# **Learning Julia**

## **Julia: Power like Python, speed like C**

### **Julia: Power like Python, speed like C**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Joe] If you look at the history of programming, selecting language to use for a particular project has often required trading off performance versus flexibility. The introduction of Julia, however, just might turn that tradition on its head, at least for some scenarios like scientific and financial computing. Hi, I'm Joe Marini, and I've been building software professionally for companies like Google, Microsoft, and Adobe for more than 30 years. In this course, we're going to learn about Julia, a relatively young language that has been gaining rapid acceptance in fields like data science and financial modeling, among others. The Julia language gives you performance approaching that of statically typed, stodgy, old C while maintaining the kind of flexibility you typically get from Python. In this course, we'll get an overview of the language and some of its basic features, such as how to control program flow, organize information using the built-in data structures like sets and dictionaries, and how to use some of the standard library modules that come with Julia that enable common scenarios like generating random numbers or working with date and time information or working with the file system. Let's get started working with Julia.

### **What you should know**

Selecting transcript lines in this section will navigate to timestamp in the video

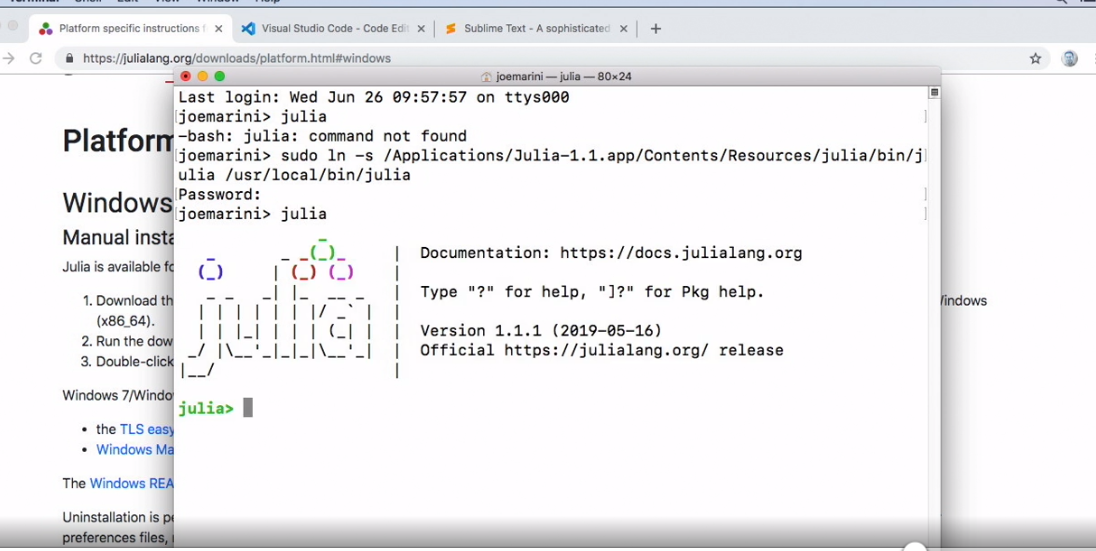
- [Instructor] There are a few things that you should already know before beginning this course. This course is intended for people who already have some fundamental knowledge of programming and want to learn the Julia language. It's not intended to be a starting point for absolute beginners. You should already be familiar with some of the main concepts of writing programs, such as writing functions, declaring variables, using control structures like loops and conditional statements, and so on. If you're just starting out learning about programming, then you should probably watch Programming Foundations: Fundamentals and you might also want to watch Programming Foundations: Algorithms. You should also be familiar with using a text editor to write code. It doesn't matter whether you want to use VS Code, or Sublime, or Atom, whatever, it doesn't really matter. We're just going to be working on text files in this course, so whatever text editor you are comfortable with that's fine with me as well. In this course, I'm going to be using Visual Studio Code because it's a great free editor from Microsoft and it has support for Julia through its extensive ecosystem of extensions. You can download it from code.visualstudio.com and it works across multiple operating systems, Mac, Windows, Linux. So it doesn't really matter what kind of computer you have. You should also be familiar with using your favorite operating system's terminal program in case you want to run your code directly from the command line. If you feel comfortable with these prerequisites, then you are ready to start the course.

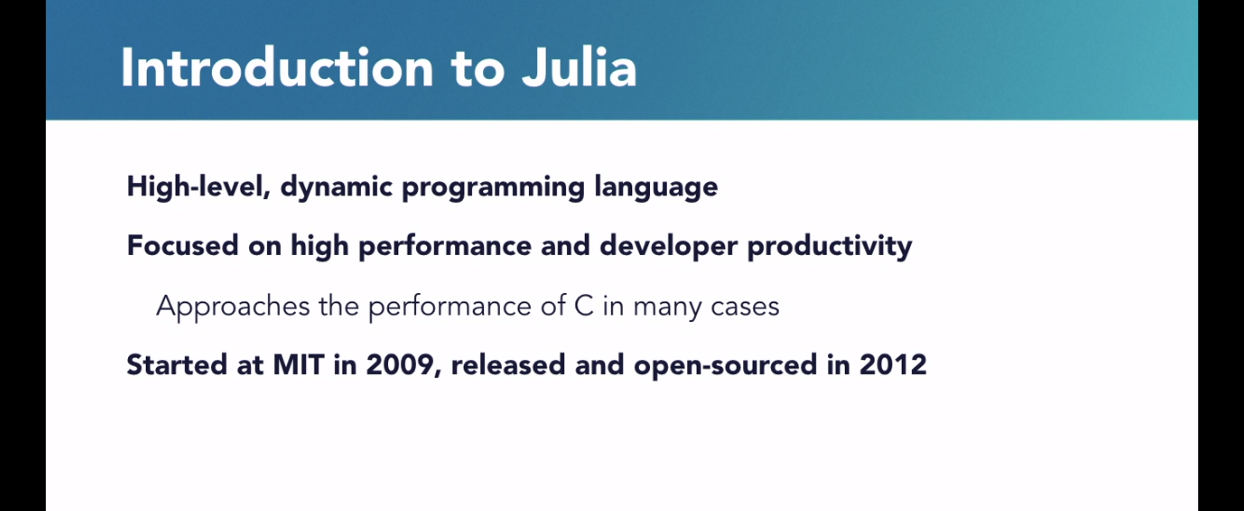
### **Setting up the development environment**

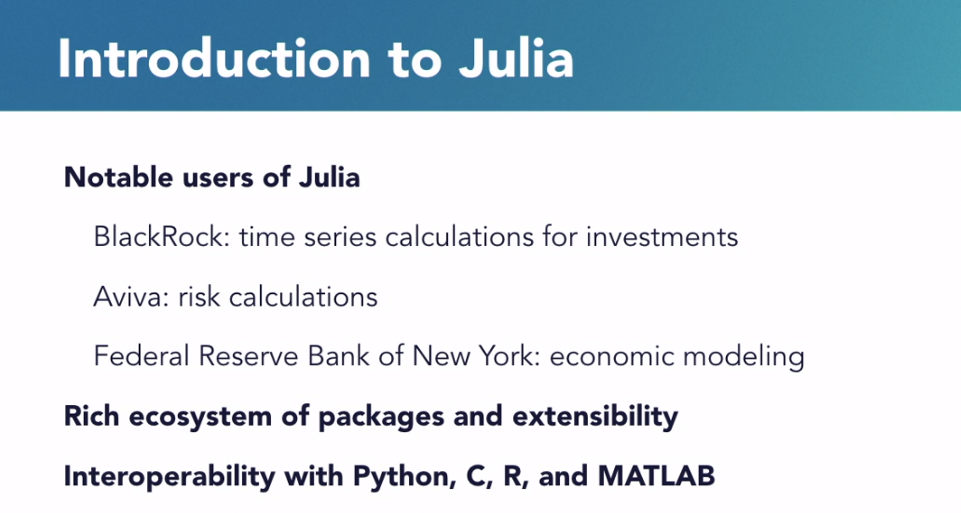
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] To set up your Julia development environment, let's start by downloading and installing Julia itself. The Julialang.org website has a download section where you can get the latest version of Julia for your operating system. So I'll go ahead and click on the Download link and you can see that there are versions for Windows, Mac, some Linux binaries, and you can even get the source code if you want. So I'm using a Mac, so I have downloaded that particular version and when you download the installer, go ahead and run it for your operating system and that will get Julia installed. Now if you're using a different operating system than I am, just follow the instructions on the site for your OS and you can see, for example, that for Windows if you click on the Help link that will take you to a platform specific instructions page for how to get Julia installed on your computer. So, I've run the installer already and I have Julia installed, but I want to be able to run Julia from the command line, but my application folder is not a part of the path environment variable. So let me show you what I mean. I'm going to start the terminal and when I type the command Julia, you can see that I'm getting an error. It says that the command is not found and that's because my environment path does not point to the Julia application. Now there's a couple of ways you can fix this, but what I'm going to do is make a symbolic link to the Julia app and put the link file in a place where my path currently points, which in this case is going to be usr/local/bin. You can do the same thing on Linux if you need to and on Windows, the installer should take care of the path variable for you, but if it doesn't you can refer to the platform specific instructions on how to fix that. So to make the symbolic link on my Mac, here in my terminal I'm going to type sudo and then the command for making the link is ln and I'm going to make a symbolic link, so that's -s. Then I have to point to the Julia application and that is in my Applications folder and then there's the Julia app and the actual executable is inside the Contents/Resources/julia and then bin and then the app name is Julia and then I'm going to create the symbolic link inside /usr/local/bin and I'll just name the link Julia. And then I'm asked for my password so I'll put that in and now it's been created. So now watch what happens when I type the Julia command. You can see that Julia starts up and we're using Version 1.1.1 here. All right, once you've done this you need to make sure that you have your favorite text editor ready to go. So two of my favorites are Visual Studio Code and Sublime Text. Now if you're using a different editor, that's fine. It doesn't really matter since we're just going to be editing text files and we'll be executing the code from the terminal window. After you download and install Julia, put the Exercise Files somewhere on your computer where they will be easy to access. So I put the Exercise Files here on my desktop, but that's not required. You can put yours wherever you like. So once you have all these pieces in place, it's time to start the course.

