqual`, and `assertGreater`. You can find these ones and more by checking out the documentation (https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertEqual).

Now, let's check out the results of these tests:

```
...F.F
======================================================================
FAIL: test_25 (__main__.Tests)
Check that 25 is not prime.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests1.py", line 26, in test_25
    self.assertFalse(is_prime(25))
AssertionError: True is not false

======================================================================
FAIL: test_8 (__main__.Tests)
```



```
Check that 8 is not prime.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests1.py", line 18, in test_8
    self.assertFalse(is_prime(8))
AssertionError: True is not false

----------------------------------------------------------------------
Ran 6 tests in 0.001s

FAILED (failures=2)
```

After running the tests, `unittest` provides us with some useful information about what it found. In the first line, it gives us a series of `.`s for successes and `F`s for failures in the order our tests were written.

```
...F.F
```

Next, for each of the tests that failed, we are then given the name of the function that failed:

```
FAIL: test_25 (__main__.Tests)
```

the descriptive comment we provided earlier:

```
Check that 25 is not prime.
```

And a traceback for the exception:

```
Traceback (most recent call last):
  File "tests1.py", line 26, in test_25
    self.assertFalse(is_prime(25))
AssertionError: True is not false
```

And finally, we are given a run through of how many tests were run, how much time they took, and how many failed:

```
Ran 6 tests in 0.001s

FAILED (failures=2)
```

Now let's take a look at fixing the bug in our function. It turns out that we need to test one additional number in our `for` loop. For example, when `n` is `25`, the square root is `5`, but when that is one argument in the `range` function, the `for` loop terminates at the number `4`. Therefore, we can simply change the header of our `for` loop to:

```
for i in range(2, int(math.sqrt(n)) + 1):
```

Now, when we run the tests again using our unit tests, we get the following output, indicating that our change fixed the bug.

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

These automated tests will become even more useful as you work to optimize this function. For example, you might want to use the fact that you don't need to check all integers as factors, just smaller primes (if a number is not divisible by 3, it is also not divisible by 6, 9, 12, ...), or you may want to use more advanced probabilistic primality tests such as the Fermat (https://en.wikipedia.org/wiki/Fermat_primality_test) and Miller-Rabin (https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test) primality tests. Whenever you make changes to improve this function, you'll want the ability to easily run your unit tests again to make sure your function is still correct.

Django Testing

Now, let's look at how we can apply the ideas of automated testing when creating Django applications. While working with this, we'll be using the `flights` project we created when we first learned about Django models. We're first going to add a method to our `Flight` model that verifies that a flight is valid by checking for two conditions:

1. The origin is not the same as the destination
2. The duration is greater than 0 minutes

Now, our model could look something like this:

```
class Flight(models.Model):
    origin = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="depart
    destination = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="a
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origin} to {self.destination}"

    def is_valid_flight(self):
        return self.origin != self.destination or self.duration > 0
```


In order to make sure our application works as expected, whenever we create a new application, we are automatically given a `tests.py` file. When we first open this file, we see that Django's TestCase (https://docs.djangoproject.com/en/4.0/topics/testing/overview/) library is automatically imported:

```
from django.test import TestCase
```

One advantage to using the `TestCase` library is that when we run our tests, an entirely new database will be created for testing purposes only. This is helpful because we avoid the risk of accidentally modifying or deleting existing entries in our databse and we don't have to worry about removing dummy entries that we created only for testing.

To start using this library, we'll first want to import all of our models:

```
from .models import Flight, Airport, Passenger
```

And then we'll create a new class that extends the `TestCase` class we just imported. Within this class, we'll define a `setUp` function that will be run at the start of the testing process. In this function, we'll probably want to create. Here's what our class will look like to start:

```
class FlightTestCase(TestCase):

    def setUp(self):

        # Create airports.
        a1 = Airport.objects.create(code="AAA", city="City A")
        a2 = Airport.objects.create(code="BBB", city="City B")

        # Create flights.
        Flight.objects.create(origin=a1, destination=a2, duration=100)
        Flight.objects.create(origin=a1, destination=a1, duration=200)
        Flight.objects.create(origin=a1, destination=a2, duration=-100)
```

Now that we have some entries in our testing database, let's add some functions to this class to perform some tests. First, let's make sure our `departures` and `arrivals` fields work correctly by attempting to count the number of departures (which we know should be 3) and arrivals (which should be 1) from airport `AAA`:

```
def test_departures_count(self):
    a = Airport.objects.get(code="AAA")
    self.assertEqual(a.departures.count(), 3)

def test_arrivals_count(self):
    a = Airport.objects.get(code="AAA")
    self.assertEqual(a.arrivals.count(), 1)
```

We can also test the `is_valid_flight` function we added to our `Flight` model. We'll begin by asserting that the function does return true when the flight is valid:

```
def test_valid_flight(self):
    a1 = Airport.objects.get(code="AAA")
    a2 = Airport.objects.get(code="BBB")
    f = Flight.objects.get(origin=a1, destination=a2, duration=100)
    self.assertTrue(f.is_valid_flight())
```

Next, let's make sure that flights with invalid destinations and durations return false:

```
def test_invalid_flight_destination(self):
    a1 = Airport.objects.get(code="AAA")
    f = Flight.objects.get(origin=a1, destination=a1)
    self.assertFalse(f.is_valid_flight())

def test_invalid_flight_duration(self):
    a1 = Airport.objects.get(code="AAA")
    a2 = Airport.objects.get(code="BBB")
    f = Flight.objects.get(origin=a1, destination=a2, duration=-100)
    self.assertFalse(f.is_valid_flight())
```

Now, to run our tests, we'll run `python manage.py test`. The output for this is almost identical to the output we saw while using the Python `unittest` library, although it also logs that it is creating and destroying a testing database:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..FF.
======================================================================
FAIL: test_invalid_flight_destination (flights.tests.FlightTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/cleggett/Documents/cs50/web_notes_files/7/django/airline/flights/tests.
    self.assertFalse(f.is_valid_flight())
AssertionError: True is not false

======================================================================
FAIL: test_invalid_flight_duration (flights.tests.FlightTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/cleggett/Documents/cs50/web_notes_files/7/django/airline/flights/tests.
    self.assertFalse(f.is_valid_flight())
AssertionError: True is not false

----------------------------------------------------------------------
Ran 5 tests in 0.018s

FAILED (failures=2)
Destroying test database for alias 'default'...
```


We can see from the above output that there are times when `is_valid_flight` returned `True` when it should have returned `False`. We can see, upon further inspection of our function, that we made the mistake of using `or` instead of `and`, meaning that only one of the flight requirements must be filled for the flight to be valid. If we change the function to this:

```
def is_valid_flight(self):
    return self.origin != self.destination and self.duration > 0
```

We can then run the tests again with better results:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
----------------------------------------------------------------------
Ran 5 tests in 0.014s

OK
Destroying test database for alias 'default'...
```

