## Video 1: What is HubL, and when would you use it?

Hey there. My name's Jeff, and I teach web development for HubSpot Academy. Today, we're going to talk about HubL. Unless this is your very first exposure to the HubSpot CMS, you've probably come across articles or videos that mention HubL but shy away from a comprehensive explanation. Well in this video, we're going to dig in and talk all about HubL from the ground up.

Let's start out with a basic definition: HubL is a markup or templating language. These are common in CMSs and most programming frameworks where templates are involved. It's a simple language that gives quick, standardized access to data without having to use a full- fledged programming language like PHP.

This is a nice approach because you get access to common programming constructs like variables, conditional logic, and functions, but there's a clear line between templating and backend logic.

We'll talk a bit more about what HubL is and what it isn't in just a moment, but let's briefly discuss where you'll encounter HubL in the HubSpot CMS.

To some degree, HubL is optional in the HubSpot CMS. If you're building drag and drop templates, you won't need to write markup code because the drag and drop builder writes it for you.

That being said, even if you're exclusively using drag and drop templates, you'll probably want to build your own modules at some point, and HubL is an important part of that process.

If you're building coded templates, especially blog templates, you'll use HubL extensively. To sum up: HubL isn't necessary for basic, out-of-the-box usage of the CMS, but as you begin to build more custom CMS assets, you'll begin to use more HubL.

Ok, so let's dive a little deeper on the technical details of HubL. HubL is essentially a subset of jinja, which is a templating language for python. The exact details are a bit more complex, but that lineage is sufficient for a fairly deep application of the language. HubL doesn't support all of jinja. It includes a lot of its own markup, especially variables and functions that are specifically related to the HubSpot CMS.

The relationship between HubL and Jinja can be handy though. Sometimes the Jinja docs can be a good secondary reference for more esoteric uses of HubL. Also, it can be helpful to familiarize yourself with python data structures since they're similar to other languages like Javascript but not exactly the same.

So we've sketched out what HubL is, let's be clear about what it isn't. HubL isn't a programming language. It's a markup language. CMSs like Wordpress have a somewhat ambiguous boundary between templating and backend logic. This ambiguity opens the door for a wide variety of approaches to building templates. There are advantages and disadvantages of this flexibility that we won't get into here. As a general rule, the HubSpot CMS is more prescriptive. If you reach the limits of what's possible with HubL and the CMS, HubSpot provides APIs for creating more customized solutions.

OK, most importantly, HubL isn't client side. Sometimes folks will try to pass data back and forth between HubL and JavaScript. It's important to remember that HubL is executed before JS, so you can pass data from HubL to JS, but never the other direction.

Finally, like most coding languages, HubL is a fairly deep topic. It's pretty easy to get started, but you'll want to get familiar with the documentation to answer complex questions as they arise. Also, in addition to the documentation, there are various developer tools and an active community of users to collaborate on projects big and small. Check out the resources tab for links.

## Video 2: Fundamentals of HubL

Let's dive into the syntax and programmatic constructs of HubL. We'll be covering common programming elements like variables and loops here. If you don't have experience with these elements in other languages like javascript, python, or php, you should supplement this content with more in-depth instruction on these topics. Check out the resources tab for some suggestions.

OK, let's take a look at HubL syntax first. All HubL code is written within curly brace delimiters. These come in three flavors: statements, expressions, and comments. Statements are written with a curly brace and percent sign. Statements do something like define a variable or loop through a list, and expressions are written with double curly braces. Expressions evaluate to a value– for instance, outputting a blog author. Comments are written with curly braces and the pound symbol. Comments are ignored by the templating engine, and since HubL is executed on the server, they won't be output into your rendered source code. This can make them a preferable alternative to client side comments.

OK, now let's take a look at some typical programmatic constructs and how they look in HubL.

Let's start at the very beginning with variables. HubL variables hold data like strings, numbers, and booleans. You'll use statement syntax to create and assign them and expression syntax to output from them. There are a number of predefined variables available as well. Check out the resources tab for links to these.

In addition to simple data types, HubL also has lists for storing sequential data. List items are accessed by position or iterated through using loops.

Finally, HubL also has dictionaries for storing data in key/value pairs. Much of the data that's built into the CMS is stored in dictionaries. Dictionary items are accessed using dot syntax.