Create symbolic link



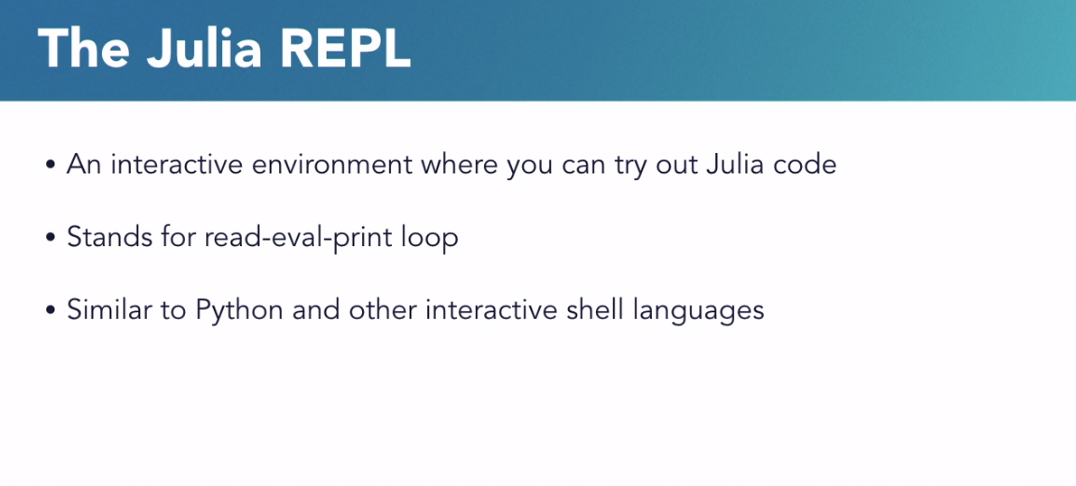




### **The Julia REPL**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Most of the programming work you will do in Julia will take place in an IDE, but Julia does provide an interactive environment where you can try out small pieces of code. This is called a REPL, which stands for the read, evaluate, print loop. Essentially, you type in Julia commands, the REPL executes the code, and then you type in more commands. If you've done any programming in a language like Python, then maybe you've seen this before. And just to be clear, this is not the best way to write a lot of Julia code at scale. You will usually use an IDE for that. This type of interactive shell is useful for learning about different parts of Julia or just for trying out small pieces of code that you want to test. We're not going to be making heavy use of the REPL in this course, but I wanted to demonstrate it so that you can try it out on your own time. To start the environment, open up your terminal program and if you've set up the path correctly on your computer, you can type the command, Julia, and that will start up the REPL environment. And then we can start trying out some code. So let's just try a simple expression. I'll just type one plus two and you can see that the result there is three. I can also use some Julia functions. So let's try out the print function and you can see the result there. To get the result of the previous expression, I can use the ANS keywork, which stands for answer. So let's try this again. I'll type two to the fifth power. Right, and that's 32 and then I'll type ans, and you can see that that gives me the previous answer. In fact, we can even write our own functions. So let's type a function definition and we'll get more into this in the course. I don't expect you to understand all of this right now, but we will cover this later. So let's go ahead and create a function and I'm going to write, helloworld, and when I hit Enter you can see that the REPL is waiting for me to type the function body. So I'll go ahead and do that and I'll just use print and I'll just say hello world and then to close off the function, I need to use the end keyword. And you can see now that I've created a function definition where it says helloworld, generic function with one method. So, I can now call that function. I can just say, helloworld, and you can see the output there. So the REPL environment can also be used to learn more about Julia itself. If I type the question mark character, you can see that the prompt has changed to help mode and I can enter the name of a function. So let's enter the name of the sort function and you can see that that prints out the help info for using that particular function and I can scroll back up through the output and I can learn about a particular function I might want to use in my code. I can also use the shell feature, which lets me type a shell command without leaving the REPL environment. So if I type a semicolon, you can see the prompt change to the word shell and then I can use a shell command like, on Linux anyway, I can type ls. If you're on Windows you might type dir and you can see that I get a directory listing when I do this. So this is pretty useful if you're working in the REPL and you want to execute a shell command, but you don't want to have to lose all the environment context or spin up a separate terminal just to do that. So to exit the REPL, we use the exit function and you can see that I'm now back out to my shell in my terminal environment. Okay, so that's a brief introduction to the Julia REPL environment. During the course if you find yourself curious about a particular subject and you want to experiment further, I recommend just firing up a Julia REPL shell and using it to explore some of the language features.



### **Building HelloWorld.jl**

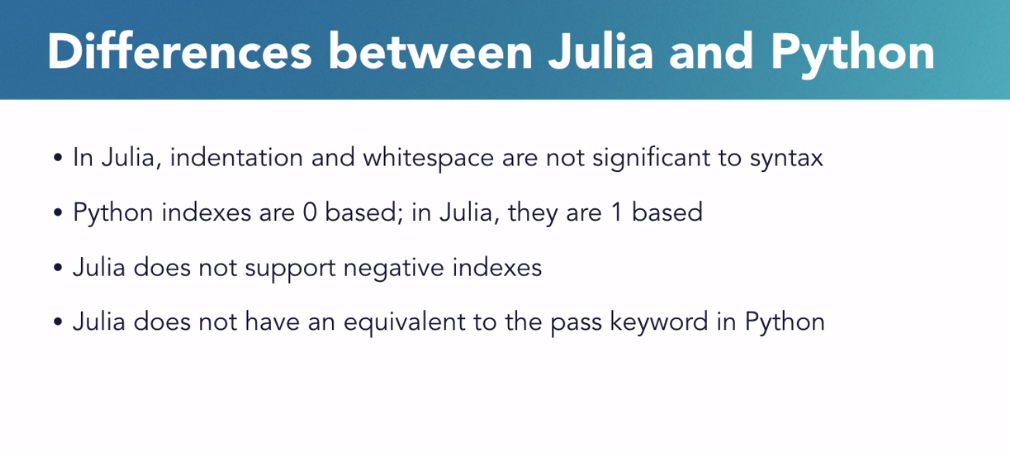
Selecting transcript lines in this section will navigate to timestamp in the video

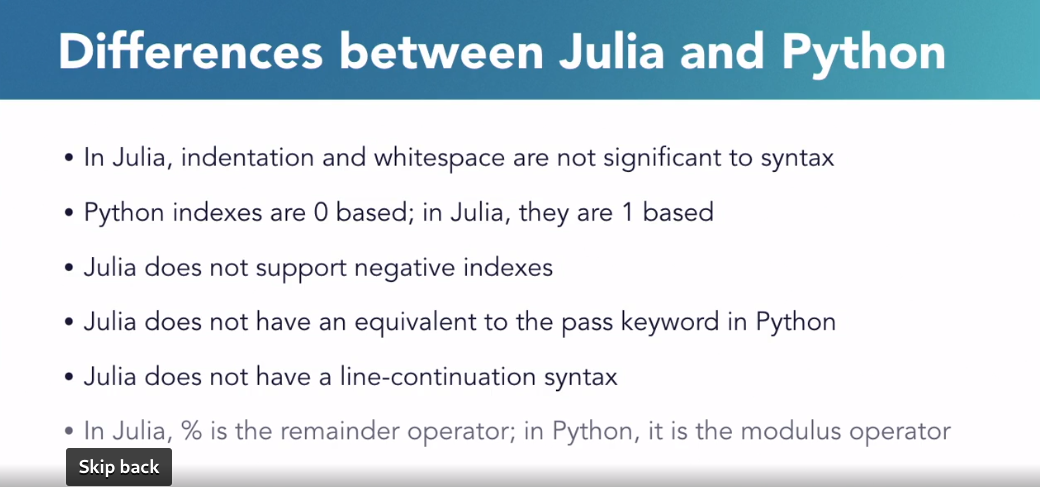
- All right, let's build our first Julia program. And in keeping with the grand and glorious history of the computer industry we're going to call our program Hello World. So, before I do that actually, I want to point something out. If you're using VS code the way I am you should know there's an extension available in the Extensions Marketplace for Julia. So, if you go to to the Extensions Marketplace and you just search on the Julia keyword it should be one of the first ones that come up and you can see right here that I've already got it installed. And it provides some really nice features like syntax highlighting and some code snippets. So, if using VS code, I recommend going and getting this extension. All right, let's go back to the code and we're going to open up our Exercise Files folder. Okay, and you can see here I've created a file named HelloWorld and I have little dot J-L suffix on there that indicates it's a Julia program. So, I'm going to open that up. So, now I'm going to write some Julia code. And, I don't expect you fully follow everything here just yet. As we go through the course, all of this will become more clear. So first, I'm going to create a function and to do that I use the Function keyword and then the name of the function. And I'm going to call it helloworld. And this syntax is very similar to languages like Python or JavaScript so far. And then, inside the function I'm going to use the println statement to print the words Hello World! and then, I'm going to end the function with the End keyword, so that closes off the function. And then, to make this work, I need to just call the function, so down here I'm going to type in helloworld with the parentheses and that will call the function. Okay, so, all of this is pretty straight forward. So, I'm going to save this and then, let's go to the terminal and run the program. So, here in my terminal, I'm going to make sure I'm in my Exercise Files folder, and I am. And then, I'll go into Chapter One. And then, I'll type Julia, and then HelloWorld dot J-L. And there, you can see the output. So, the println statement is running and we're printing out Hello World. All right, so Julia has some interesting features that may be a little different from some other languages that you've used before. One of these features is that you can define multiple different methods for a given function. So, let's see how that works. Let's go back to the code. And I'm going to create another function definition for hello world. And this time, it will take an argument. So, I'll write function helloworld and then, inside the parentheses I'm going to give it an argument. And I'm going to annotate that argument with a type to indicate that it is an integer. And inside the function I'm going to print helloworld the number of times that the argument represents. And, again, I'm going to write some code here that you might not fully understand just yet, but bear with me. So, I'm going to write "for i in 1:count" "for i in 1:count" Then, I'm going to write println and it's going to be Hello World and then I'll use the i-variable and then I'll use the End keyword to close off the foreloop and then another End keyword to close off the function. So now, I can call the function with no arguments and I can call the function that takes a number. So, let's try that. So, let's go back to the terminal and run this again. So now, I'll type Julia helloworld dot J-L and you can see the output of both functions. So, here's the one with no arguments and then, the one with arguments executed three times. So, essentially, Julia figured out which version of the function I was calling by looking at the arguments and then invoking the proper method of that function. Now, if I try to call a version of the function using an argument that I haven't defined yet that's going to result in an error. So, let's try that. I'll go back in here and type helloworld and I'll call it with a string. All right, so let's go back to the terminal. And now, let's run the program again. And you can see that I got an error indicating that there isn't a method of helloworld that operates on a string argument. So, this feature is called multiple dispatch and it's just one of the powerful features of Julia that we're going to learn more about as we go through the course.