Client Testing

When creating web applications, we will probably want to check not just whether or not specific functions work, but also whether or not individual web pages load as intended. We can do this by creating a `Client` object in our Django testing class, and then making requests using that object. To do this, we'll first have to add `Client` to our imports:

```
from django.test import Client, TestCase
```

For example, let's now add a test that makes sure that we get an HTTP response code of 200 and that all three of our flights are added to the context of a response:

```
def test_index(self):

    # Set up client to make requests
    c = Client()

    # Send get request to index page and store response
    response = c.get("/flights/")

    # Make sure status code is 200
    self.assertEqual(response.status_code, 200)

    # Make sure three flights are returned in the context
    self.assertEqual(response.context["flights"].count(), 3)
```

We can similarly check to make sure we get a valid response code for a valid flight page, and an invalid response code for a flight page that doesn't exist. (Notice that we use the `Max` function to find the maximum `id`, which we have access to by including `from django.db.models import Max` at the top of our file)

```
def test_valid_flight_page(self):
    a1 = Airport.objects.get(code="AAA")
    f = Flight.objects.get(origin=a1, destination=a1)

    c = Client()
    response = c.get(f"/flights/{f.id}")
    self.assertEqual(response.status_code, 200)

def test_invalid_flight_page(self):
    max_id = Flight.objects.all().aggregate(Max("id"))["id__max"]

    c = Client()
    response = c.get(f"/flights/{max_id + 1}")
    self.assertEqual(response.status_code, 404)
```

Finally, let's add some testing to make sure the passengers and non-passengers lists are being generated as expected:

```
def test_flight_page_passengers(self):
    f = Flight.objects.get(pk=1)
    p = Passenger.objects.create(first="Alice", last="Adams")
    f.passengers.add(p)

    c = Client()
    response = c.get(f"/flights/{f.id}")
    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.context["passengers"].count(), 1)

def test_flight_page_non_passengers(self):
    f = Flight.objects.get(pk=1)
    p = Passenger.objects.create(first="Alice", last="Adams")

    c = Client()
    response = c.get(f"/flights/{f.id}")
    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.context["non_passengers"].count(), 1)
```

Now, we can run all of our tests together, and see that at the moment we have no errors!

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..........
----------------------------------------------------------------------
Ran 10 tests in 0.048s
```



```
OK
Destroying test database for alias 'default'...
```

Selenium

So far, we've been able to test out the server-side code we've written using Python and Django, but as we're building up our applications we'll want the ability to create tests for our client-side code as well. For example, let's think back to our `counter.html` page and work on writing some tests for it.

We'll begin by writing a slightly different counter page where we include a button to decrease the count:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Counter</title>
        <script>

            // Wait for page to load
            document.addEventListener('DOMContentLoaded', () => {

                // Initialize variable to 0
                let counter = 0;

                // If increase button clicked, increase counter and change inner html
                document.querySelector('#increase').onclick = () => {
                    counter ++;
                    document.querySelector('h1').innerHTML = counter;
                }

                // If decrease button clicked, decrease counter and change inner html
                document.querySelector('#decrease').onclick = () => {
                    counter --;
                    document.querySelector('h1').innerHTML = counter;
                }
            })
        </script>
    </head>
    <body>
        <h1>0</h1>
        <button id="increase">+</button>
        <button id="decrease">-</button>
    </body>
</html>
```

Now if we wish to test this code, we could just open up our web browser, click the two buttons, and observe what happens. This, however, would become very tedious as you write larger and larger

single page applications, which is why several frameworks have been created that help with in-browser testing, one of which is called Selenium (https://www.selenium.dev/).

Using Selenium, we'll be able to define a testing file in Python where we can simulate a user opening a web browser, navigating to our page, and interacting with it. Our main tool when doing this is known as a **Web Driver**, which will open up a web browser on your computer. Let's take a look at how we could start using this library to begin interacting with pages. Note that below we use both `selenium` and `ChromeDriver`. Selenium can be installed for python by running `pip install selenium`, and `ChromeDriver` can be installed by running `pip install chromedriver-py`

```
import os
import pathlib
import unittest

from selenium import webdriver

# Finds the Uniform Resourse Identifier of a file
def file_uri(filename):
    return pathlib.Path(os.path.abspath(filename)).as_uri()

# Sets up web driver using Google chrome
driver = webdriver.Chrome()
```

The above code is all of the basic setup we need, so now we can get into some more interesting uses by employing the Python interpreter. One note about the first few lines is that in order to target a specific page, we need that page's **Uniform Resource Identifier (URI)** which is a unique string that represents that resource.

```
# Find the URI of our newly created file
>>> uri = file_uri("counter.html")

# Use the URI to open the web page
>>> driver.get(uri)

# Access the title of the current page
>>> driver.title
'Counter'

# Access the source code of the page
>>> driver.page_source
'<html lang="en"><head>\n        <title>Counter</title>\n        <script>\n

# Find and store the increase and decrease buttons:
>>> increase = driver.find_element_by_id("increase")
>>> decrease = driver.find_element_by_id("decrease")

# Simulate the user clicking on the two buttons
>>> increase.click()
>>> increase.click()
>>> decrease.click()
```

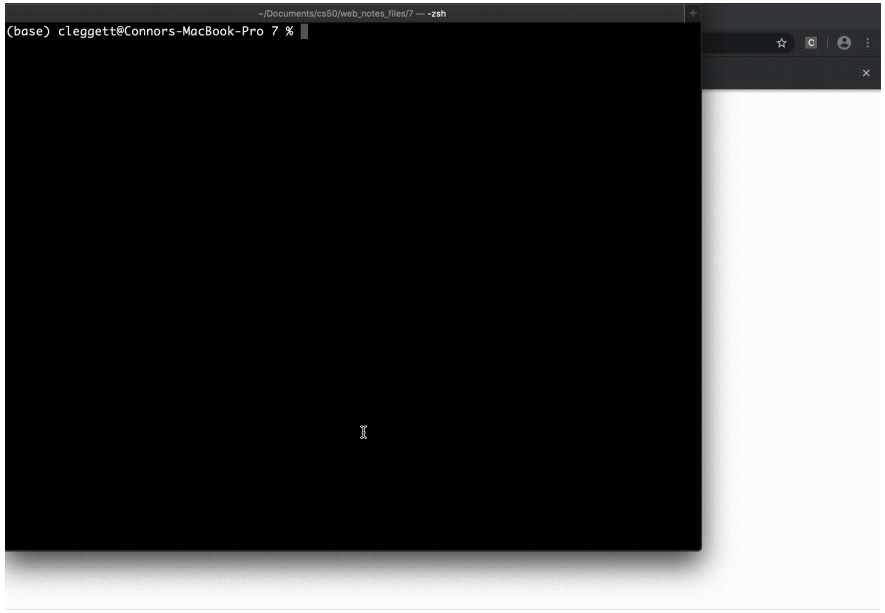


```
# We can even include clicks within other Python constructs:
>>> for i in range(25):
...     increase.click()
```

