This is CS50x

OpenCourseWare

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://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Web

What to Do

1. After watching Introduction, HTTP, HTML, CSS, JavaScript, and Homepage, submit Homepage.
2. After watching Flask, Databases, and Finance, submit Finance.

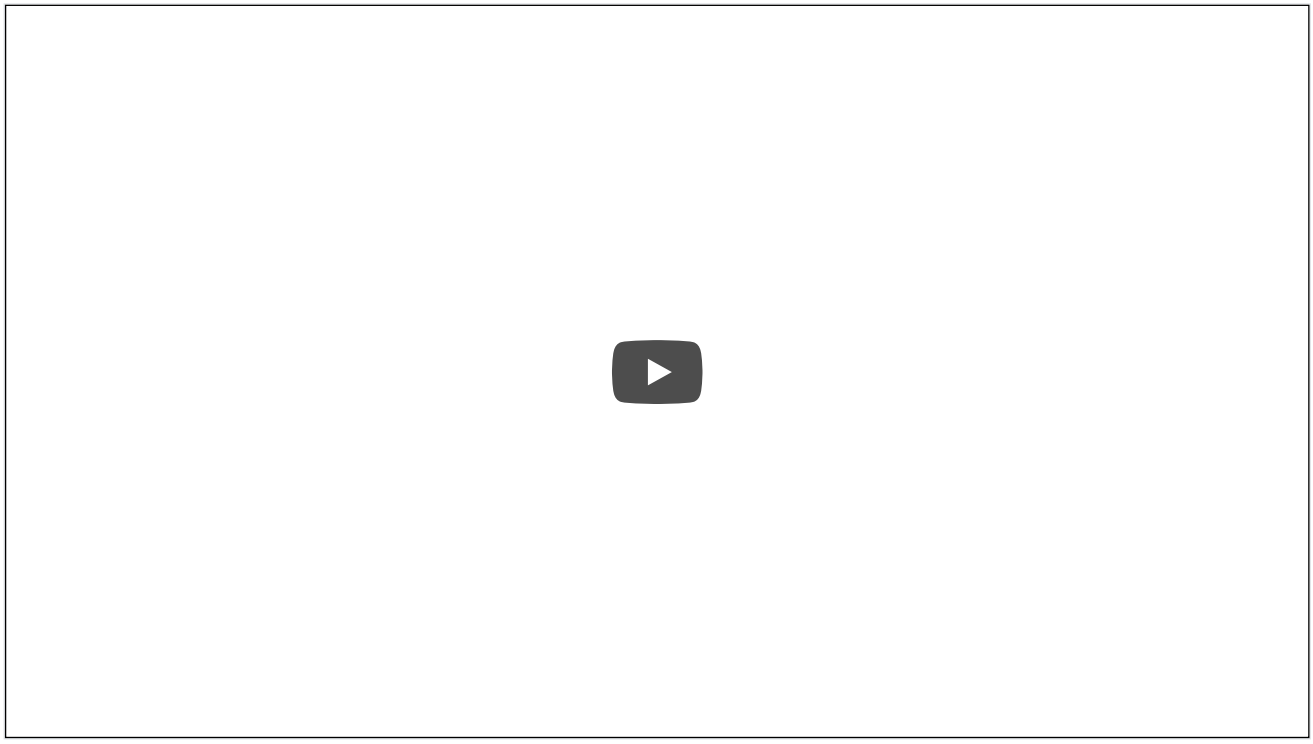
When to Do It

By 11:59pm on 31 December 2020.

How to Do It

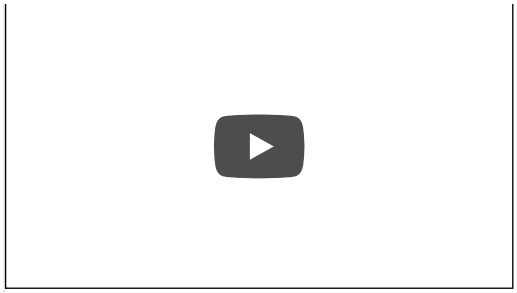
⊞ Source Code

▼ **Introduction**



- In this track, we'll write programs that can run on the internet. We'll first learn about the basics of the internet and how it works, and then dive into the languages of the internet, from HTML and CSS to JavaScript to frameworks in Python and SQL that can turn a webpage into an application.