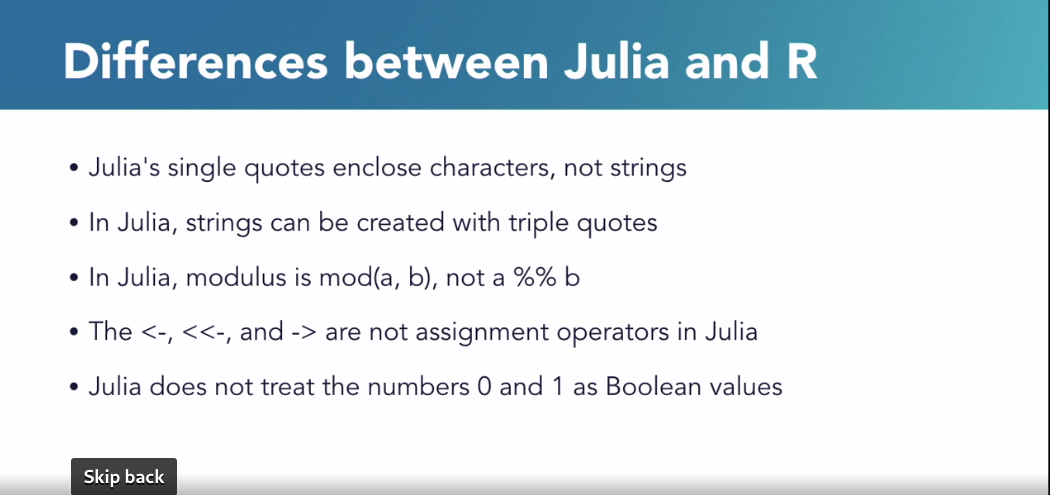
### **Building HelloWorld.jl**

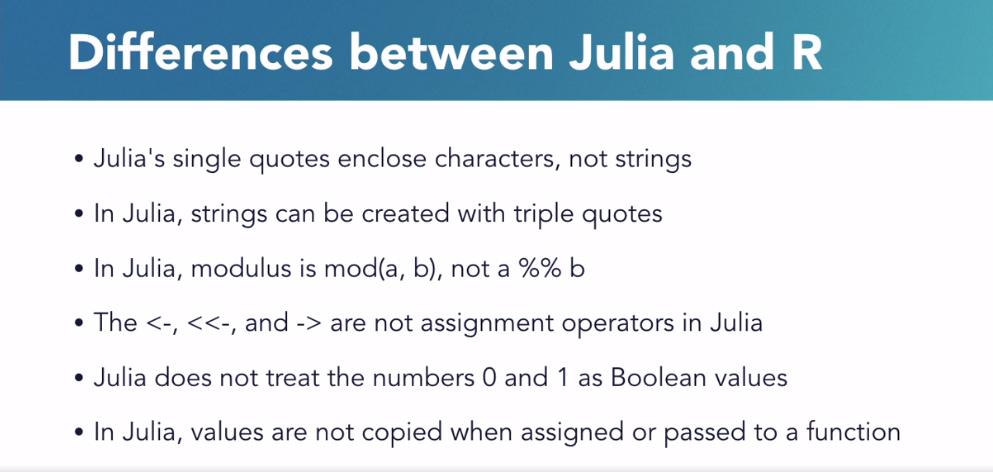
Selecting transcript lines in this section will navigate to timestamp in the video

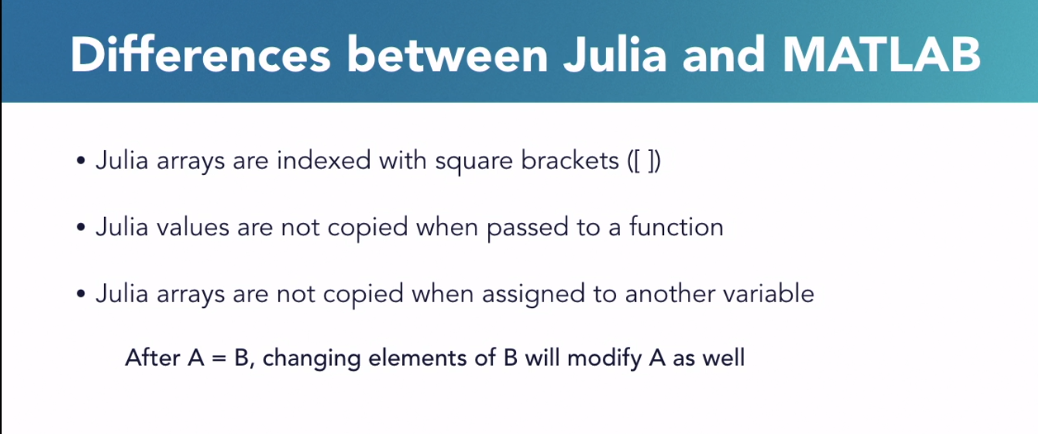
- All right, let's build our first Julia program. And in keeping with the grand and glorious history of the computer industry we're going to call our program Hello World. So, before I do that actually, I want to point something out. If you're using VS code the way I am you should know there's an extension available in the Extensions Marketplace for Julia. So, if you go to to the Extensions Marketplace and you just search on the Julia keyword it should be one of the first ones that come up and you can see right here that I've already got it installed. And it provides some really nice features like syntax highlighting and some code snippets. So, if using VS code, I recommend going and getting this extension. All right, let's go back to the code and we're going to open up our Exercise Files folder. Okay, and you can see here I've created a file named HelloWorld and I have little dot J-L suffix on there that indicates it's a Julia program. So, I'm going to open that up. So, now I'm going to write some Julia code. And, I don't expect you fully follow everything here just yet. As we go through the course, all of this will become more clear. So first, I'm going to create a function and to do that I use the Function keyword and then the name of the function. And I'm going to call it helloworld. And this syntax is very similar to languages like Python or JavaScript so far. And then, inside the function I'm going to use the println statement to print the words Hello World! and then, I'm going to end the function with the End keyword, so that closes off the function. And then, to make this work, I need to just call the function, so down here I'm going to type in helloworld with the parentheses and that will call the function. Okay, so, all of this is pretty straight forward. So, I'm going to save this and then, let's go to the terminal and run the program. So, here in my terminal, I'm going to make sure I'm in my Exercise Files folder, and I am. And then, I'll go into Chapter One. And then, I'll type Julia, and then HelloWorld dot J-L. And there, you can see the output. So, the println statement is running and we're printing out Hello World. All right, so Julia has some interesting features that may be a little different from some other languages that you've used before. One of these features is that you can define multiple different methods for a given function. So, let's see how that works. Let's go back to the code. And I'm going to create another function definition for hello world. And this time, it will take an argument. So, I'll write function helloworld and then, inside the parentheses I'm going to give it an argument. And I'm going to annotate that argument with a type to indicate that it is an integer. And inside the function I'm going to print helloworld the number of times that the argument represents. And, again, I'm going to write some code here that you might not fully understand just yet, but bear with me. So, I'm going to write "for i in 1:count" "for i in 1:count" Then, I'm going to write println and it's going to be Hello World and then I'll use the i-variable and then I'll use the End keyword to close off the foreloop and then another End keyword to close off the function. So now, I can call the function with no arguments and I can call the function that takes a number. So, let's try that. So, let's go back to the terminal and run this again. So now, I'll type Julia helloworld dot J-L and you can see the output of both functions. So, here's the one with no arguments and then, the one with arguments executed three times. So, essentially, Julia figured out which version of the function I was calling by looking at the arguments and then invoking the proper method of that function. Now, if I try to call a version of the function using an argument that I haven't defined yet that's going to result in an error. So, let's try that. I'll go back in here and type helloworld and I'll call it with a string. All right, so let's go back to the terminal. And now, let's run the program again. And you can see that I got an error indicating that there isn't a method of helloworld that operates on a string argument. So, this feature is called multiple dispatch and it's just one of the powerful features of Julia that we're going to learn more about as we go through the course.