OK, beyond data storage, HubL also has Macros which are similar to functions in other languages. You can pass data to macros via arguments to repeat blocks of statements with small variations. Notice the syntax for opening and closing a macro. This block syntax is used throughout the language.

Conditional statements in HubL use traditional comparison operators that are common to most languages, but there are also a number of sophisticated expression tests that can save you a ton of coding and make statements much easier to read. Check out the resources tab for links to documentation.

Finally, loops in HubL use the for syntax to loop over lists or dictionaries. In addition to built-in loop properties for common tasks like detecting the index and first and last cycles, there are also filters available to modify iterable objects on the fly. Filters are covered in more detail in another video.

OK, before we wrap up this overview of the language, let's revisit something I mentioned about dictionaries and predefined variables.
One of the most useful tools for writing HubL is something called developer info. This is available via the sprocket menu on any published page.

This tool outputs a big JSON tree of all the data available on the page. It's output as JSON in this tool, but this data is stored in dictionaries, so you can access it via dot syntax in HubL. Some of the most important bits of this are documented, but seeing everything all at once in context can be invaluable for complex projects. Once again, check out the resources tab for detailed documentation on developer info.

## Video 3: Module markup in coded templates

OK, let's take a look at module markup in HubL. If you're building coded templates instead of drag and drop templates, you'll need to use tags to add modules to your templates instead of dragging them out of the inspector. There are two flavors of this syntax:

Basic module syntax and block syntax.

The basic module syntax is a single line. This line starts with the name of the module, then the name of the instance inside quotes, and then a comma separated list of parameters. This is essentially the same as tags in HTML. The instance name is roughly equivalent to an html id. It needs to be unique on a page in order to map data added via the page editor to the appropriate module in the template.

There is also a block syntax for modules. With this syntax, you can list attributes as blocks instead of simple name/value pairs. It can be much easier to write and read when you're dealing with large chunks of data. In this example, the HTML attribute of the rich text module would be extremely unwieldy written in one line.

There are many module tags available. These tags are documented along with their unique parameters and code snippets inside of the design manager for quick reference. Also, check out this video's resources tab for links to further documentation.

In addition to each module's unique parameters, there are a handful of parameters that are supported by every module. One of these universal parameters is worth a special mention here:

no_wrapper is a boolean and when set to true, no wrapping HTML will be added to your module. This can be very handy for controlling your markup in coded templates.

So to quickly recap, module markup is fairly straightforward. The specific syntax for marking up individual modules can be found in the in-app reference materials, but you'll want to decide for yourself which parameters are appropriate for your needs, and which flavor of markup to use to keep your templates clear.

## Video 4: What are HubL filters, and how are they used?

Most aspects of HubL are loosely equivalent to conventions in HTML and Javascript. One notable exception is filters. HubL filters handle a wide variety of tasks in both statements and expressions on many types of data. Filters are used for everything from formatting a date object, to casting data types, to filtering sequences.

Many filters are shortcuts for more complicated loops and conditional statements. This makes a lot of sense considering HubL is a templating language. The main job of HubL is to flow data in a template as quickly and clearly as possible.

So let's cover a couple of filter examples to get a feel for the syntax and usage.

To start, let's take a look at getting the length of a sequence. The syntax for this varies delightfully from language to language. In HubL this is achieved with the length filter. In this example, we create a sequence of three strings. Then we use the length filter to access the length of this sequence. Filter syntax in HubL is the pipe character followed by the name of the filter. In this case, the output will be 3.

To take it up a notch, let's look at an example of HubL in CSS. The convert_rgb filter outputs a hex value as its equivalent rgb value. So in this example, there's a brand color variable specified in hex that might be used throughout the CSS file. Then that same variable is used for this semi-transparent link hover state. Since an rgb value is needed for this CSS property, we can convert it with this filter on the fly and leave the initial variable in hex format.

OK, to take it one step further, filters can also be used to quickly create subsets of data. In this example, we've created a list of books with attributes for title and a boolean to keep track of whether or not we've read this book. Then we use reject attribute to create a subset of books where the read attribute is false. Then we loop over those unread books instead of looping over everything.

Filters like reject attribute can be very handy for working with large data sets, for instance in blogs or HubDB where extra loops and nested conditionals can make code unwieldy and unclear.

To build upon this example slightly, it's also possible to apply the filter when writing the loop like so. The filter gets executed first, and the loop only runs on the returned subset of data. So these two examples of the reject attribute filter return the same result.