▼ **HTTP**



- Computers talk to each other across the network by sending and receiving messages. At the most basic level, there are standard protocols, or rules to follow, for sending and receiving messages. In the context of the internet, the standard protocol is TCP/IP, Transmission Control Protocol and Internet Protocol. We can think of this at a high-level as sending a letter in the mail, with an address for the recipient and the address of the sender. On the internet, computers have IP addresses, usually in the format `#.#.#.#`, so our digital envelope might include `1.2.3.4` for the address of the computer we want to message, and our own address `5.6.7.8`, so that we can get a response.
- [2:16] With four numbers of one byte each, an IP address is 32 bits, which only allows us to count up to about 4 billion. It turns out that we now have more devices than 32 bits will support, and so in addition to IPv4, the protocol with 32-bit addresses, we also have IPv6, a protocol with 128-bit addresses.
- [4:10] In addition to the address of the recipient, we also specify a port number, or a number assigned to a particular service or type of message, like emails, webpages, or files. This way, the recipient computer can process incoming messages with the right program. So our envelope might say `1.2.3.4:80`.
- [5:50] But when we visit a website, we probably type in something like `example.com`, and it turns out that there's something called DNS, Domain Name System, which maps domain names to IP addresses of the servers that can respond for that domain.
- [7:40] And we might notice URLs are the form `http://www.example.com`, and HTTP is short for another protocol, Hypertext Transfer Protocol, which essentially describe the format of the contents inside each digital envelope. The content of a request in HTTP might look like:

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

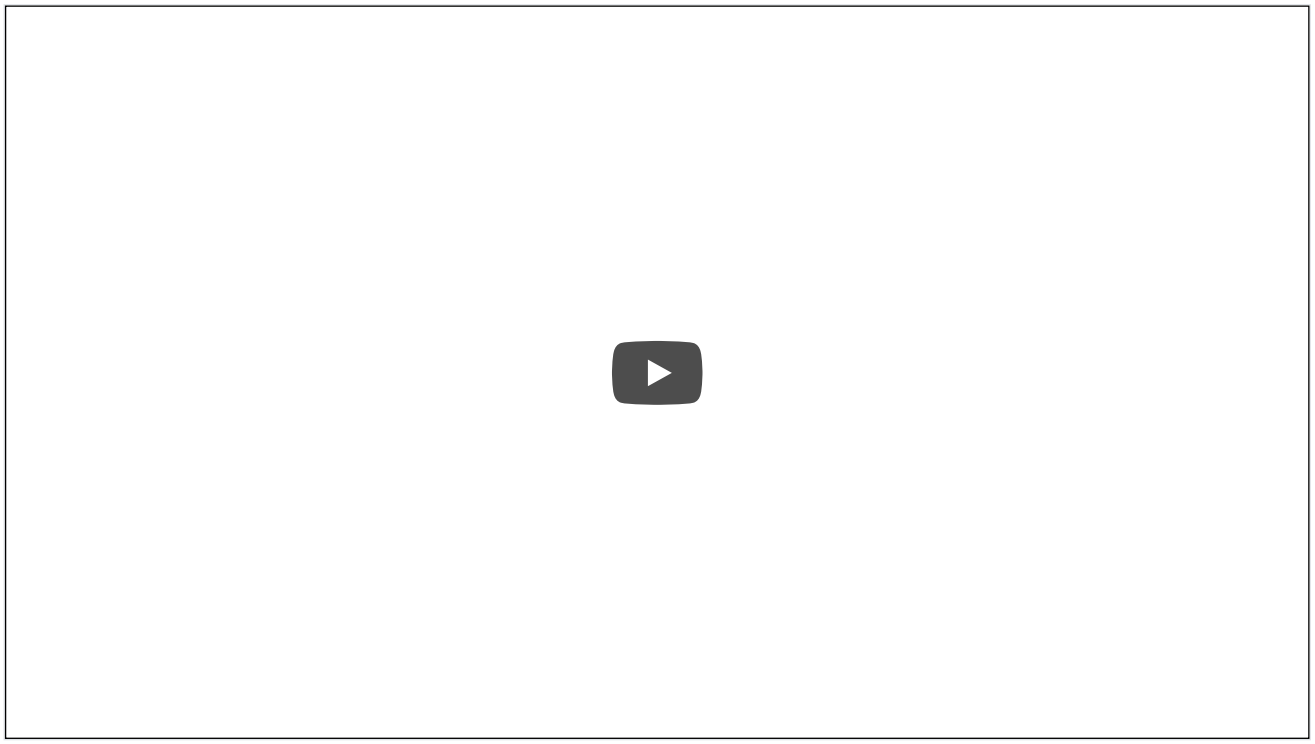
  - The first parameter, `GET`, specifies what the action we're trying to do here, which is just getting something. The next one, `/`, stands for the root, or the top-most directory. Finally, `HTTP/1.1` is the version of protocol we're asking to use. We also specify the host, or the website, since the same server might be able to handle multiple, and there's also additional information in a request that are less important.
- [10:15] The response we get back might look like:

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

  - Here we get an HTTP status code of 200, which means "OK", and then a line describing the type of content. HTML, Hypertext Markup Language, is a format that webpages use to markup content. Finally, we'll get the actual data for the page.
- [11:40] Other common status codes include 404, for a page not found, and 500 for an internal server error, where the server itself had an error trying to respond.
- [13:05] We can open Google Chrome, and open the Developer Tools panel. In the Network tab, we can load a site, and see lots of requests. At the very top, we can see the original request for `google.com`, and we'll see the Request Headers that we sent, and the Response Headers we got back. In fact, the first response we got back was `HTTP/1.1 301 Moved Permanently`, to `http://www.google.com`, since by convention URLs for websites start with `www`. Next, we get redirected to `https://www.google.com`, with the more secure, encrypted

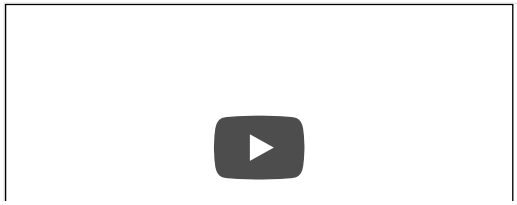
version of HTTP. In this response, we finally get a `200 OK` code and some content to load the page. Later, we'll be writing our own server programs that return these codes and content in response to requests from browsers.