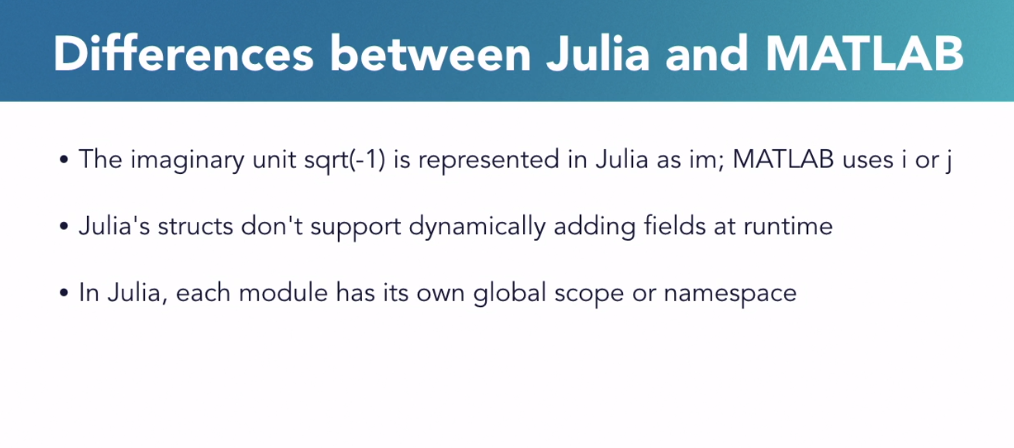












### **Julia variables and data types**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter, we're going to learn about some of the basics of the Julia language and we're going to start off with an overview of Julia variables and data types. So, I'm going to open up the variables\_start file here in my editor and, okay, I'm just going to scroll right to there. So, there are five basic data types in Julia. There are numbers such as integers and floating points, there are Boolean values of true or false, there are strings and individual characters. The integer and floating point numbers have some additional subtypes and we'll look at those a little bit later. Now, Julia is a dynamically typed language, similar to JavaScript and Python. And what that means is that you don't need to declare the type of a variable before you use it. I can just assign a value directly to a variable and Julia will infer what type that variable should be. So, you can see I've already created some variables here. I have an integer, a floating point, a string, a character and a Boolean. So, variable names must begin with an underscore or a letter and can contain letters, numbers and certain symbols. The convention is to use lowercase letters in a continuous fashion although, if the name is hard to read, you can insert underscores to make it easier but that's not really common. So, let's run the code that we already have here. So, we can go out to the terminal and in the terminal, I'm going to make sure I'm in the proper directory. So, I think right now I'm in chapter one. Let's go up to chapter two. Okay, there we go. And I'm going to go ahead and run Julia and it's going to be variables\_start and there you can see the output. So, we've got the various data types being output and by the way, if you're using VS code and you've got the Julia extension installed, you can actually run the code from right in here. So, on a Mac, I'm going to type Command + Shift + P and on Windows, this is Ctrl + Shift + P and if you type in Julia, you'll see that there's an execute file command right here but if you don't see that, just type in Julia and then execute and you can see that there's an execute file, execute code and execute code block commands. If you choose execute file, you can see that right here in VS code in the terminal window, you can run the Julia code. If you're not using VS code, just continue to use your system's terminal. So, there's a couple of things to point out here about variables. So first, in Julia, you are allowed to redefine variables. So, I can redefine the first variable from an integer to a float with no problems. So, I can do something like, first equals 1.0 and then I can print line first. Second, it's important to note that strings and characters are different from each other and you have to use the right quotes for each one. I can't, for example, use single quotes to define a string. So, if I comment this back in and I try to use these single quotes to define a string and I try to print it out, so, when I run this, you can see that there's an error. So, strings need to be in double quotes. It's also possible to define constant values so, let's go back to the code. So, you can define a constant value with the const keyword and by convention, these are all caps so, I'm going to just make a long name here so, I'm going to say const meaning of life and we're going to assign that to the number 42. And let's go comment out some of these statements and as you can probably tell, the hash character is used to comment out code. So, I'm going to just go ahead and comment some of these guys out. And we'll just comment these guys out as well so they don't interfere with us. All right, so here, I defined a constant. Now, what's interesting is that, I can actually redefine a constant although, that produces a warning. So, if I say, meaning of life is equal to 43 now, okay, what I'm going to do is run this so, I'll save and back in the terminal I'll run this and you can see that it says, warning, redefining constant, meaning of life. However, I cannot redefine a constant to be of a different type. So, let's go back to the code. So, if I try to say meaning of life is equal to 43.0 so, let's save that and let's run it again, well, now, you see I'm getting an error. So, you can't redefine constants to be of a different type and the reason for this is that the compiler can do some optimizations when it knows that it is dealing with a value that won't change type. If you try to change the type, that throws everything off. All right, one last thing I want to demonstrate is that you can annotate variable definitions with type names to help the Julia compiler optimize the code. Now, I have to do this in a function because as of this recording, Julia doesn't yet support global variables that have type definitions. And we will look at functions in more detail later in the course but for now, just bear with me as I do this. So, let's go back to the code and back in the code, I'm going to go ahead and comment some of these guys out. Okay, so, here's what I'm going to do, I'm going to make a function and my function is going to be named testfunc and inside the function, I'm going to write X and then two colons and then the name string. And I'm going to define a string just, I'll give it some text. Okay. And then, I'm going to try to redefine this variable. I'm going to say X is equal to 10 and then I'll just try calling this function so, I'll call it testfunc, all right. So, I'll save. Now, let's go ahead and try to run this code. So, back here in the terminal, let me clear the screen and we'll run this again and now you can see, I get an error when I try to assign a value that doesn't match what the declared type of the variable is. All right so, quick introduction to variables in Julia.

### **Numbers in Julia**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Okay, in this example we'll see how Julia works with different kinds of numbers. So, I'm going to open up numbers\_start. So, Julia defines integer subtypes that have different byte sizes for both signed and unsigned integers. So, there's Int8, Int16, Int32 and so on. And there are unsigned versions of these as well. So, for example, I can specify that I want a 16-bit signed integer by making a variable. And I can say Int16 and then give it the value I want. In this case, I'll just use 1. And then if I print out the type of that variable. Or I can have Julia infer the type when I assign the value. So, for example, I can make a variable b and I can say it's an Int of 2000. And then I can do the same thing with the print line and the type of. So, Julia will assign an integer type based on whether the system you are working on is either 32 or 64-bits. Now, I could do the same thing with unsigned integers, which cannot be negative. It causes an error if you try it. So, I won't try it. If you want to try it, it's up to you. So, I'm going to make a Uint16 of 1. And then we'll just print the type of that as well. All right, so let's go ahead and try these three statements out. So, I'm going to Save. And in my terminal, I'll just make sure I'm in the right folder, which I am. So, I'll type julia and then numbers\_start and you can see that in the output there, I've got different types. So, here is a Int16, which is the first variable. And then there's an Int64. So, I want a 64-bit system. So, the inferred type of the variable with no type is Int64. And then you can see that I defined another UInt16. All right, let's go back to the code. So, sometimes it's useful to know what the minimum and maximum value is for a given type are. And you can use the typemin and typemax functions for this. So, for example, I'll print out the typemax for an 8-bit integer. And then I'll do the same thing. So, I'll Copy and Paste this a couple times. So, we'll use typemax for an Int32. And then, let's just try an Int64. And instead of typemax, use typemin for that one. Okay. And you can also use the word size property to see what kind of system your code is running on. So, I'll use the println function and I'm going to call on the Systems package. And I'm going to ask for the word size. All right. So, let's go ahead and comment out this previous code. Okay. So, let's try this out. So, we'll run that. So, here are the two max values for an 8-bit signed integer. What's the second one there? That's a Int32. So, this assigned Int32. And this is the minimum value for a signed 64-bit integer. And you can see that the output of the word size property is 64-bits. So, if you try to assign a number that is too large to fit in a given type, that's going to throw an error. So, let's clear this. And let's go back to the code. So, let's try this out. I'll say a = UInt8 and I'll give it a number like 300. Okay. So, let's go ahead and comment these guys out right here. So, now when I save and run this, you can see that there's an error that it's unable to truncate a value of 300 to fit in an 8-bit unsigned integer. There are special values to represent infinity and values that are not a number. So, once again let's clear the terminal and let's comment this out. So, let's try doing some mathematically silly things. So, I'll try 1.0 divided by 0.0 and I'll Copy that and Paste it. Let's also try dividing 1.0 by the special word infinity. And then, let's also try dividing 0 by 0. Okay. There are also special functions to produce zeros and ones for any given number type. So, for example, I can use the zero function to ask for a zero value for the type of a Float64. And I can do the same thing. In this case, I'll ask for the one value of, let's do something else, let's use UInt32. All right. So, let's Save that. And let's go back to the terminal. All right, let's run that code. And you can see that one divided by zero is infinity. One divided by infinity is zero. When you try to divide zero by zero, you get not a number. And then we have the output of the zero and one functions.

### **Strings in Julia**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Okay, now let's take a little bit of a closer look at some of Julia's string types, and some related operations you can do on strings. So, first, remember that Julia provides separate data types for characters and strings. Characters are defined using single quotes like this. So I can say my char is equal to X. You can also convert character values to integers and we'll take a little bit of a closer look at casting values a little bit later in the course. But, for the moment, let me just show you that you can do things like this. I can say, get me the integer value of what my character is, and you can convert the other way as well. So, for example, I can print out the character value of, say, the number 120, okay. So let's quickly run that, so I'll save, and then in the terminal, I'll just go ahead and run "Julia strings start." And you can see that the value of the X character is 120, and then I can go back the other way to get the lowercase X. Okay, let's go back to the code. So remember that strings in Julia are defined with double or triple quotes, and you can see that I've already created two sample strings using each one of these methods. So, once you have a string, there are some common operations that you can perform on it, so first, keep in mind that strings are encoded as UTF-8, unless they only contain ASCII characters, in which case, they are ASCII encoded. And you can also encode them as UTF-16 or -32 if you need to, there's functions to do that. So, to get the length of the string, you can use the length function, so I'm going to print out the length of my str... And you can also access individual characters. Now, a key difference with Julia, remember, and I mentioned this earlier, is that strings start with index of one, not zero like in other languages. So to get the first character, you would use something like this. I would print out my str and that would be suffix one. To get the last character, you can use the special index value of end, so I can do something like this... And the end value will get me the last character, and you can also perform calculations on that value. So, for example, I can do something like end minus three, all right. So, let's go ahead and try those examples out and I'll comment out those guys right there, so let's save and then let's run this again. Right, and there you can see that the length of the string is 32, and then we're accessing the individual characters using those square brackets. You can also slice and extract substrings from strings in a pretty straightforward way. So, what I can do is I can use the print line function, and I'll use my string, and inside the square brackets, I can give a starting point and then separate it with a colon and then do something like end minus one, so this would get me the substring for the my str characters, so this would get me from H up to what looks like I. So let's go ahead and run this, and let me comment out some of my previous work. All right, so let's save and let's try it out, and sure enough, you can see there's the substring right there. So, to iterate over characters, you can use a loop construct, so let's go back to the code. Now we'll talk about loops later in the course, but for now, I'll just show a simple example. Let me comment this one out. So I can, for example, say "for C in my str." And then I can print C. And then I close off my for loop with an end statement, and then I'll just print out a blank line. We can also concatenate strings using the star character, so if I have one word and that is "hello," and I have another word and we'll use "world," right, I can then do something like this, where I can have my variable and then a star character, and then I can give it another substring, so we'll put a comma and a space, and then another star with the second word, and then another concatenation with a period. All right, so let's run those two examples. So we'll save and let's run, and you can see that, right there, there is my sample string in Julia, and it's being printed out by iterating over each character, and notice I'm using print instead of print line, so that prints out each character and advances one position. It doesn't print out a character return. And then, I've got my concatenation where I'm building "Hello, world" and you can see, here in the output, that my concatenation put together the words and the punctuation. All right, so one more thing to demonstrate, and that's going to be string interpolation. So let's clear the terminal and let's go ahead and comment out these previous examples. All right, so let's define two variables. I'll have A is equal to five and B is equal to 10. What I'm going to do is have a string and I'll say the result of dollar sign, A, plus dollar sign, B, is, and then I'm going to enclose that inside parentheses, and I'm going to say A plus B. So let's go ahead and save, and back in the terminal, we will run that. And so you can see that, in the string, we've replaced the variable references with values, and so that's called string interpolation, where you can embed little expressions inside strings and have Julia execute them.