Now let's take a look at how we can use this simulation to create automated tests of our page:

```
# Standard outline of testing class
class WebpageTests(unittest.TestCase):

    def test_title(self):
        """Make sure title is correct"""
        driver.get(file_uri("counter.html"))
        self.assertEqual(driver.title, "Counter")

    def test_increase(self):
        """Make sure header updated to 1 after 1 click of increase button"""
        driver.get(file_uri("counter.html"))
        increase = driver.find_element_by_id("increase")
        increase.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "1")

    def test_decrease(self):
        """Make sure header updated to -1 after 1 click of decrease button"""
        driver.get(file_uri("counter.html"))
        decrease = driver.find_element_by_id("decrease")
        decrease.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "-1")

    def test_multiple_increase(self):
        """Make sure header updated to 3 after 3 clicks of increase button"""
        driver.get(file_uri("counter.html"))
        increase = driver.find_element_by_id("increase")
        for i in range(3):
            increase.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "3")

if __name__ == "__main__":
    unittest.main()
```

Now, if we run `python tests.py`, our simulations will be carried out in the browser, and then the results of the tests will be printed to the console. Here's an example of what this might look like when we have a bug in the code a test fails:



CI/CD

CI/CD, which stands for **Continuous Integration and Continuous Delivery**, is a set of software development best practices that dictate how code is written by a team of people, and how that code is later delivered to users of the application. As the name implies, this method consists of two main parts:

- Continuous Integration:
  - Frequent merges to the main branch
  - Automated unit testing with each merge
- Continuous Delivery:
  - Short release schedules, meaning new versions of an application are released frequently.

CI/CD has become more and more popular among software development teams for a number of reasons:

- When different team members are working on different features, many compatibility issues can arise when multiple features are combined at the same time. Continuous integration allows teams to tackle small conflicts as they come.

- Because unit tests are run with each Merge, when a test fails it is easier to isolate the part of the code that is causing the problem.
- Frequently releasing new versions of an application allows developers to isolate problems if they arise after launch.
- Releasing small, incremental changes allows users to slowly get used to new app features rather than being overwhelmed with an entirely different version
- Not waiting to release new features allows companies to stay ahead in a competitive market.

GitHub Actions

One popular tool used to help with continuous integration is known as GitHub Actions (https://github.com/features/actions). GitHub Actions will allow us to create workflows where we can specify certain actions to be performed every time someone pushes to a git repository. For example, we might want to check with every push that a style guide is adhered to, or that a set of unit tests is passed.

In order to set up a GitHub action, we'll use a configuration language called **YAML**. YAML structures its data around key-value pairs (like a JSON object or Python Dictionary). Here's an example of a simple YAML file:

```yaml
key1: value1
key2: value2
key3:
    - item1
    - item2
    - item3
```

Now, let's look at an example of how we would configure a YAML file (which takes the form `name.yml` or `name.yaml`) that works with GitHub Actions. To do this, I'll create a `.github` directory in my repository, and then a `workflows` directory inside of that, and finally a `ci.yml` file within that. In that file, we'll write:

```yaml
name: Testing
on: push

jobs:
  test_project:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Run Django unit tests
      run: |
        pip3 install --user django
        python3 manage.py test
```

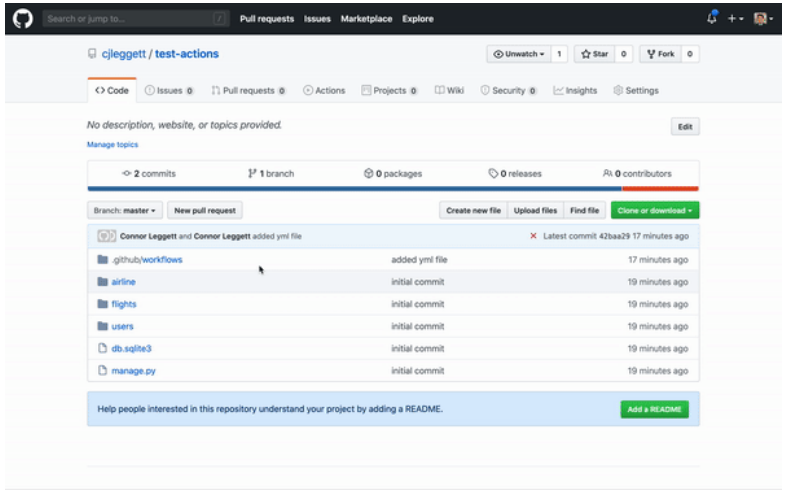
Since this is our first time using GitHub Actions, let's go through what each part of this file is doing:

- First, we give the workflow a `name`, which in our case is Testing.
- Next, with the `on` key, we specify when the workflow should run. In our case, we wish to perform the tests every time someone pushes to the repository.
- The rest of the file is contained within a `jobs` key, which indicates which jobs should be run at every push.
  - In our case, the only job is `test_project`. Every job must define two componenets
    - The `runs-on` key specifies which of GitHub's virtual machines we would like our code to be run on.
    - The `steps` key provides the actions that should occur when this job is run
      - In the `uses` key we specify which GitHub action we wish to use. `actions/checkout@v2` is an action written by GitHub that we can use.
      - The `name` key here allows us to proide a description of the action we're taking
      - After the `run` key, we type the commands we wish to run on GitHub's server. In our case we wish to install Django and then run the testing file.

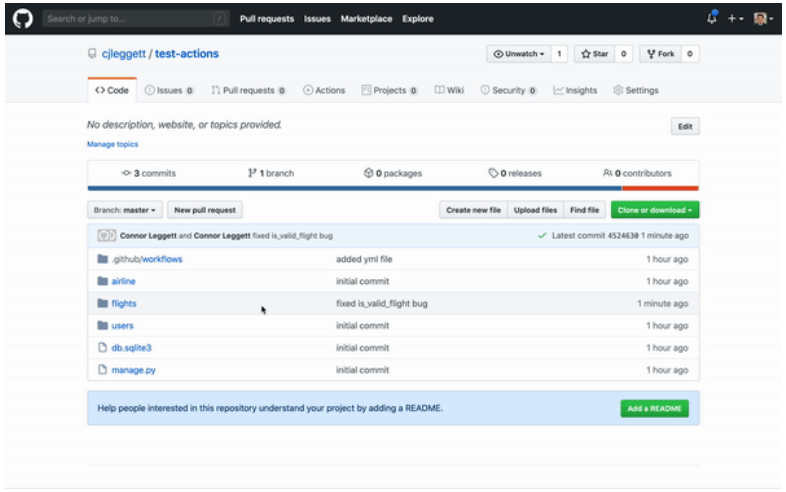
Now, let's open up our repository in GitHub and take a look at some of the tabs near the top of the page:

- **Code**: This is the tab that we've been using most frequently, as it allows us to view the files and folders within our directory.
- **Issues**: Here we can open and close issues, which are requests for bug fixes or new features. We can think of this as a to-do list for our application.
- **Pull Requests**: Requests from people who wish to merge some code from one branch into another one. This is a useful tool, as it allows people to perform **code reviews** where they comment and provide suggestions before code is integrated into the master branch.
- **GitHub Actions**: This is the tab we'll use when working on continuous integration, as it provides logs of the actions that have taken place after each push.