▼ **HTML**



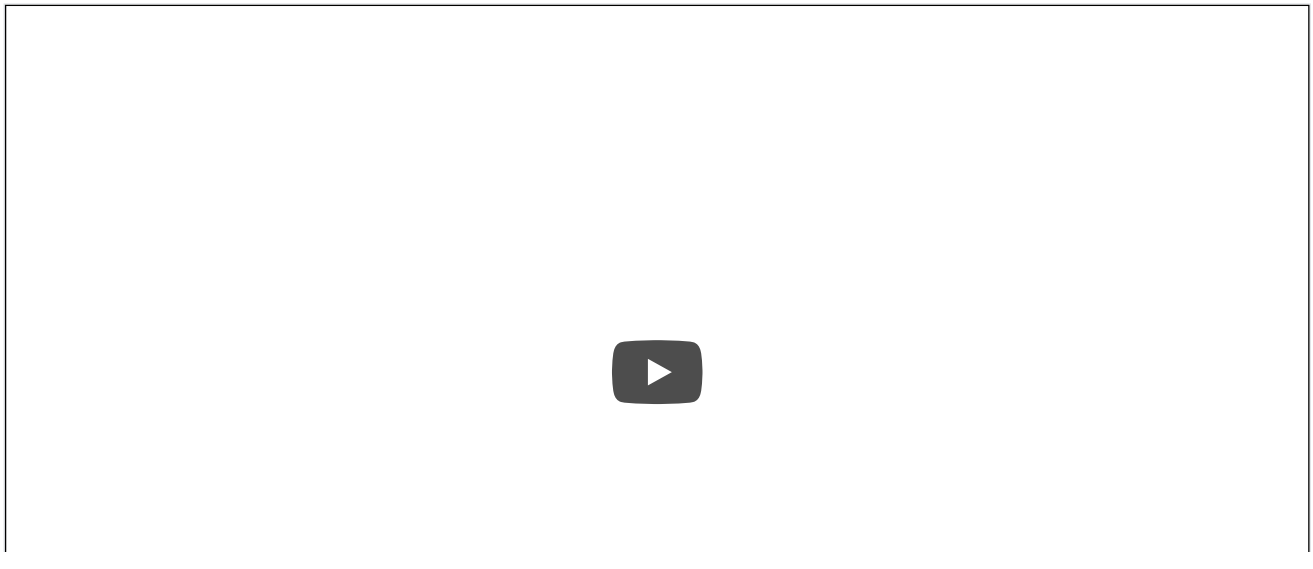
- Now that our computers can communicate over the internet, we can take a closer look at the actual data we get back. In Chrome, we can go to View > Developer > View Source, to see the HTML, Hypertext Markup Language, that makes up the text-based content of a webpage.
- [1:30] We'll look at a simple HTML page, where we first declare to the browser of the version and format of the page. Then, we have a tag, `<html>`, which starts the HTML content. Generally, HTML is made up of lots of nested tags that map to a tree structure, with opening tags and closing tags that determine the structure of the page. Next we have the `<head>` tag, which includes metadata, data about the page, such as the `<title>` tag inside that defines what the title of the webpage will be, as displayed in the tab of the browser. After, we have the `<body>` tag, which contains the visible content displayed by the browser.
- [6:00] In the CS50 IDE, we can start by writing this code in a file called `index.html`. And the CS50 IDE has a built-in server we can use. In the terminal, we can run `http-server`, and there will be a URL for our IDE's server that we can open. Then, we'll see the files in our IDE, and we can open `index.html`. We can change our file, save, and refresh to see what it looks like.
- [10:20] We take a look at an example where we use an `<img>` tag to display an image. Here, we add attributes, or additional parameters to the tag, like `src="cat.jpg"` to indicate that the source of the image is a file called `cat.jpg`, and `alt=""` to indicate alternative text for the image. And the `<img>` tag doesn't have a closing tag, since it doesn't make sense for there to be other tags inside the image.
- [13:30] We add links to go between pages with the `<a>`, or anchor, tag. Notice that we can have any text for any URL for our link, so we should pay attention to the URL we end up at.
- [18:00] We can add additional elements, like paragraphs with the `<p>` tag, headings with `<h1>` or `<h2>`, or tables with `<table>`.
- [22:35] We'll add aesthetic styling like borders and colors later, but we can think about HTML as describing the structure of the content of our webpage.
- [22:55] We'll add a `<form>` element with some `<input>` elements where we can get some information from the user. Finally, we can redirect ourself to Google's search page for whatever we typed in, by using `https://www.google.com/search`. We noticed that `https://www.google.com/search?q=cats` takes us to a search page for cats, and the `?` indicates some HTTP GET parameters, where here we have a `q`, or query, parameter, with the value `cats`. So our form can have an `action` that submits our text input with `name="q"`, to `https://www.google.com/search`.
- [29:35] There are so many more HTML elements. We can likely find an HTML tag that lets us add a particular feature, just by searching Google for relevant documentation.

▼ **CSS**



- To style webpages, we'll use another language, CSS, Cascading Style Sheets.
- [0:40] First, in our HTML, we'll need to add a `style` attribute to a tag, and set the value to something like `style="color: blue;"`. The key-value pairs in the style will change how the browser displays the element. In fact, we can add a style to the `<body>`, and all the elements inside the body will inherit the style unless they specifically have a different style.
- [5:20] We can also change the alignment, like centering or right-aligning text, or the font size. We can add multiple properties by separating them with semicolons.
- [8:40] We might have multiple elements of the same type, like `<h1>`, and we can add a common set of styles in the `<head>` element with the `<style>` tag. In that tag, we can specify that all `h1` elements share some set of styles.
- [14:00] If we want set the same styles to multiple types of elements, we can add classes, which we can think of as names, to any number and type of element. We'll do this by adding the `class="title"` attribute, with a class name of our choosing, to elements we want to style the same way. Then, in our CSS we can select all elements with the class with `.title`.
- [18:25] We can create another class, and even give the same element multiple classes with `class="title green"`, and the styles for both will apply.
- [20:40] We can include CSS in a separate file, like `styles.css`, so all of our webpages can share the same styles. We'll use a new tag, `<link>`, to link a file to our HTML page. And we can include many different CSS files, each of which having some subset of styles.
- [24:00] With CSS, we can also style tables in HTML by selecting the `table`, `tr`, and `td` classes. By looking at CSS documentation online, we can figure out what styles will give us the border styles we want.
- [27:40] We can add padding, or spacing, within each table data cell. And we can select the first row by adding a class like `header`, or use a special table header cell element `<th>` that we can select precisely.
- [31:05] It turns out that there are lots of CSS libraries, written by other people, that will include styles for common elements that can quickly apply a theme or aesthetic to our HTML. Bootstrap is one such popular library, and its documentation will include a `<link>` element we can add, such that our page will use Boostrap's CSS files. The documentation will also show us various components we can use, and classes we can use to style them easily. A `<div>` element in HTML is like a generic container or section, so we'll see that commonly used for elements that don't have a more semantic HTML tag.