### **Built-in functions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Just like some other popular languages, Julia has some built-in useful functions. So let's take a look at some of those in this example. Go ahead and open up the builtin\_start file and let's begin by looking at some of the math functions. So you can use the round function to round numbers in a variety of ways so I'll go ahead and just use the round function to round this number and then we can print that out. I can also specify a rounding direction. So for example, I can write x = round and then same number but now I can specify a direction. So you can see I can choose round down, round from zero, so on. We'll go ahead and choose round up and then print that out. I can use the floor and ceiling functions to perform rounding operations as well. So if I have a value like this, I can print out the floor result of that, and I can also print out the ceiling of that, which will round up, and I can use a function like absolute value to print out the absolute value of the number, so I'll just write A-B-S and give the absolute value of that negative number. Okay, so let's go ahead and just run what we have here. So I'm going to save this and go to the terminal and let's just run builtin\_start and there you can see the effects of the rounding So this rounds down, then I have round up, and then I've got the floor and ceiling of that number and the absolute value of -15 is obviously 15. Okay, now let's look at some of the functions for output and input. So let's go back to the code and comment down our work so far. We've been using the println function in the course for quite a bit already, but there's also a print function that outputs a string without a carriage return. So if you want to progressively build up a string in the output, you can do it like this So I'll print hello, there and then I'll use the carriage return version. There's also a printstyled function that you can use to print text with formatting. So I'm going to make a variable here, just some text, and then I'll use printstyled to print out just the string by itself, and then I'll print out a space there, so just a blank line, and I'll copy and paste this. So, for the second version, I'll specify that the text should be bold, and then for the third example, I'll specify that the text should be bold and that we're going to give it a color, and this color I'm going to give it is going to be red, and there are some built-in symbols. I've got red, blue, green. There are different color symbols you can use. All right, so let's go ahead and try that out. Now, unfortunately, on the Mac that I'm using, the terminal, the built-in terminal, doesn't display colors or bold text. So I'm going to run this in the terminal window that comes with Visual Studio Code. So if you're using Visual Studio Code, and you've got the Julia extension installed, I'm going to type Command + Shift + P and that's Ctrl + Shift + P for you Windows users, although in Windows, the terminal does print color so you may not have to do this. So I'll choose execute file and there in the output you can see that here's the "Hello there world!" string that got built up using print and print line, and there's the printstyled with no styling information. Here's the bold text and here's the bold and red text. All right, so let's go ahead and keep on truckin' along. We'll go ahead and comment these out. To read input from the user, you can use the readline function. So what I'm going to do here is print, "What is your name?" And then I'll have a variable that calls the readline function, and then I'll just print out whatever string is given to us. All right, so let's go over to the terminal and run this. And you can see it's asking me for my name, so I'll put my name in, and it prints out what I've input. So it's an easy way to get input from the user. And then finally, let's take a look at some string testing functions. So I'm going to comment this code, and I'm going to un-comment the lines of code I already have in here. All right, so the isascii function can be used to see if a string contains only ASCII characters, and you can see that I have an example here with A, B, and C, and then I've got some Greek letters here. Those are clearly not ASCII. The isdigit function can check an individual character to see if it's a digit or not. So I've got two examples of that. And then there's an isspace function, which can check a character to see if it's a white space, so here I have a space character, I've got a carriage return, I've got a new line, and then a letter that's clearly not a space. Now, obviously, there's a lot more of these but let's just go ahead and give these a shot. So, save this, go to the terminal and run, and you can see that abc is ASCII, the Greek letters are not. 9 is a digit and A is not, and the space, carriage return, and new lines are all considered white space and the A is clearly not. Now, obviously there are a lot more built-in functions than these, so I would suggest taking a look at the Julia docs and trying out some of your own.

### **Vectorization operator**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] When working with sets of data, it's fairly common to have to apply a transformation to all of the data items in a collection. And some languages provide a map function for this but Julia actually has a built in operator that performs this function as part of the language itself. It's called the vectorization operator and sometimes it's just known as the dot operator because of the syntax. So let's see how this works. And I'm going to open up the vector\_start file. So here I have an array of some data and we're going to learn more about arrays later in the course, but for now just know that an array is a list of data items. So let's suppose I wanted to multiply each of these data items by two. Now I could do that by looping over each item in the array and multiplying each one, but it's much easier to use the dot notation. So here's what I'm going to do. I'm going to write result is equal to array1 and then I'm going to put a dot operator and then the operator I want to run is going to be the multiplication operator and then I'll choose the number two. So what this basically says is take this array and then apply this operator to each one of the items that's in the array. That's what this little dot operator does right here. And then I'm going to apply this, this is the operation and then this is the value of the operation I'm going to apply. So let's go ahead and print out the result after I do that. So let's save and let's go over to the terminal and now let's run that. So I'll type Julia and vector\_start. And now you can see the original array and there's the output array where each value has been multiplied by two. Now what's interesting about this is that you can use this notation with Julia functions as well as operators, since operators in Julia are basically just functions themselves. So, for example, let's go back to the code. To calculate the square root of each item in R1, I can write something like this. So I'll use the square root function and then the dot operator and then I'll pass in the array. And then, once again, I'll just print out the result. And let's go ahead and comment out that previous one there. So I'll save. And then back here in the terminal, I'll run it again. And there you can see that the result array now has the square root value of each item. And this also works with your own custom functions. So, again, we're going to learn more about functions later, so bear with me as I write this last example. So back in the code I'm going to create my own function. And I'm going to create a new function named F that returns the triple of whatever its argument is. And I'm going to write this using a shorthand way of writing functions in Julia. And again, we'll see this later in the course, so don't worry too much about this. I'm going to say fx is equal to 3x. And then once I have that, I can use the dot notation. So I'll say result is equal to f and then array1. I'll have to put that in parentheses. All right. And then, once again, I'll just print the result out, so I'll copy and paste that line and then I'll comment out this. All right, so let's save and let's go back to the terminal and let's try that out. And there you can see the results, where each item has now been multiplied by three. So this is one of the great features in Julia that makes it really easy to work with collections of data.

### **Data type casting**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Since Julia has the ability to assign types to pieces of data, sometimes the need will arise to have to convert or cast one type of data to another. So let's take a look at some examples of doing this. I'm going to open up my convert underscore start file, and I have some variables with different data types already defined. So let's start with a couple of simple examples. So first, I can use the character constructor to convert an integer number to a character. So I'm going c is equal to Char and I can give it a number like 74 and then I can print that out and I can also convert the other way, I can take a character value and convert it to an integer type. So I just simply use the Int converter and I'll convert the J character and let's print that out as well. All right so let's go ahead and save and let's run this. So let's go ahead and run Julia and let's convert start and you can see that when I convert 74 to a character, I get the J and then the J becomes the 74 here. It's also possible to convert between numeric types. So let's go back and let's add some code to convert between floats and integers using the x and y variables I have here. So I'll make a variable Float one and I'll convert that to a Float32. So I'm taking an integer value and convert it to a Float32 and I can convert this floating point value to an Int16. All right so let's go ahead and comment out these previous examples. All right, let's go ahead and run this. Op, I had to print them out, that would be helpful. All right, let's run it now. And you can see that the integer has been converted to a float and the float is being converted to an integer. There are also functions for converting values from strings and as well as converting to a string. So let's give those a try next. And I'll comment these out. So let's parse this value here, the z value, so we have a string with 40 and a string with 30.0, so I can use the parse function and I have to tell Julia I'm going to be parsing an integer and I'm going to give it the z string and then I can do that again, I can parse and this time, I'll parse a Float64 using the f string, so those are these two values right here, z and f. And then just for good measure, I'm going to use the string function on the y variable which is this 24 and zero right here to convert that into a string and then I will print each one of this out so we print a and then I'll just copy and paste that and c. All right, so let's go ahead and run this. And you can see that in each of those cases, we have the results. So we got 40, 30 and 20. All right, one more thing to demonstrate. If you try to convert a piece of data that won't fit in the destination type, that's going to cause an error. So for example, let's go back to the code and let's comment these out. So for example, the character code of 281 is a real character, so I'll just go ahead and take that character code and we print it out and just to show that it's real, let's go ahead and run this. And you can see it's this little e with this cedilla on the bottom of it there. So let's go back to the code now. If I try to convert that character to a UInt eight, that's going to cause an error. So I'll cast this to a UInt eight and I'll try to print it out. All right so let's go ahead and run this code in the terminal. And you can see that it's causing an error because the value of 281 is too large to fit in an unsigned eight bit integer. So just going to keep in mind when you're casting and converting values in Julia.

### **Conditional statements**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] One of the key parts of programming is controlling the execution flow of your program. In this chapter we're going to learn about the different kinds of flow control provided by Julia. Now most of this is pretty standard, so if you are already familiar with other languages, then all of this will look very familiar to you. And we're going to start off with conditional statements. These are statements that evaluate a logical expression and then take action based up on the result. So I'm going to open conditionals\_start. So one of the most basic of these is the if statement. It takes a condition and executes some code if the condition evaluates to true. So to take a very simple example, let's imagine that we had a variable with a value like five and you can see I've got a variable named x there and we wanted to print a statement based upon that value. So I can use an if statement to do that. What I'll write is if x is less than 10, whoops, then print line x is small. And then we close off the if statement with an end. And that's pretty straightforward. But sometimes we want execute some code if a statement is not true. In that case we can add an else clause to the statement, in which case I will put that in here. And I'll write else print line and I'll say x is big. And as you can see, the end keyword is used to close off the functional blocks inside the if and else statements. So now I have a fully formed if, else statement. So if x is less than five than this code will run, otherwise this code will run. So let's go ahead and save and let's go out to our terminal. And let's make sure I'm in the right directory. So I have to go into my chapter four and then I'm going to execute the conditional\_start and sure enough you can see that when I run this that the x is small statement is the one that executes. So for more complex cases, you might want to evaluate several different expressions. To handle this case, Julia has the else if clause. So let's go back to the code and what I'm going to do is copy this and put it down here. And let's go ahead and comment this out. So in this case, I'm going to write, instead of else I'm going write elseif and then the elseif clause is going to be x is greater than or equal to 10 and x is less than 25, I'm going to print out x is medium. And then I'll have my else clause that just simply says x is big. So I'll copy that and paste it in there. All right, so now let's set x to be 17 and we'll run the code. So let's run this again. And now you can see that the middle print line is the one that runs. So Julia also provides an expression called the ternary operator, which can perform an evaluation in one statement. So let's take a look at that. We'll go back to the code. And let me go ahead and comment this out. So the ternary operator uses the question mark and colon characters. What I'm going to do is write a print line statement. And inside the print line statement I'm going to say if x is less than 10, then I'm going to write the question mark, and then say x is less than 10. So if this expression evaluates to true, then the statement after the question mark executes. Otherwise, I'm going to put a colon in here, and say x is 10 or greater. So in this cases I'm basically condensing the if else statement down to an question mark and a colon. So let's go ahead and try that out. So I'll save and we'll just leave x as equal to 17. So back to the terminal and let's run it. And now you can see that the second statement, which is the one after the colon, is the one that's executing. So conditionals are one of the core building blocks of programming. And Julia may not be as fancy as some other languages when it comes to the variety of conditionals that are supported, but if else gets the job done.