Here, let's imagine that we pushed our changes *before* we fixed the bug we had in the `is_valid_flight` function in `models.py` within our `airport` project. We can now navigate to the **GitHub Actions** tab, click on our most recent push, click on the action that failed, and view the log:



Now, after fixing the bug, we could bush again and find a better outcome:



Docker

Problems can arise in the world of software development when the configuration on your computer is different than the one your application is being run on. You may have a different version of Python or some additional packages installed that allow the application to run smoothly

on your computer, while it would crash on your server. To avoid these problems, we need a way to make sure everyone working on a project is using the same environment. One way to do this is to use a tool called **Docker**, which is a containerization software, meaning it creates an isolated environment within your computer that can be standardized among many collaborators and the server on which your site is run. While Docker is a bit like a **Virtual Machine**, they are in fact different technologies. A virtual machine (like the one used on GitHub Actions or when you launch an AWS server) is effectively an entire virtual computer with its own operating system, meaning it ends up taking a lot of space wherever it is running. Dockers, on the other hand, work by setting up a container within an existing computer, therefore taking up less space.

Now that we have an idea of what a Docker container is, let's take a look at how we can configure one on our computers. Our first step in doing this will be to create a **Docker File** which we'll name `Dockerfile`. Inside this file, we'll provide instructions for how to create a **Docker Image** which describes the libraries and binaries we wish to include in our container. Here's an example of what our `Dockerfile` might look like:

```
FROM python:3
COPY .  /usr/src/app
WORKDIR /usr/src/app
RUN pip install -r requirements.txt
CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]
```

Here, we'll take an in-depth look at what the above file actually does:

- `FROM python3`: this shows that we are basing this image off of a standard image in which Python 3 is installed. This is fairly common when writing a Docker File, as it allows you to avoid the work of re-defining the same basic setup with each new image.
- `COPY . /usr/src/app`: This shows that we wish to copy everything from our current directory (`.`) and store it in the `/usr/src/app` directory in our new container.
- `WORKDIR /usr/src/app`: This sets up where we will run commands within the container. (A bit like `cd` on the terminal)
- `RUN pip install -r requirements.txt`: In this line, assuming you've included all of your requirements to a file called `requirements.txt`, they will all be installed within the container.
- `CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]`: Finally, we specify the command that should be run when we start up the container.

So far in this class, we've only been using SQLite as that's the default database management system for Django. In live applications with real users though, SQLite is almost never used, as it is not as easily scaled as other systems. Thankfully, if we wish to run a separate server for our database, we can simply add another Docker container, and run them together using a feature called **Docker Compose**. This will allow two different servers to run in separate containers, but also

be able to communicate with one another. To specify this, we'll use a YAML file called `docker-compose.yml`:

```
version: '3'

services:
    db:
        image: postgres

    web:
        build: .
        volumes:
            - .:/usr/src/app
        ports:
            - "8000:8000"
```

In the above file we:

- Specify that we're using version 3 of Docker Compose
- Outline two services:
  - `db` sets up our database container based on an image already written by Postgres.
  - `web` sets up our server's container by instructing Docker to:
    - Use the Dockerfile within the current directory.
    - Use the specified path within the container.
    - Link port 8000 within the container to port 8000 on our computer.

Now, we're ready to start up our services with the command `docker-compose up`. This will launch both of our servers inside of new Docker containers.

At this point, we may want to run commands within our Docker container to add database entries or run tests. To do this, we'll first run `docker ps` to show all of the docker containers that are running. Then, well find the `CONTAINER ID` of the container we wish to enter and run `docker exec -it CONTAINER_ID bash -l`. This will move you inside the `usr/src/app` directory we set up within our container. We can run any commands we wish inside that container and then exit by running `CTRL-D`.

That's all for this lecture! Next time, we'll working on scaling up our projects and making sure they are secure.

CS50's Web Programming with Python and JavaScript

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Lecture 8

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, started using Django to create web applications, and learned how to use Django models to store information in our sites. We then introduced JavaScript and learned how to use it to make web pages more interactive, and talked about using animation and React to further improve our User Interfaces. We then talked about some best practices in software development and some technologies commonly used to achieve those best practces.
- Today, in our final lecture, we'll discuss the issues of scaling up and securing our web applications.

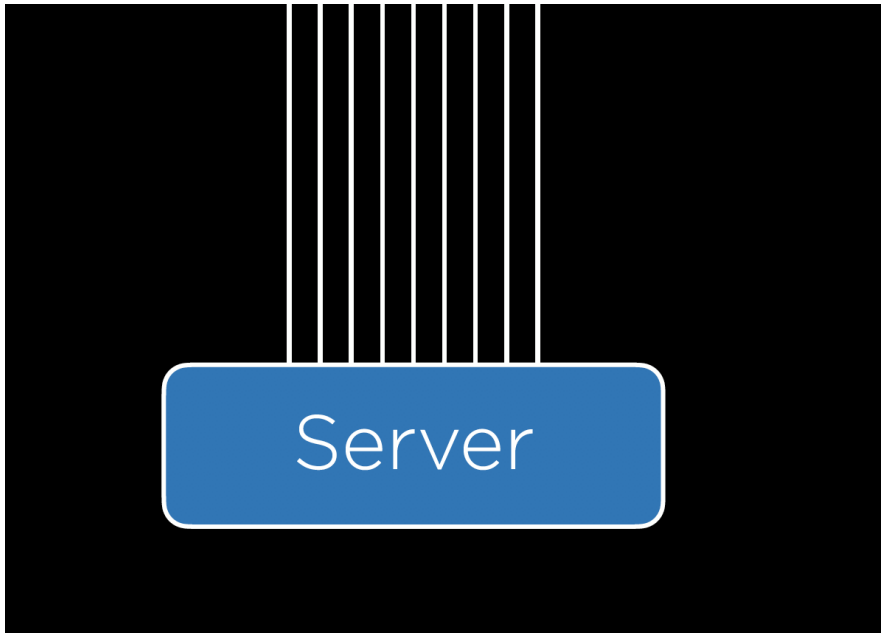
Scalability

So far in this course, we've built applications that are run only locally on our computers, but eventually, we'll want to launch our sites so they canbe accessed by anyone on the internet. In order to do this, we run our sites on **servers**, which are physical pieces of hardware dedicated to running applications. Servers can either be on-premise (We own and maintain physical servers where our application is hosted) or on the cloud (servers are owned by a different company such as Amazon or Google, and we pay to rent server space where our application is hosted). There are benefits and drawbacks to both options:

- **Customization**: Hosting your own servers gives you the ability to decide exactly how they work, allowing for more flexibility than cloud-based hosting.
- **Expertise**: It is much simpler to host an application on the cloud than it is to maintain your own servers.
- **Cost**: Since server-hosting sites need to make a profit, they will charge you more than it costs them to maintain their on-premise servers, making cloud-based servers more expensive. However, the startup costs of running on-premise servers can be high, as you need to purchase physical servers and potentially hire someone with the expertise to set them up.