▼ **JavaScript**



- To build a more interactive website, we'll need a programming language that will allow us to run code on the browser that changes how it behaves with our webpage, beyond just the content and style. The language that we'll use is JavaScript, a language that browsers can interpret and run, with syntax similar to that of C.
- [0:35] We take a look at syntax for declaring and changing variables, conditions, loops, and functions.
- [5:00] A simple webpage has elements that we can represent as a graphical tree, where each nested element is a child of a node in the tree. This is called the Document Object Model, and JavaScript can manipulate, or change this, without having to refresh the page.
- [7:15] We'll add JavaScript to our page with a `<script>` tag inside our `<head>` tag. We can call a built-in function, `alert()`, to show an alert on our page. After we save our file, we can run a server in our IDE with `http-server`, and see our page.
- [9:20] We can add a form, and have our form call a function and `return false;` to stop any default behavior after our function is called.
- [12:00] Our form can have a text field, and our JavaScript button can get its value. Fist, we need to add an ID to our element with an attribute to the element, like `id="name"`. And in Javascript, we can use `document.querySelector('#name')` to get that element by its id.
- [17:25] We can change our alert to display something else with a condition.
- [18:45] Instead of just reading the content of the DOM, we can also change the contents of elements by setting their `innerHTML` property, after selecting them with `document.querySelector`.
- [22:00] We'll look at another example that has a counter, or a variable that we can increment by pressing a button.
- [24:25] It turns out that we can even change these variables or call these functions in our browser, with View > Developer > Developer Tools in Chrome. In the Console tab, we can type in JavaScript code, and it will run in our page. If our JavaScript code has errors, those errors will also show up in the console.
- [26:00] We can dynamically change the style of the page. We'll create three buttons, each with a unique `id`. And in our script tag, we'll select each button, and we'll set their `onclick` property to a function that our browser will call when the button is clicked. We can create an anonymous function, or a function with no name, directly with `function() { ... }`, instead of defining it separately first. And in our function, we can select the `body` tag by type since there's only one of them on our page, and set the `style.backgroundColor` property to a color.
- [30:25] It turns out that we can't add the `onclick` function in the beginning of our JavaScript code, since our browser interprets the code from top to bottom, and our code can't find the buttons. There are a few ways to solve this problem, but for now we can simply move our `script` tag to the end of our `body` tag.
- [33:55] The `onclick` function is an event handler, or a function that is called when an event happens. There are many such events that we can listen for, like a change to the selected option in a dropdown menu. We'll look at another example, where we add `onChange` to a `<select>` element. Here, inside our event handler function, we can use `this.value()` to get the value of the option that was just selected. We can think of `this` as a special variable that contains some kind of context for how a function is called. In this case, `this` is the event that triggered our event handler.
- [39:20] We can update our page periodically with `window.setInterval`, which calls a function for us at some interval of time. We'll create a function, `blink()`, that will change the `body`'s visibility to be either `visible` or `hidden`.
- [43:10] We can also create a separate file like `blink.js`, where we only have our JavaScript code, and include it in our HTML file with `<script src="blink.js"></script>`.
- [44:45] Finally, we can ask the browser to give the user's location to our JavaScript code, with `navigator.geolocation.getCurrentPosition`. The argument we pass in is a callback function, or a function that will be called by the browser when the `getCurrentPosition` finishes running. Inside our function, we'll just write the coordinates we get to the page.
- [47:05] With JavaScript, we can read and write to the DOM, and take advantage of even more features that browsers provide.