### **Loops**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Another very common task in programming involves iterating over multiple items of data in order to process each one. To accomplish this, we use a programming construct called a loop, and in Julia, there are a few different ways to construct loops, and we'll take a look at those in this video. So I'm going to go ahead and open up loops\_start in my editor. So let's begin with a basic for loop. So to create a for loop, I use the for keyword, and then I'm going to define a local loop variable. In this case, I'll call it i, and then use the in keyword, and then specify a range. So for a loop that goes from one to 10, and therefore iterates 10 times, I'm going to give it a one, a colon, and a 10 for the range, and then inside the loop, I'll just simply print out whatever the current value of i is. And then I'll use the n keyword to close off my loop. So that's a basic for loop on a range, but you don't need to specify a numerical range. You can supply any iterable object for the loop control. So for example, I can write a for loop that iterates over a list of items. So for example, I'll just define a list of some strings, and then inside the loop, I'll just simply print out the value of each item. Okay, so let's save this and let's run it, so we will hop over to the terminal, and I'll just execute this file. This is loops\_start. All right, and then you can see there the results of the range loop. So here's i going from one to 10, and then there is the looping over the list of strings here. So sometimes, you need to have nested loops which is a loop inside of a loop, and in other languages, this can be a bit tedious to read, but Julia makes this a little bit easier. So let's go back to the code, and let's go ahead and comment out some of the previous examples. So to create a nested loop, I'm going to write the for keyword and then I'm going to declare one variable called i, and I'll have that variable go from one to three, and then I'll write a comma, and then have another available named j and that will go from two to four. And then inside this loop, I'll simply print out the value of each. So I'll have i separated by a comma and a space, and then j. In this case, the i variable represents the outer loop, and j represents the inner loop, all right? So let's go ahead and try this. And then you can see that the outer loop is going from one to three, and each time through the outer loop, the inner loop is executing. So we have two, three, four, two, three, four, and so on. So in addition to the for loop, Julia provides a while loop construct. So let's clear the terminal and let's go back to the code. Okay, so let's go ahead and comment this out. A while loop is a loop that executes while a particular logical expression evaluates to true. So I'll declare a local variable here. I is equal to one, and then I'll write while i is less than or equal to five, and then inside my loop, I'll just print out the value of i, and now I need to increment the i variable. Now, because i is defined outside of the scope block of the while loop, I need to indicate to Julia that it's a global variable. So I'm going to write global i plus equals one. And then I'll close it off my end statement. So each time through the loop, the condition is evaluated right here, and if it evaluates to true, then the body of the loop runs. So once again, let's try this in the terminal, and there you can see that the loop runs until i reaches the value of five. Okay, so, it's often useful to be able to iterate over a collection of data and have access to an index variable into that collection at the same time. So let's go back to the code and try this out. And I'll comment this previous example. So rather than make you construct that yourself, Julia provides an enumerate function that does this automatically, and if you've Python before, you've probably seen this. You can see here that I have an array of sport team names and if I wanted to loop over this array and have access to the index of each item, I can write code like this. I'll write for and I'll declare two variables, one called index and one called value, and then the rest of the loop looks the same. I'll say in and then I'll use the enumerate function to enumerate the teams collection and then I'm going to print out the index separated by a comma and a space and then the value. All right, so let's close that off. All right, so let's go ahead and save and let's go over to the terminal, and let's try running this. And you can see that each time through the loop, I have both the index value and the value of the string that's in the list. All right, let's take one more example. So let's go back to the code. And I'm going to comment this out. So let's take a look at how we can control our loops while they are running. So I'm going to uncomment this code here, and this is a for loop like we saw earlier that goes from one to 10. So I can use the continue statement if I want to skip a particular iteration of the loop. So for example, if I wanted to skip over numbers that are divisible by three, I can add a statement like this. I can write if i percent three is equal to zero, then continue. All right, so let's go ahead and try that out. And you can see that when I run, the print statement isn't running for the values three, six, and nine. So similarly, I can use the break statement to stop the loop before it would normally have terminated on its own. So let's go back and try that. So I can put an elseif condition in here. I can say elseif i is greater than seven, then break. So now, when the value of i becomes greater than seven, the loop will terminate. So let's go ahead and save and let's go back to the terminal. And now, when I run, you can see that three and six are being skipped and once the value of the loop reaches seven, and then gets incremented to eight, the loop terminates.

### **Functions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Functions are yet another foundational building block of modern programming that encapsulate distinct pieces of logic. So here in my editor I'm going to open the functions\_start file and we're going to exercise some of Julia's function features. Now we've already seen a few simple examples of functions in the course but here we're going to take a look a little bit deeper at how you can use the functions in Julia. So to start off, functions are defined using the function keyword and they're given a name. So in this case I'll use myfunc and I'll give my function two arguments; a and b. Now by convention function names are usually all lower case and might contain underscores if the name is hard to read. So inside the function we're going to define the logic and I'll just have a print line statement that says This is a function and then I'm going to return adding the two arguments together. And then at the end of the function we have the end keyword which is what closes the function off. So Julia functions return the result of the last expression in the function by default. So, you actually don't even need the return keyword technically. So let's take that out and let's try the example. So this function just adds two numbers, so let's try it out. I'll declare a variable needs a result and I'll simply call myfunc and I'll call it with two values and then I will print the result. Okay, let's go ahead and try it out. Here in the terminal we will just execute that function. There we go and you can see that there's the result. There's the print line that's inside the function and you can see that even without the return keyword we were able to return the result of the function. Function arguments can also be assigned default values which are used if you don't specify them when you call the function. So, let's go back to the code. I'm going to comment this out. So I'll create a new function and I'll call it function foo and I'll give it a and b as arguments and then I'll define a third argument named z and I'm going to type equals 10 inside the function definition. Then for the body I'll have a return statement and I'll return a plus b times z and then we'll end that off and then we'll just try exercising it a couple times. So I'll print the result of calling foo with two and three and then I'll call it again with two, three and five. So, if I don't specify a value for z like in this case then z is going to default to whatever I put in this equals statement right here. So let's go ahead and save and let's try it out and there's the results. You can see that in the first case the default value of 10 is used for the z argument and in the second case it's using the value of five which I specified. All right, let's continue along. Julia also supports keyword arguments for functions. You just need to define them after the regular arguments and have them be separated by a semicolon. So, once again let's comment this. And I'll define a new function named bar and once again it will take two arguments a and b and then I'll have a semicolon and then I'll create a named argument, it's called multiplier, and I'll give it the value of 10 again and just like in the previous example I'll return a plus b times multiplier. So, what's the difference here? The difference is that for this argument that's specified after the semicolon you have to call it by name. Notice that in the previous example I was able to call foo with just the value because this is not a named argument. When I try this one out, I'm going to use the print line function here and I'm going to call this function with four and five and then I'll try calling it again but now because I'm using the named argument I can actually put that anywhere in the function call I want to. So, for example, I can put it right here in the front. I can say multiplier equals five. All right, so let's go ahead and try that out. And there you can see the result. So, here's where I don't specify the multiplier and here's where I do. Julia provides a shorthand way of defining simple functions. So, let's take a look at that next. So comment this. I don't need to use the function keyword or the end keyword I can just write the function directly in line. So I'll write myfunc again and this time I'll give it two parameters x and y and then I just simply define the function right in line by using the equals keyword and I'm going to be creating a compound expression so I'm going to use parentheses for this. So I'll write a is equal to x minus one and then a semicolon and then two a plus y. Now remember I don't need the return keyword. The last expression that's evaluated with a function is what's returned as the result. So that also saves on space. Now this is just a simple formula. We subtract one from x and then we return two times that plus whatever y is. So we invoke this function as we normally would. So I'll write result equals myfunc and we'll pass in three and four. So, let's give that a try. Oh, whoops, I have to print it out. All right, so let's save and try it. And there you can see the results. So, three minus one is two and then times two is four and then plus four is eight. Okay, for the last example we're going to see how to handle variable argument lists. This feature gives you a way to pass different numbers of arguments to a single function. So, let's go ahead back to the code and we'll comment this out. So here in the code I have a function named summit which essentially computes the sum of all the numbers that are supplied as arguments. So to specify that a function accepts variable arguments you use three dots in the function declaration. So I'm going to write args and then dot dot dot and then in the body of the function you can use the args keyword to access each of the arguments that are passed. So what I'm going to write is four a in args sum plus equals a and then I'll end that off and then I just return the sum. Now, we haven't yet covered some of the data structures like arrays or tuples but we will soon. So, for now just understand that the special args variable contains all of the argument values and then it's just a matter of returning the value at the end. So, let's go ahead and uncomment these last two lines here and let's run our final example and there you can see the results. So, we're summing up each one of these arrays and in each case we have the output which is all the numbers added together.

### **Exception handling**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Even when your code is solid and well written, your programs will encounter unexpected conditions. To handle these conditions, we use something called exception handling. This is a pretty common feature of other programming languages, and Julia provides it as well. Let's go ahead and open up the exceptions\_start file and you can see that we have some simple code here that performs a square root calculation. So there's a variable, and we give it a value. We call the square root function, and we print the result. So let's go ahead and run that, and you can see that the square root of nine is three. But now let's go back and change that value to a negative number and then let's save and run it again. And you can see now that we're getting an error because you can't perform a square root calculation on a negative integer number. You have to call it with a complex argument. So you can see that we're getting this domain error here. Now this error causes our program to exit in a very unfriendly way. So a better way to handle this condition is with an exception that catches the error and lets us handle it more gracefully. To do that we use what's called the try catch construct. So let's go back to the code here, and what I'm going to do is put the keyword try here, and then I'm going to indent these two lines, and then I'll write catch and then a variable to hold the exception. And then inside the catch block I'll write print line ex, and then I'll close off the try catch with my end statement. So what's happening here is if anything goes wrong in the try block then control will immediately flow over to the catch block where I can handle the exception. Now here I'm just printing it out, but you can handle the error however you want. So let's go ahead and save and now let's run the code again. So we'll clear the console and now you can see that instead of the program crashing out with a long stat trace we're handling the error ourselves. Now there's one more part of exception handling I want to show here, and that's called the finally block. So I'll go back to the code and right above the end statement I'm going to put in finally and I'll just put a print line statement in here that just says this section of code always runs. All right, so now let's save, and let's go back to the terminal, and let's run it again, and you can see the print line output of the finally section there. So the code that's in the finally section will always run. And that's useful, for example, if you're writing some code that deals with, say, files, and you open a file and then in the process of working with the file there's an exception. Inside the finally block you can do things like close the file handle so that you don't leak any resources. All right, so exception handling is an integral part of writing a well structured program that needs to handle error conditions gracefully. So get in the habit of using it in your code.