- **Scalability**: Scaling is typically easier when hosting on the cloud. For example, if we host a site on premise that gets 500 visits per day, and then it starts getting 500,000 visits per day, we would have to order and setup more physical servers to handle the requests, and in the mean time many of our users will not be able to access the site. Most cloud hosting sites will allow you to rent server space flexibly, paying based on how much action your site sees.

When a user sends an HTTP request to this server, the server should send back a response. However, in reality, most servers get far more than one request at a time, as depicted below:



This is where we run into the issue of scalability. A single server can handle only so many requests at once, forcing us to make plans about what to do when our one server is overworked. Whether we decide to host on premise or on the cloud, we have to determine how many requests a server can handle without crashing, which can be done using any number of **benchmarking** tools including Apache Bench.

Scaling

Once we have some upper limit on how many requests our server can handling, we can begin thinking about how we want to handle the scaling of our application. Two different approaches to

scaling include:

1. **Vertical Scaling**: In vertical scaling, when our server is overwhelmed we simply buy or build a larger server. This strategy is limited however, as there is an upper limit on how powerful a single server can be.
2. **Horizontal Scaling**: In horizontal scaling, when our server is overwhelmed we buy or build more servers, and then split the requests among our multiple servers.

Load Balancing

When we use horizontal scaling, we are faced with the additional problem of how we decide which servers are assigned to which requests. We answer that question by employing a **load balancer**, which is another piece of hardware that intercepts incoming requests, and then assigns those requests to one of our servers. There are a number of different methods for deciding which server receives which request, but here are a few:

- **Random**: In this simple method, the load balancer will decide randomly which server it should assign a request to.
- **Round-Robin**: In this method, the load balancer will alternate which server receives an incoming request. If we have three servers, the first request might go to server A, the second to server B, the third to server C, and the fourth back to server A.
- **Fewest Connections**: In this method, the load balancer looks for the server that is currently handling the fewest requests, and assigns the incoming request to that server. This allows us to make sure we're not overworking one particular server, but it also takes longer for the load balancer to calculate the number of requests each server is currently handling than it dows for it to simply choose a random server.

There is no method of load balancing that is strictly better than all other methods, and there are many different methods used in practice. One problem that can arise when scaling horizontally is that we might have sessions that are stored on one server but not another, and we don't want users to have to re-enter information just because the load balancer pushes their request to a new server. Like many problems of scalability, there are multiple approaches to solving the problem of sessions:

- **Sticky Sessions**: Once a user visits a site, the load balancer remembers which server they were sent to first, and makes sure to send them to the same one. One big concern with this method is that we could end up having a large number of users sticking to one server, causing it to crash.
- **Database Sessions**: All sessions are stored in a database that all servers have access to. This way, a user's information will be available no matter which server they are assigned to. The

drawback here is that it takes additional time and computing power to read from and write to a database.

- **Client-Side Sessions**: Rather than storing information on our servers, we can choose to store them locally on the user's web browser as cookies. The drawbacks to this method include the security concern of users creating false cookies that allow them to log in as another user, and the computational concern of sending cookie information back and forth with every request.

Like with load balancing, there is no best answer to the sessions problem, and the method you choose will often depend on your specific circumstances.

Autoscaling

Another problem that we could run into is that many websites are visited much more frequently at certain times. For example, if we decide to launch our "Is it New Year's?" app from earlier, we might expect it to get a lot more traffic in late December to early January than any other time of year. If we buy enough servers for the site to stay active during the winter, those servers would be sitting idle for the rest of the year, wasting space and energy. This scenario has brought about the idea of **autoscaling** which has become common in cloud computing, where the number of servers being used by your site can grow and shrink based on the number of requests it gets. Autoscaling is not a perfect solution though, as it takes time to determine that a new server is needed and to launch that server. Another potential problem is that the more servers you have running, the more opportunity there is for one to fail.

Server Failure

Having multiple servers though, can help to avoid what is known as a **Single Point of Failure**, which is a piece of hardware that, after failing, will cause the entire site to crash. When scaling horizontally, the load balancer can detect which servers have crashed by sending periodic **heartbeat** requests to each server, and then stop assigning new requests to servers that have crashed. At this point, it seems we have simply moved our single point of failure from a server to the load balancer, but we can account for this by having backup load balancers available if our original happens to crash.

Scaling Databases

In addition to scaling our servers that process requests, we'll also want to think of ways to scale our Databases. In this course we use SQLite which stores data inside a file on the server, but as we store more and more data, it sometimes makes more sense to store data in a number of different files, and maybe even on a separate server. This brings up the problem then of what to do when

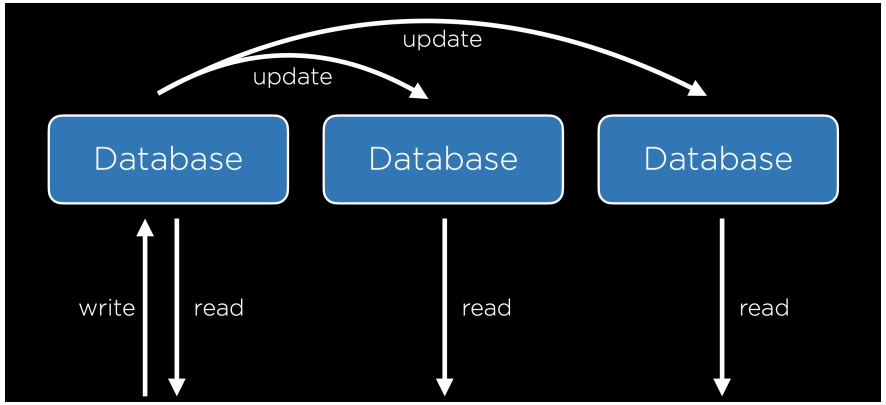
our database server can no longer handle all of the requests coming in. Like in other issues of scalability, there are a number of methods we can use to mitigate this problem:

- **Vertical Partitioning**: This is a method similar to the one we used when first discussing SQL, where we split our data into multiple different tables rather than having redundant information in one table. (Feel free to look back on lecture 4 where we split the `flights` table into a `flights` table and an `airports` table).
- **Horizontal Partitioning**: This method involves storing multiple tables with the same format, but different information. For example, we could split a `flights` table into a `domestic_flights` table and an `international_flights` table. This way, when we wish to search for a flight from JFK to LHR, we don't have to waste time searching through a table full of domestic flights. One drawback to this method is that it can be expensive to join multiple tables once they have been split.