▼ **Homepage**

- Our first assignment will be to create a homepage of our choice using HTML, CSS, and JavaScript.
- We'll create four different pages in HTML, each linked to one another somehow. Recall that we can use the `<a>` tag, with the link to another file in our IDE.
- We'll also use at least five different CSS selectors, for five different types elements, classes, or IDs. And we'll want to use at least five different properties overall to style our page, and documentation online will help us find what we're looking for. We'll also use the Bootstrap library to style at least one of our components, so we don't have to write the CSS ourselves for that.
- Finally, after we've written the content for our pages and styled them, we'll use JavaScript to make our page interactive somehow, through alerts, buttons, dropdowns, forms, intervals, or even more.
- Be as creative as you'd like!

▼ **Flask**

- So far, we've learned how to write webpages that are saved as a file and returned by an HTTP server. But we can also have web servers, or applications, that generate content dynamically before returning it as a response.
- [1:00] We'll use a framework in Python called Flask, which allows us to write a web server with many features. We'll create a new folder in our IDE, called `hello/`, and create a new file called `application.py`. By reading the documentation and experimenting, we can write our first Flask application which returns something for the `/` route. And in our terminal, we can `cd` into our folder and run `flask run`, which will find our `application.py` file and run it. We'll open the URL, and see our returned string.
- [4:10] We'll add another route, `/goodbye`, and a function that returns different content. We can return any content we want in our routes.
- [6:00] It turns out that Flask allows us to use template files, or files with HTML that are like format strings, with some parts that are the same every time, and some parts that will contain variables that we can substitute in. The `render_template` function in the Flask library will allow us to use templates and plug in variables like ``.
- [10:35] We can generate a random number, for example, and display it each time our page is loaded. We can use `control + c` to stop our server, and then restart it, to make sure any changes we make are reloaded. And once we load our page in the browser, we can view its source to make sure that Flask substituted our variable as we expected.
- [13:25] We can add conditions to our templates, with `if ...`, so depending on the value of our variables, we can return different content

entirely.

- [16:25] We can even write a form that our server can accept, with another route that the form can submit to. Then, in that route, our server can receive and use the form data. We write a form that has a name input, and write a route function that gets the input with `request.args.get()`, and returns a template with the input substituted in.
- [21:30] We see an Internal Server Error, and in our terminal we see the error that `request` is not defined, and it turns out that we need to import it from Flask. We try again, and see that the GET parameters in the URL changes based on what we submit in the form.
- [24:00] We can add additional logic in our route to handle the case where `name` is empty, and return a different template.
- [26:00] It turns out that we can have templates for our templates, since many of our pages might have similar HTML code around its content. We'll create `layout.html`, and add a special block inside the `<body>` tag. Then, our other files like `index.html` can use the template with `extends "layout.html"`, and only have the content block for the `body`.
- [30:35] And we can add additional blocks, like for content we would want to have inside a `<style>` tag in the page.
- [32:20] We'll start writing a new application by creating a new folder called `tasks`, and creating an `application.py` file. Inside, we'll create routes for `/` to list tasks and `/add` to add a new task. We'll create a `templates` folder with a `layout.html` before, a `tasks.html` showing a list of items, and a `add.html` that includes a simple form. We'll have our routes render each of these templates, and set our form to use a new method, `POST`, to send the form's data back to the `/add` route. Our `add()` function can then either display the form for a `GET` request, or create a new task for a `POST` request.
- [42:30] We can create a global variable, `todos`, to store a list of task names that we can display later. In our `add()` function, if we get a `POST` request with some data, we'll add the new task name to our list on the server, and redirect back to the default route, which will show a list.
- [44:15] And in our `tasks.html` template, we can loop over our `todos` list variable with `for todo in todos`, and create a `<li>` element with the contents set to each item.
- [48:00] We can also make sure that the task name is not empty, by adding some JavaScript code that only enables the submit button if the input field's value is not empty. Otherwise, we disable the submit button. We do this by adding an event handler to listen to the `onkeyup` event for our `task` input, which is triggered by the browser every time the user presses a key and releases it.
- [52:40] But our task list goes away when we stop and start our web server, since we initialize our `todos` variable to an empty list each time. Next, we'll use a database with SQL to store and modify data.