### **Arrays**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter we're going to learn about some of Julia's data structures for working with collections. And we're going to start off by looking at arrays. So here in chapter five I'm going to open up the arrays\_start file. So an array is an ordered collection of elements. And this data structure is found in most modern programming languages. In Julia you can create an array using square bracket notation like this. So I'll simply write a variable, and I'll give it some values. And if you've worked in JavaScript or Python, then this probably looks pretty familiar to you. In this case I have it specify the type of data that the array will hold so Julia will try to determine it. And you can see that these are all integers. So once we've defined the array, we can print it out. And we can also perform some other operations on the array such as determining what type it is. So I'll print out the type of array. And I'll also print out the array size by using the length function. Okay, and before we run that let's also show how to access an array element. So to access the array element you also use the square bracket notation. And it's important to note, and I think I said this earlier in the course, that Julia array indexes start at one. Not in zero like in Python or JavaScript or C. So let's go ahead and add a print line statement. And we'll say that the element at position two is, and then we'll print array. Now in Python or JavaScript you use one. But here we're going to use two. All right, so let's save and let's go over to the terminal. And let's go ahead and make sure we're in the right folder. And we are, we're in chapter five. So I'm going to write julia, and then I'll use the start version. Okay, so when I run this you can see in the output, like here's the contents of the array right here, and it has five elements. And you can see that the type is an array, and Julia inferred that it is a one-dimensional array of int64 integers. And then finally here you can see that the element at position two is in fact the number two. Now of course, arrays are not just limited to single data types. You can define an array that holds different types of data. So let's go back to the code and try that. I'll make a different array. And in this case I'll give it a string, and I'll give it an integer and a floating point number, and a negative number. And then once again we'll print out the contents and type. So I'll just copy and paste these two statements and make sure I'm operating on the right array. So let's comment out my previous example. Okay, so let's save and let's run this. And now you can see that when I run the code the type of the array is of type any. And again, it's a one-dimensional array, and here are the contents of the array. You can create an array that holds a specific type of element. So you specify that by indicating a data type when you declare the array. So let's go back to the code and try that out. In this case I'll make array three. And now I'm going to specify a type of Float64 in front of the square brackets. So once I have the array declaration I can add elements to the array dynamically. For example, by using a function like push. So I'll write push, and it is a Julia convention to put an exclamation point on the names of functions that directly modify one of their arguments. So in this case I'm going to push content into array three, and since I'm modifying this argument, there's a little exclamation point on the function call that tells me that that's going to happen. So I'll add 1.0, 2.0, and 3.0, and then let's go ahead and print the array out. And we'll comment out this previous example. Okay, so let's go ahead and run this updated code in the terminal. And you can see that there is the contents of the arrays. We have these three floating point numbers. Now remember, it expects floating point numbers. So let's watch what happens when I try to put a value on there that's not a floating point number. So I'm just going to put a string on here with four. And when I try to run this you can see that I'm getting an error. All right, let's try something else. Let's go ahead and comment this. There are some convenience functions that you can use to quickly fill an array with data. So for example, there's a function called zeros. And I can ask for four zeros and this will give me an array that has four zeros in it. There's also a ones function. So I can just copy and paste this. And I'll call ones instead. And there's a generic fill function, which takes any number along with a repetition value. So in this case I'll make array five and I'll call the fill function with five and seven, and then we'll print that out as well. All right, so let's go ahead and save this and let's run it. And there you can see that the arrays have been filled with those values. So here's the zeros, here's the ones, and then this array has been filled with seven instances of the number five. Okay, just a couple more examples for arrays. Let's go ahead try sorting some of the information in an array. So we'll comment this out, and I'll make a new array, and I will sort the original array that we have, and I'm going to specify that the reverse should be true, which means I have to go back up and uncomment the original array. And then I will print line the array. So the sort function also has an in place variant, which I can use with the exclamation point in the name. So for example, I can just write sort! new array. And then I can print line that back out as well. Well actually I can just use the new array there. There we go. And remember, this means that the data will be modified by the function in place. So let's go ahead and run these two examples. And there you can see that the array is being sorted in reverse order and then ascending order once again. All right, so one last example, we're going to use the join function to create a string from the elements of the array using a delimiter character. So let's go back to the code, and let's go ahead and comment this out. I'll make a variable named stir val. And I'll call the join function on our original array with a dash character. And then I'll print that out. And one last time let's run this. And you can see the resulting string with all the numbers and dashes in between each one. So that's just a quick introduction to Julia arrays. And of course there's much more to learn than this. So I encourage you to check out the documentation and try some of your own experiments.

### **Arrays**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter we're going to learn about some of Julia's data structures for working with collections. And we're going to start off by looking at arrays. So here in chapter five I'm going to open up the arrays\_start file. So an array is an ordered collection of elements. And this data structure is found in most modern programming languages. In Julia you can create an array using square bracket notation like this. So I'll simply write a variable, and I'll give it some values. And if you've worked in JavaScript or Python, then this probably looks pretty familiar to you. In this case I have it specify the type of data that the array will hold so Julia will try to determine it. And you can see that these are all integers. So once we've defined the array, we can print it out. And we can also perform some other operations on the array such as determining what type it is. So I'll print out the type of array. And I'll also print out the array size by using the length function. Okay, and before we run that let's also show how to access an array element. So to access the array element you also use the square bracket notation. And it's important to note, and I think I said this earlier in the course, that Julia array indexes start at one. Not in zero like in Python or JavaScript or C. So let's go ahead and add a print line statement. And we'll say that the element at position two is, and then we'll print array. Now in Python or JavaScript you use one. But here we're going to use two. All right, so let's save and let's go over to the terminal. And let's go ahead and make sure we're in the right folder. And we are, we're in chapter five. So I'm going to write julia, and then I'll use the start version. Okay, so when I run this you can see in the output, like here's the contents of the array right here, and it has five elements. And you can see that the type is an array, and Julia inferred that it is a one-dimensional array of int64 integers. And then finally here you can see that the element at position two is in fact the number two. Now of course, arrays are not just limited to single data types. You can define an array that holds different types of data. So let's go back to the code and try that. I'll make a different array. And in this case I'll give it a string, and I'll give it an integer and a floating point number, and a negative number. And then once again we'll print out the contents and type. So I'll just copy and paste these two statements and make sure I'm operating on the right array. So let's comment out my previous example. Okay, so let's save and let's run this. And now you can see that when I run the code the type of the array is of type any. And again, it's a one-dimensional array, and here are the contents of the array. You can create an array that holds a specific type of element. So you specify that by indicating a data type when you declare the array. So let's go back to the code and try that out. In this case I'll make array three. And now I'm going to specify a type of Float64 in front of the square brackets. So once I have the array declaration I can add elements to the array dynamically. For example, by using a function like push. So I'll write push, and it is a Julia convention to put an exclamation point on the names of functions that directly modify one of their arguments. So in this case I'm going to push content into array three, and since I'm modifying this argument, there's a little exclamation point on the function call that tells me that that's going to happen. So I'll add 1.0, 2.0, and 3.0, and then let's go ahead and print the array out. And we'll comment out this previous example. Okay, so let's go ahead and run this updated code in the terminal. And you can see that there is the contents of the arrays. We have these three floating point numbers. Now remember, it expects floating point numbers. So let's watch what happens when I try to put a value on there that's not a floating point number. So I'm just going to put a string on here with four. And when I try to run this you can see that I'm getting an error. All right, let's try something else. Let's go ahead and comment this. There are some convenience functions that you can use to quickly fill an array with data. So for example, there's a function called zeros. And I can ask for four zeros and this will give me an array that has four zeros in it. There's also a ones function. So I can just copy and paste this. And I'll call ones instead. And there's a generic fill function, which takes any number along with a repetition value. So in this case I'll make array five and I'll call the fill function with five and seven, and then we'll print that out as well. All right, so let's go ahead and save this and let's run it. And there you can see that the arrays have been filled with those values. So here's the zeros, here's the ones, and then this array has been filled with seven instances of the number five. Okay, just a couple more examples for arrays. Let's go ahead try sorting some of the information in an array. So we'll comment this out, and I'll make a new array, and I will sort the original array that we have, and I'm going to specify that the reverse should be true, which means I have to go back up and uncomment the original array. And then I will print line the array. So the sort function also has an in place variant, which I can use with the exclamation point in the name. So for example, I can just write sort! new array. And then I can print line that back out as well. Well actually I can just use the new array there. There we go. And remember, this means that the data will be modified by the function in place. So let's go ahead and run these two examples. And there you can see that the array is being sorted in reverse order and then ascending order once again. All right, so one last example, we're going to use the join function to create a string from the elements of the array using a delimiter character. So let's go back to the code, and let's go ahead and comment this out. I'll make a variable named stir val. And I'll call the join function on our original array with a dash character. And then I'll print that out. And one last time let's run this. And you can see the resulting string with all the numbers and dashes in between each one. So that's just a quick introduction to Julia arrays. And of course there's much more to learn than this. So I encourage you to check out the documentation and try some of your own experiments.

### **Random numbers**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] In this chapter we're going to learn about some useful features of Julia, that are either built into the language or are available as modules that you can import into your apps. And we'll start off with working with random numbers. It's fairly common when working with data to have to generate and use random numbers for statistics purposes and Julia provides many features for using random data, some of which we'll take a look at in this example. So I'll start up Random Start in my Editor here. And you can see that I imported the Random Module at the top of my file using the Using statement. So this imports the Random Module into my code and gives me access to the functions and features available in that module. So let's try something simple. We'll us the Rand function to create a random number. So I'm going to Print Line, and I'm going to call the Rand function with no arguments. Let's go ahead and save that and let's go to the terminal and let's just make sure I'm in the right place. So I'll type pwd, I'm in Chapter Six, so I will write Julia and it's going to be Random Start. And you can see that it generates a random number between zero and one, and if I run it a couple of times you can see that each time the number is different. But the Rand function is more versatile than that. We can use it to select a random item from a collection of values. So let's go ahead and try that. Let's write a loop. So for i in one to three, we are going to print out a rand call and for the random call I'm going to pass in a array of values. So I'll just give it six values, and then I'll close off my for statement. So we're going to choose three random items from this list. Let's comment that out and let's go ahead and run this. And you can see that we're choosing three random items from that list of numbers. Sometimes however what you want to do is the opposite. You want to populate a list with a series of random values. We can do that too. Let's go back to the code and let's comment this out. I'll create an array and I'm going to call the Rand function with a type, so I'm going to ask for unsigned eight-bit integers. And I want five of them and we will print out the array. Let's go ahead and run that. And there you can see that the results there are five random eight-bit integers. And notice that each value is limited to what will fit inside a UInt8. So let's switch gears a little bit and try generating a random stream of characters. So let's go back to the code. And we'll comment this. So to do this I'm going to use the Rand String function and this function will randomize a string of characters based on the characters that you give it to work with. So for example I will print out a call to Rand String, and I'll give it four characters here. And then I'll also print out Rand String and I'll ask for a range. I'm going to give it a range of characters from a to z. And I want six of those. So the default length of the string that rand string generates is eight. And in the second case you can see that I'm giving it a parameter that indicates that I want six characters. Let's go ahead and run this. And there you can see the random strings being created. So in this case I give it, A, B, C and D to work with and this is what it came up with. And you can see that it's eight characters long. In the second example I gave it a lower-case range from a to z and it gave me six characters. You can use the shuffle function to shuffle the contents of a collection in place. So let's go back to the code. And let's go ahead and comment these. So here I already have a variable that contains a collection of vowel characters and I can use the shuffle function to switch these around. So I'm going to write random.shuffle and I'm going to use the exclamation point version because I want to actually modify the array in place. And then once I've called that I'll print out the results. So let's go ahead to the terminal. And let's run that. And you can see that when I run this the vowels list is randomized in place. All right so one more example and here I'm going to show how you can control the randomization selection process. So just like many other languages, Julia's random number generator uses a seed value to tell the generator where to start selecting values. You can use the seed function to set the generator to a predictable location. And this is useful for scenarios where you want to generate the same data set over and over again when you run multiple experiments. So I'll set the seed value and create a random number each time. So I'll go ahead and comment that out and I'll use random.seed and remember I have to use the exclamation point because I'm changing the generator in place. And then I'll print line just a call to rand and I'll do that a couple times and then I'll print out a little separator so we can easily see the results. Then I'll just copy these lines and run them again. So let's save and let's run this. And you can see that the random numbers being generated are exactly the same in both cases because I'm setting the seed location to the same place in both cases. Of course there's a lot more to the random number generator than what I've shown here and I encourage you to do some experiments on your own.