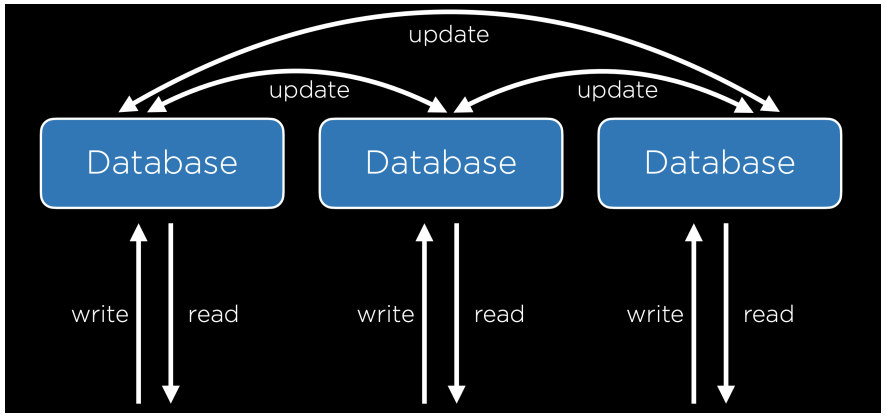
Database Replication

Even after we've scaled a database, it seems we're still left with a single point of failure. If our database server crashes, all of our data could be lost. Just as we added more servers to avoid a single point of failure, we can add copies of our database to make sure the failure of one database does not shut down our application. Also like before there are different methods of database replication, two of the most popular of which are:

- **Single-Primary Replication**: In this method there are multiple databases, but only one of them is considered to be the primary database, meaning you can read from and write to one of the databases, but only read from each of the others. When the primary database is updated, the other databases are then updated to match the primary one. One drawback of this method is that it still contains a single point of failure when it comes to writing to the database.



- **Multi-Primary Replication**: In this method, all of the databases can be read from and written to. This solves the problem of a single point of failure, but it comes with a tradeoff: it is now much more difficult to keep all databases up to date because each database must be aware of changes to all other databases. This system also sets us up for the possibility of some conflicts:
  - **Update Conflict**: With multiple databases, one user may attempt to edit a row in one database while another user attempts to edit that same row in a different database, causing a problem when the databases sync up.
  - **Uniqueness Conflict**: Every row in a SQL database must have a unique identifier, and we may run into the problem that we assign the same id to two different entries in two different databases.
  - **Delete Conflict**: One user may delete a row while another user attempts to update it.



Caching

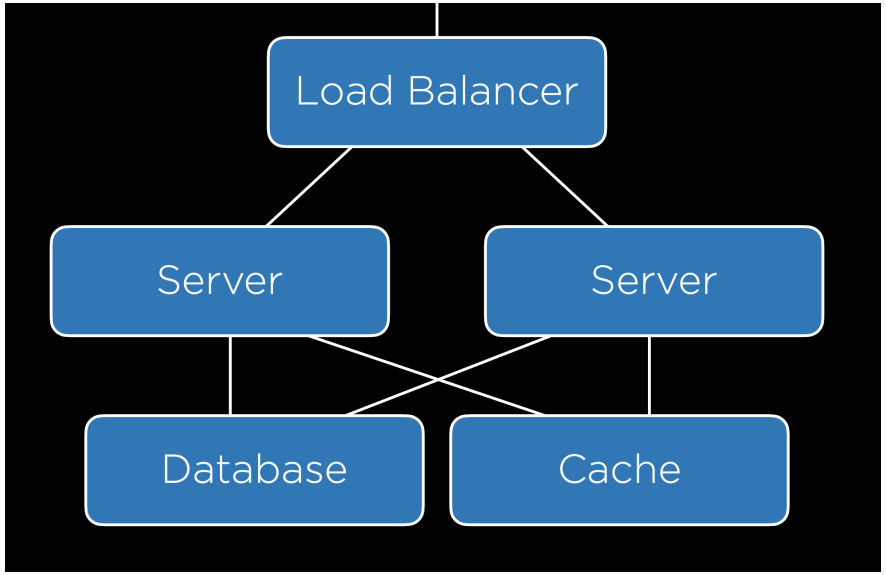
Whenever we're working with larger databases, it is important to recognize that every interaction we have with a database is costly. Therefore, we wish to minimize the number of calls to our database server. Let's look, for example, at the New York Times (https://www.nytimes.com/) website. The New York Times may have some database with all of their articles which is queried and some template that is rendered every time someone loads the home page, but this would be a waste of resources, as the articles displayed on the home page likely do not change much from second to second. One way we can deal with this problem is by using **Caching**, which is the idea of storing some information in a more accessible location if we anticipate needing it again in the near future.

One way that caching can be implemented is by storing data on the user's web browser, so that when a user loads certain pages, no request to the server even needs to be sent. One way to do this is by including this line in the header of an HTTP response:

```
Cache-Control: max-age=86400
```

This will tell the browser that when visiting a page, as long as I have visited that page within the last 86400 milliseconds, no request has to be made to the server. This method is used commonly by web browsers especially with files that are less likely to change over short periods such as a CSS file. To take more control over this process, we can also add an `ETag` to the HTTP response header, which is a unique sequence of characters that represents a specific version of a document. This is useful because future requests can include this tag and compare it to the tag of the latest document on the server, only returning an entire document when the two differ.

In addition to the client-side caching discussed above, it can often be helpful to include a cache on the server side. With this cache, our backend setup will look a bit like the one below, where all servers have access to a cache.



Django provides its own cache framework (https://docs.djangoproject.com/en/4.0/topics/cache/) which will allow us to incorporate caching in our projects. This framework offers several ways of implementing a cache:

- **Per-View Caching**: This allows us to decide that once a specific view has been loaded, that same view can be rendered without going through the function for the next specified amount of time.
- **Template-Fragment Caching**: This caches specific parts of a template so they do not have to be re-rendered. For example, we may have a navigation bar that rarely changes, meaning we could save time by not reloading it.
- **Low-Level Cache API**: This allows you to do more flexible caching, essentially storing any information you would like to.

We won't go into the details here of how to implement caching in Django, but do take a look at the documentation (https://docs.djangoproject.com/en/4.0/topics/cache/) if you're interested!

Security

Now, we'll begin to discuss how to make sure our web applications are secure, which will involve many different measures that span nearly every topic we've discussed in this course.

Git and GitHub

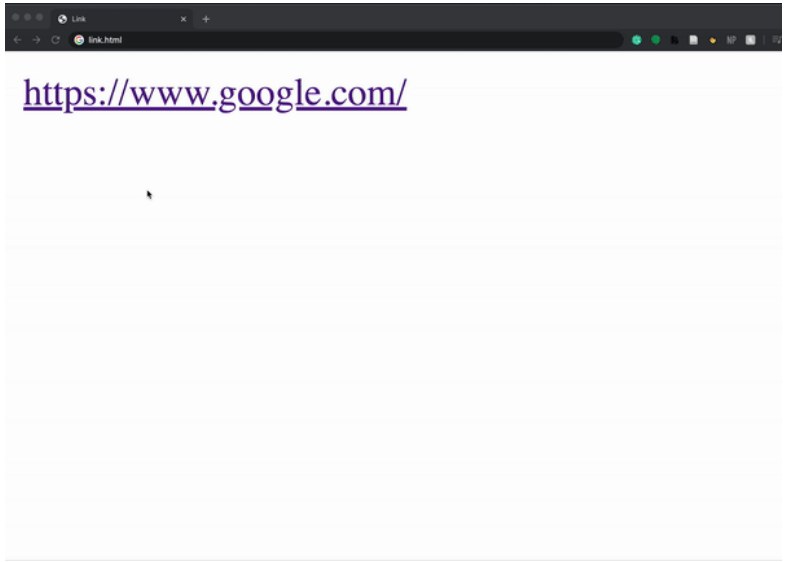
One of the greatest strengths of Git and GitHub is how easy they make it to share and contribute to **open-source software**, which can be seen and contributed to by anyone on the internet. One drawback to this is that if at any point you commit a file that includes some private credentials like a password or API key, those credentials could be publicly available.