▼ **Databases**

- So far, we've learned how to write a server that can respond with webpages that are the same for every user. But there are websites where we can log in, and it will show us information specific to us.
- Recall that cookies are small files that websites ask our browser to store on our computer, with some kind of identifier that our browser shows the website the next time we go there, so the website knows who we are. This allows our server to have sessions, or data for users' interactions with a website, specific to each of them.
- [1:20] We'll look at the task list application we made last time. Since our task list was stored in a global variable in our server application, everyone who visits our page will see the same list.
- [2:40] To solve this, we can use sessions from Flask, by importing and initializing their implementation. By doing so, our `tasks()` function

can look in the global `session` variable, and read, set, or update a `todos` key within it. Flask will take care of making sure that the global `session` variable is actually specific to the user who made that request, by storing and checking some cookies.

- [7:30] If we want to store more complex data, it would make more sense to use a database instead of session objects. So we'll create a new application to store registration information, like names and emails.
- [9:25] We'll make a new empty file, `lecture.db`, and run `sqlite3 lecture.db` to create a table and set column names and types for the data we think we'll need.
- [11:00] In `sqlite3`, we can run queries to select or insert into the table to check that everything works. In our new Flask application, we'll import the SQL library from CS50 so we can work with our database more easily, and establish a connection to our `lecture.db` file. In our `/` route, we can run a `SELECT` query to get the rows from our `registrants` table, and pass them into our template. Our template will in turn iterate over each row, and generate an `<li>` item with the values of each column in each row.
- [17:35] Once we have our index route, we can add more rows to our table with the `sqlite3` prompt, and see our server return the new data.
- [18:05] We can add a new route to our application that will insert new data, too. In our `register()` function, we can return a `register.html` file with a form that has the inputs we need, and ensure that the form submits to our `register` route with the `POST` method. Then, in our `register` route, we can check for a `POST` request, insert the data from the request into our table, and redirect to the main route. In our SQL query, we'll be careful to substitute our variables safely with the `db.execute` function, instead of combining the strings ourselves, to avoid SQL injection attacks.
- [23:05] We'll try out our application, and everything seems to be working as we expect. To improve the design of our server's code, we'll factor out some common template code into `layout.html`, and create an `apology.html` page where we'll tell the user an error message if something in their form is blank.
- [28:40] Now we can write Flask applications to read and store data in a database, saving our data efficiently for the long term.