Finally, there's a debugging filter named pretty print that's extremely helpful for getting info on variables. Use this filter in an expression to print the data type and all data contained in the variable. This can be a handy way to inspect variables you're not familiar with, such as the ones built into the CMS. In this example, we're printing everything contained in site settings. We're not showing the result in this case since it's a pretty huge chunk of data, but give this a try in one of your own templates. Along with developer info, pretty print can be a great way to familiarize yourself with all  the data available on a page.

So those are just a few examples to give you an idea of what's available. Check out the resources tab for a link to the documentation. As a general rule, if you're trying to figure out how to manipulate some piece of data in HubL, you should check to see if there's a filter available since they can make quick work of things that might otherwise consume a lot of time.

## Video 5: Building a CSS rem macro with HubL

Ok, let's take a look at an example of HubL in action. In this example we'll build a macro for converting pixel values to rem values in css. This macro will make our css a bit easier to read and save us from having to calculate rem values every time we want to use them. Even if you're not a fan of using rem units, this example will give us a chance to see some basics of HubL and then take things up a notch to explore further.

So first off, here's how we'd like to use this basic rem converter. We write a css rule, and call our rem macro with a nice simple pixel value. The macro takes that value, calculates the rem equivalent and outputs it like so with the unit suffix.

So to build this macro we'll start with declaring the macro and the argument it will take, in this case the single value we'd like to convert.

Now inside this declaration we need to take this value, calculate the rem equivalent, and print out the result. To calculate the rem value, we need to know the baseline font size. Let's create a variable at the top of this macro to hold this number. This way it's obvious what this number is doing in the macro and if we want to change it later on or hook it up to some other variable, we can easily make that tweak. Let's use 16 here.

Ok. Now we need to perform our calculation. Let's create another variable for this to make this as obvious as possible. Rems are expressed as a ratio of the baseline font. So we'll set remValue equal to the value passed to this macro divided by the baseline font size.

Now we simply need to print out the calculated number and tag on the unit suffix. We'll just use expression syntax to output the value of our variable and write rem in plain old text. This is where macros shine. Unlike functions in other languages, any text we include that isn't part of HubL is simply spit out as-is. Remember, HubL is a templating language, so the primary purpose of macros is to help us construct HTML and CSS.

Ok, let's check out a little more HubL by making this macro more flexible. We'll expand this to output more than one value. This will make it easier to use this macro with shortcut css properties. Here's how we'll use our expanded rem macro. Instead of passing a single value, we can pass an list of values to the macro.

So here’s the finished macro from before. Our updated macro will have quite a few more moving pieces. Let’s get set up for the new code by modifying what we’ve got here so far. First, let’s make the argument plural since we’re now passing in multiple values. This isn’t a technical requirement, it’s just a good idea to make this more readable. We’ll leave the baseline font size variable alone since that will be used the same way it was before, but we’ll delete the line where we calculate the rem size. We’re going to rewrite that a bit differently in a moment.
Ok. We’ve got this prepped for our new code. Let’s get started. Our new macro is going to need to take a list of values, convert them to rem values, and then print them out in the correct CSS syntax. There are a few ways we could accomplish this, but in this example we’ll take the opportunity to demonstrate a couple of powerful features of HubL.

Let’s start by creating a new list to hold the finished values. This will be empty for the time being.

Now we'll create a loop to loop over the values passed to this macro.

Inside this loop, we'll create a temporary variable to hold our calculated rem value. We'll also concatenate the rem suffix here with the tilde character.

Now, on this next line, we'll use the do statement and the append function to add our computed value to that cssValues list. This is the syntax for modifying lists. You can also modify dictionaries with a combination of do and the update function.

Ok, now our loop has populated the cssValues list with all of our calculated values, let's spit them back out into the CSS. Below the loop, we'll use an expression to output that cssValues list. In order to properly format this list, we'll use a very handy filter. The join filter concatenates list items and builds a string. It takes a separator as an optional parameter. We'll use a coma and a space here to format this for CSS.

And there we have it. A simple macro for converting values from pixels to rems. There are quite a few things we could do to expand this. There aren't any failsafes for auto or inherit values for one thing. Take a look at the resources tab in this lesson for a link to a rem macro that approaches this a bit differently and accounts for more complex scenarios.