HTML

There are many vulnerabilities that arise from using HTML. One common weakness is known as a **Phishing Attack**, which occurs when a user who thinks they are going to one page is actually taken to another. These are not necesarily things we can account for when designing a website, but we should definitely keep them in mind when interacting with the web ourselves. For example, a malicious user might write out this HTML:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Link</title>
    </head>
    <body>
        <a href="https://cs50.harvard.edu/">https://www.google.com/</a>
    </body>
</html>
```

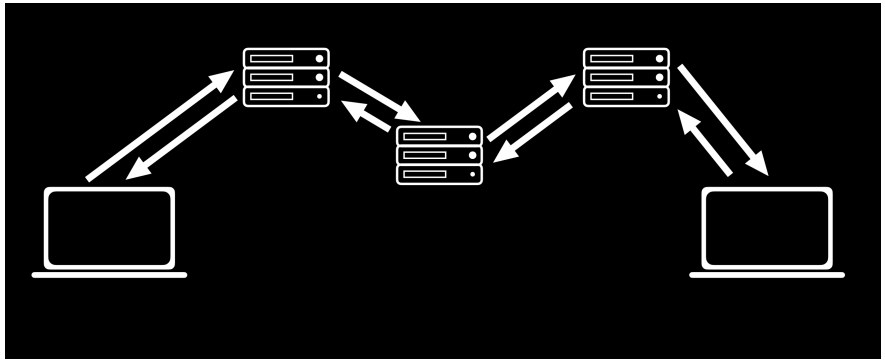
Which acts like this:



The fact that HTML is actually sent to a user as part of a request opens up more vulnerabilities, because everyone has access to the layout and style that allowed you to create your site. For example, a hacker could go to bankofamerica.com (https://cs50.harvard.edu/), copy all of their HTML, and paste it in their own site, creating a site that looks exactly like Bank of America's. The hacker could then redirect the login form on the page so all usernames and passwords are sent to them. (Also, here's the real Bank of America link (https://www.bankofamerica.com/) – just wanted to see if you were checking urls before clicking!)

HTTPS

As we discussed earlier in the course, most interactions that occur online follow HTTP protocol, although now more and more transactions use HTTPS, which is an encrypted version of HTTP. While using these protocols, information is transferred from one computer to another through a series of servers as pictured below.



There is often no way to ensure that all of these transfers are secure, so it is important that all of this transferred information is **encrypted**, meaning that the characters of the message are altered so that the sender and receiver of the message can understand it, but no one else can.

Secret-Key Cryptography

One approach to this is known as **Secret-Key Cryptography**. In this approach, the sender and receiver both have access to a secret key that only they know. Then, the secret key is used by the sender to encrypt a message which is then sent to the recipient who uses the secret key to decrypt the message. This method is extremely secure, but it produces a big problem when it comes to practicality. In order for it to work, both the sender and the receiver must have access to the secret key, which means they must meet in person to exchange a key securely. With the number of different websites we interact with on a daily basis, it is clear that in-person meetups are not an option.

Public-Key Cryptography

An incredible advancement in cryptography that allows the internet to function as it does today is known as **Public-Key Cryptography**. In this method, there are two keys: one is public and can be shared, and the other must be kept private. Once these keys are established (there are several different mathematical methods for creating pairs of keys which could make up an entire course on their own, so we won't discuss them here), a sender could look up the public key of a recipient and use it to encrypt a message, and then the recipient could use their private key to decrypt the message. When we use HTTPS rather than HTTP, we know that our request is being secured using public-key encryption.

Databases

In addition to our requests and responses, we must also make sure that our databases are secure. One common thing we'll need to store is user information, including usernames and passwords as in the table below:

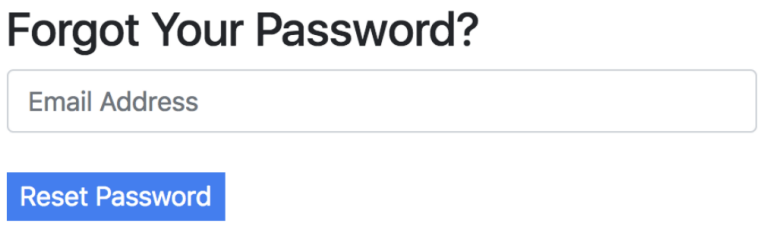
id	username	password
1	harry	hello
2	ron	password
3	hermione	12345
4	ginny	abcdef
5	luna	qwerty

However, you never actually want to store passwords in plaintext in case an unauthorized person gets access to your database. Instead, we'll want to use a **hash function**, a function that takes in some text and outputs a seemingly random string, to create a hash of each password, as in the table below:

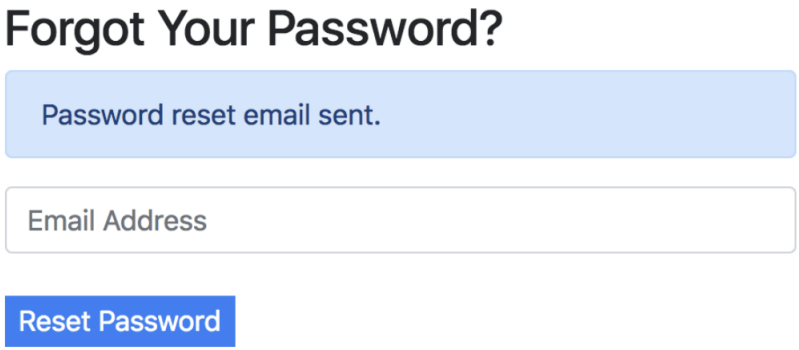
id	username	password
1	harry	48c8e8c3f9e80b68ac67304c7c510e9fcb
2	ron	6024aba15e3f9be95e3c9e6d3bf261d78e
3	hermione	90112701066c0a536f2f6b2761e5edb09e
4	ginny	b053b7574c8a25751e2a896377e5d477c5
5	luna	a4048eaaee50680532845b2025996b44a9

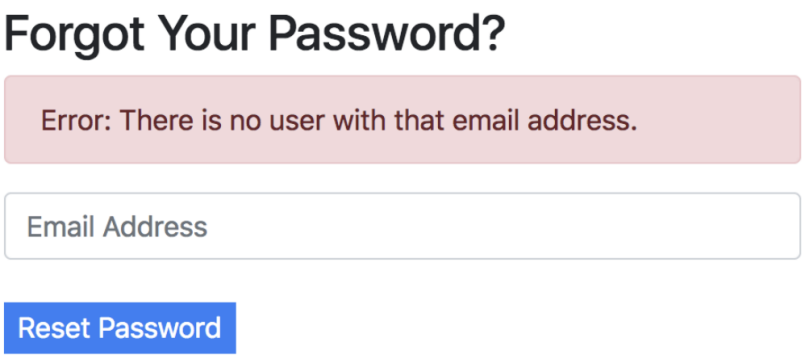
It is important to note that a hash function is **one-way**, meaning it can turn a password into a hash, but cannot turn a hash back into a password. This means that any company that stores user information this way does not actually know any of the users' passwords, meaning each time a user attempts to sign in, the entered password will be hashed and compared to the existing hash. Thankfully, this process is already handled for us by Django. One implication of this storage technique is that when a user forgets their password, a company has no way of telling them what their old password now, meaning they would have to make a new one.

There are some cases where you'll have to decide as a developer how much information you are willing to leak. For example, many sites have a page for forgotten passwords that looks like this:



As a developer, you may want to include either a success or error message after submission:





But notice how by typing in emails, anyone could determine who has an email registered with that site. This could be totally fine in cases where whether or not a person uses the site is inconsequential (maybe Facebook), but extremely reckless if the fact that you are a member of a certain site could put you in danger (maybe an online support group for victims of abuse).

Another way data could be leaked is in the time it takes for a response to come back. It probably takes less time to reject someone with an invalid email than a person with a correct email address and a wrong password.

As we discussed earlier in the course, we must be ware of SQL Injection Attacks whenever we use straight SQL queries in our code.

APIs

We often use JavaScript in conjunction with APIs to build single-page applications. In the case when we build our own API, there are a few methods we can use to keep our API secure:

- **API Keys**: Only process requests from API clients who have a key you have provided to them.
- **Rate Limiting**: Limit the number of requests any one user can make in a given time frame. This helps protect against **Denial of Service (DOS) Attacks**, in which a malicious user makes so many calls to your API that it crashes.
- **Route Authentication**: There are many cases where we don't want to give everyone access to all of our data, so we can use route authentication to make sure only specific users can see specific data.

Environment Variables

Just as we want to avoid storing passwords in plaintext, we'll want to avoid including API keys in our source code. One common way of avoiding this is to use **environment variables**, or variables that are stored in your operating system or server's environment. Then, rather than including a string of text in our source code, we can include a reference to an environment variable.

JavaScript

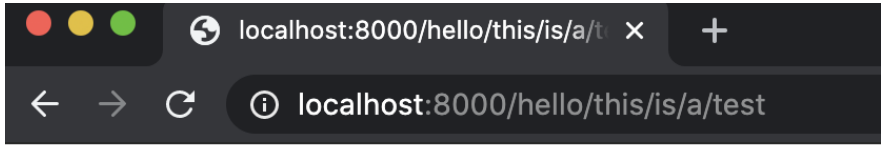
There are a few types of attacks that malicious users may attempt using JavaScript. One example is knwon as **Cross-Site Scripting**, which is when a user writes their own JavaScript code and runs it on your website. For example, let's imagine we have a Django application with a single URL:

```
urlpatterns = [
    path("<path:path>", views.index, name="index")
]
```

and a single view:

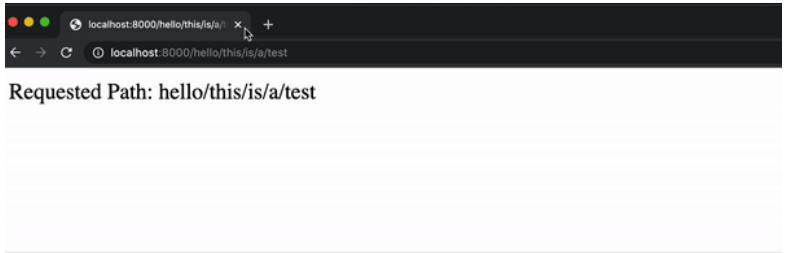
```
def index(request, path):
    return HttpResponse(f"Requested Path: {path}")
```

This website essentially tells the user what URL they have navigated to:



Requested Path: hello/this/is/a/test

But a user can now easily insert some Javascript into the page by typing it in the url:



While this `alert` example is fairly harmless, it wouldn't be all that more difficult to include some JavaScript that manipulates the DOM or uses `fetch` to send a request.

Cross-Site Request Forgery

We already discussed how we can use Django to prevent CSRF attacks, but let's take a look at what could happen without this protection. As an example, imagine a bank has a URL you could visit that transfers money out of your account. A person could easily create a link that would make this transfer:

```
<a href="http://yourbank.com/transfer?to=brian&amt=2800">
    Click Here!
</a>
```

This attack can be even more subtle than a link. If the URL is put in an image, then it will be accessed as your browser attempts to load the image:

```
<img src="http://yourbank.com/transfer?to=brian&amt=2800">
```

Because of this, whenever you are building an application that can accept some state change, it should be done using a POST request. Even if the bank requires a POST request, hidden form fields can still trick users into accidentally submitting a request. The following form doesn't even wait for the user to click; it automatically submits!

```
<body onload="document.forms[0].submit()">
   <form action="https://yourbank.com/transfer"
   method="post">
       <input type="hidden" name="to" value="brian">
       <input type="hidden" name="amt" value="2800">
       <input type="submit" value="Click Here!">
   </form>
</body>
```

The above is an example of what **Cross-Site Request Forgery** might look like. We can stop attacks such as these by creating a CSRF token when loading a webpage, and then only accepting forms

with a valid token.

What's next?

We've discussed many web frameworks in this class such as Django and React, but there are more frameworks you might be interested in trying:

- Server-Side
  - Express.js (https://expressjs.com/)
  - Ruby on Rails (https://rubyonrails.org/)
  - Flask (https://flask.palletsprojects.com/en/1.1.x/)
  - ...
- Client-Side
  - Angular JS (https://angularjs.org/)
  - React (https://reactjs.org/)
  - Vue.js (https://vuejs.org/)
  - React Native (https://reactnative.dev/)
  - ...

In the future, you may also want to be able to deploy your site to the web, which you can do through a number of different services:

- Amazon Web Services (https://aws.amazon.com/getting-started/hands-on/websites/)
- GitHub (https://github.com/)
- Heroku (https://www.heroku.com/)
- Netlify (https://app.netlify.com/)
- Google Cloud (https://cloud.google.com/)
- Microsoft Azure (https://azure.microsoft.com/en-gb/)
- ...

We've come a long way and covered a lot of material since the beginning of this course, but there's still a lot to learn in the world of web programming. Although it can be overwhelming at times, one of the best ways to learn more is to jump into a project and see how far you can run with it. We believe that at this point you have a strong foundation in the concepts of web design, and that you have what it takes to turn an idea into your own working website!