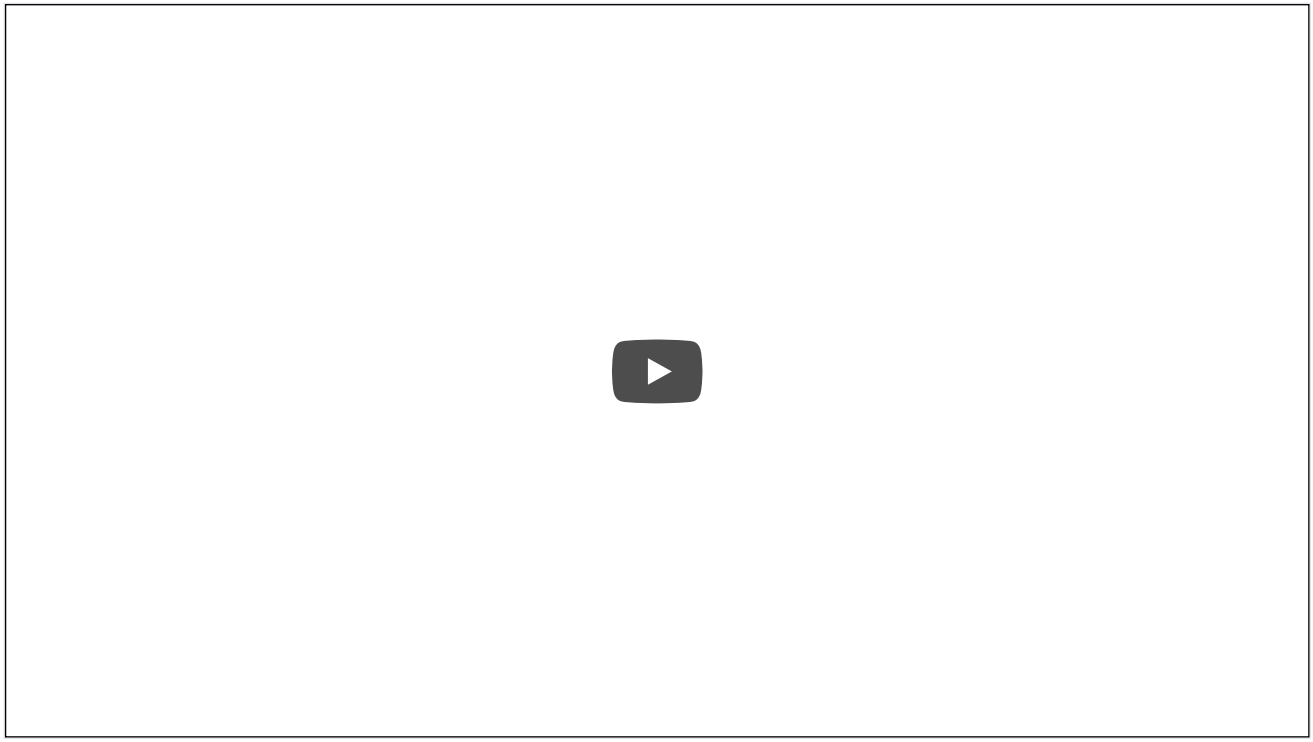
▼ **Finance**

- We'll take the concepts we've seen to create CS50 Finance, a virtual stock trading website with an account for users to register for, the ability to get quotes for shares of stocks and to virtually buy or sell them. We'll also have a history page for each account to see what we've done in the past.
- [2:45] We look at the distribution code for CS50 Finance, or the code that we'll all start off with. We have an `application.py` file that our Flask app will run, with various configuration options, a connection to a database file `finance.db`, and routes for . This follows the MVC, Model-View-Controller, pattern, which generally separates the concerns of data and how that's stored (our database), the views that display some amount of data (our templates), and controllers that control the logic of what is displayed when (our `application.py` routes).
- [4:45] Since we're using a third-party API, or Application Programming Interface, some code that someone else wrote designed for us to use, we'll also need an API key to get stock information.
- [5:30] Notice that our routes also have a `@login_required` decorator, or extra attribute in Python to indicate that the function should behave differently. Flask allows us to automatically redirect users to a login page, and we have the login functionality implemented in our distribution code too. The `/login` route checks whether a matching user and password exists in our database (for a `POST` method, as from the login form), or displays the login form for a `GET` method. And in our database, instead of storing the user's raw password, which

is more insecure since hackers might use them against other websites, we store the hash of their password which is sufficient for verification, but difficult from which to recover the original password.

- [14:30] After the `login` route we have `logout`, which just clears the session, and we have `quote`, `register`, and `sell` routes left to implement.
- [15:10] We'll implement:
  - `register` so we can register for a new account
  - `quote` so we can get a price quote for a stock
  - `buy` to buy some shares of a stock
  - `index` to show the stocks in our account
  - `sell` to sell some shares of a stock
  - `history` to show transactions in the past
  - and a personal feature of our choice
- [15:55] We talk about the requirements for each of these routes, and how they might be implemented with conditions based on the request's method, and either display forms or perform some action after validating the request.
- [20:50] We have an existing `finance.db` database, and we can use `sqlite3 finance.db` to run queries that add columns or tables that we might want to use to store additional data to support our routes.
- [23:00] `index` will query our database for a user's stocks and their cash balance, along with using an API to get the current price of each and displaying all this data with a template. `sell`, too, should have validation and update our data in the database.
- [25:25] Finally, we might need another table (in our database) to support our `history` page, and display the data for each user's transactions in a table (in our template).
- [26:25] And we'll need to add a personal touch, whether that's allowing users to change their password, add cash, or additional features.

▼ **Conclusion**



- In this track, we learned about how computers communicate over an internet, structured web pages with HTML and styled them with CSS, and added some interactivity with JavaScript. Then we learned how to write a web server application with Flask, that can dynamically generate web pages and use a database to read and write data.