### **String processing**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now let's take a look at some of Julia's functions for working with string data. I don't need to import any modules here, all the functions I'm going to show are built into the language. So let's go ahead and open up the strproc\_start. All right, so first, let's perform some simple operations like finding out the length and size of a string. And we kind of saw this earlier in the course, but I want to show some interesting things here. So I have, couple of variables here, and they've got some special characters. This one has Greek characters, this one has some nonstandard English characters, so let's go ahead and see what happens when we use the length and the size string functions. So first I'm going to print out the String length, and I'll use the length function for that. And then I'll print out the Size of string, and that's going to be sizeof. And let's do that with both of these, okay. So when I run this, notice how the length of the string in the first case is 5, but the size is 8. And in the second case, the length is 8 but the size is 10. This is happening because the strings contain Unicode characters for Greek and other nonstandard letters, and those take up more than one byte. This is why it's important to use the right functions when calculating the length of a string versus the size of a string in bytes. Okay, so let's go back to the code, and now let's try the concatenation and repetition string operators. Now we've already seen, in the past, how to use the concatenation operator, so I'll just go ahead and add it here, for just reference. And we can println that out. We can also repeat a string, actually let me go ahead and comment this first, so we can repeat a string by using the hat operator, so I'll have ABCD, and then I'll use the little upper hat there, and 3. All right, so let's go ahead and save that, and let's go ahead and run it in the terminal. And there you can see the Hello World, and, oh, whoops, I forgot to print this out, let's do that. All right, let's run it again. So there's the Hello World, and there you can see the ABCD being repeated three times. For the next example, let's test out some of Julia's substring search functions. So we'll go ahead and comment these out, and you can see that I have a test string right here. So I'm going to print out the result of calling the findnext function, and findnext takes a substring, so I'll just give it the word Julia, and the string I want to search in, and the position I want to start at, and then I'll also use the occursin function, to see if the word some appears in the teststr. All right, so let's run this, and you can see that findnext finds the text range where the word Julia appears, which is 1:5 in the string, and the occursin function returns true, because the term some does appear in the word awesome. All right, now let's try some string padding operations. So we'll comment these, and I'll make a new teststr, and I'm going to call the lpad function with the term Test String, and I'm going to pad this out by 20 characters, and then I'll print that, and then I'll have teststr = rpad, and once again I'll have Test String and I want 20 characters of padding, and I'm going to provide a default character to use instead of spaces. So I'll print that as well. All right, so let's save, let's clear the terminal, and let's run this, and you can see that now my Test String is being padded to take up a total of 20 spaces. So in this case, it's being left-padded by some spaces, and in this case, it's being right-padded by the character I gave it to use instead of the space character. All right, finally, let's try creating a string from an array. Let's comment this. So to do this, I'm going to use the join function, and I already have an array of values here. So the join function takes an array as the first argument, so I'm going to write join, and I'll give it the array, and then the character that you want to use between each element. So I'm going to give it a comma and a space, so, it's more than just a character, it's a string, and then you can specify a value to insert between the last two values, which is pretty handy. So I'll use the word and, all right, and then we'll print that out. Okay, let's go ahead and save, and let's run this. And you can see the result of the three words, forming a sentence. So here are the three elements, there's the spacing, character and space, and here's the word that appears between the last two elements. And again, I'm just scratching the surface here. I'd encourage you to take a look at the Julia docs, and maybe try out a few string functions on your own, to see how they work.

### **Using the filesystem**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this example we're going to see how to work with the file system. Julia provides a full set of functions and features for working with the file system. You can create and delete files, read and write data, and work with directories. So let's open the files\_start. Let's try some of these out. So for our first couple of examples, let's look at some simple directory operations. To get the current working directory we can use the pwd function. So I'll just print out the results of calling pwd. And we can also read the contents of the current directory by using the readdir function. So let's just try that out. All right let's go ahead and try running that. So I'll jump over to the terminal and I'll just run this code. And there we can see that here's the current working directory. It's chapter six of the example files. And here is a collection of all the contents in that directory. All right so now let's try working with some individual files. So we'll comment that. So first I'll create a function that will create a file with some content. So I'm going to write a function and the function's going to be called createafile. And I'll give it a file name parameter, that takes a string. And text, that takes a string. It's going to be equal to This is some text. And then I'll fill that in with the code to do the file creation. So I'll have a variable named io. And I'll call the open function and I'll give it the file name. I'm going to request write permission. And then I'll use the write function to actually write to that string, the text that is in the argument. And then I'll close our stream. So the open function opens a function for the specified kind of access, in this case write access, since I specified the w permission. And the file is created if it doesn't already exist. The return value is a reference to a file stream. And then the write function writes the data to that stream identifier. And then the close function closes the file. So now all I have to do is call this function. So I'll call create a file with mytestfile.txt. And I'll just write out the term Hello world! All right, so let's go ahead and run this. And if we look at the directory we can see now that mytestfile.txt has been created and there's the contents, right. Now let's write some code to read the file. So once again, I'll make a function for this. And I'll call it readafile. And I'll give it a filename parameter, which takes a string. Let's fill that in. So once again I'm going to use the open function. And in this case, I'm going to open with read access. And to get the content I'll call the read function on that stream. And I'll tell it that I'm reading string data. And then we'll just print out the content. And then I'll call this function. And it's mytestfile.txt. All right, let's go ahead and comment this so we don't create it again. So let's run. And sure enough you can see that now we are reading the content. All right one more example. Let's try appending data onto an existing file. So once again I'm going to make a function. And we're going to call it appendtoafile. And we'll give it a file name and some text. And I'll add some code here. So once again we'll open the file name. And I'm going to use append permission. And then we'll write the data. And then we'll close it. And then let's go ahead and append our data to the file. And that's going to be mytestfile.txt. And This text was added and then to make sure it works we'll call readafile. Actually we'll just cut it so it doesn't run again and we'll paste it there. All right so let's go ahead and run our updated example. And you can see that now the content was in fact added. All right let's try something a little bit different. So to rename an existing file we can use the move function. So I'm going to write a function called renamefile. And we're going to call mv and we're going to rename mytestfile.txt to newfilename.txt. And let's go ahead and call that. And I'll comment out the previous code. All right, and we'll close this. So let's go ahead and run that. And sure enough, you can see that the file name has now changed. It's now newfilename.txt. And then finally, let's go ahead and delete it. And I'm just going to call the rm function just right here without enclosing it. And that's going to be this file name right here. Let's go ahead and comment that out. And let's run this. And now you can see that the file has disappeared. So it's no longer in the list over here. All right, so of course, I've only covered a small part of what you can do with files. So for more information check out the file system docs on the Julia site at julialang.org and I'm sure that you'll find some ways that you can experiment and learn more.

### **Dates and times**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Julia also provides a module for working with date and time information which is what we will learn about in this example. So I'll go ahead and open up the datetime\_start in my editor. And you can see that I've imported the Dates module with the using dates statement here and that gives me access to the Date and DateTime classes. So let's start with something simple. I'm going to define two variables that represent a Date and a DateTime combination. So for the first one I'll have d1 and that's going to be a Date so I'll specify the year, month and day. And then I'll have dt1 and I'll create a DateTime and that's also going to have a year, month and day and I'm going to leave the time information out and then I'll just print these out to the output. (keyboard clacking) So, there's that and then we'll have the carriage return and dt1. Okay, so let's go ahead and try running that. So let's switch over to the terminal and all right so there you can see the Date output right there and the DateTime. So for the DateTime, since I didn't specify any values for the time, they just defaulted to zeros, so hours, minutes, and seconds are all zeros. You can also create date objects by parsing a text string and using a date format specification. So let's go ahead and try that. So to do this, you use a date format object to define a date format. So I'll define df1 is equal to date format, and that's going to be a year, month, and day format. And then you can parse a given string that uses that format. So I'll create date two, and that's going to be a date parser, and I'm going to parse this string, which is just that using the date format that I just created. So let's go ahead and save, and let's run that, and I'll just go ahead and comment out those lines. And, oh woops, I had to print it. (keyboard clacking) All right, so, and there you can see that the date is properly constructed when I do that. So sometimes it's necessary to calculate the differences between dates and times, and we can use the dates module to help with this. So, I'll start by creating some dates and times. So, I'll create date three, which is going to be a date here. And I'll create date four, and that will be a date as well, and then I'll create a couple of date times, and one more. So now that I've done that, I can perform some simple comparisons. Like seeing if one date comes before another by using the less than operator. So for example, I can write print line and see if d4 is less than d3. I can also calculate the differences between two dates. So, for example, I can print out, I can subtract d3, I can subtract d4 from d3, and I could do the same thing with the date times. So let's go ahead and put those in here. All right. So let's go ahead and comment out the prior code. Woops. Let's comment these guys out, and let's go ahead and save and run this. So you can see that d4 does in fact come before d3, so that result is true and then when I subtract d3 from d4, we can see the different is 147 days, and then, when we subtract dt4 from dt3, right, that's basically an entire year, and that is 31.5 billion milliseconds between those two dates. So I'll go ahead and comment these out. Julia provides convenient access or methods for several of the properties of date time objects. So for example, I could access the fields by using dates.year, and I can give that a date of d3 and I can also ask for something like, say, the week, and there's more than this, this is just a couple of examples. So I could ask for the week number for d4, and there's also some useful query functions that you can perform on dates. So, for example, I can write... The dates, dot, I can ask for the day of week for a given date. I can ask for the day name of d3, and I can do things like I can ask is leap year? And then finally, let's try some of what's called the adjuster functions. So, I can do something like, I can ask for the first day of week using d4. So if we scroll back up, you can see d4, this is a date sometime in July, and I could ask what was the first day of that week and I can also ask, say, what's the last day of month for d4? Okay, so let's go ahead and save and run these examples. All right so let's clear this and run it. All right, so here we have the year of d3, so we can see that that's 2019, and then the week number of d4 is the 30th week of the year. Then we print out the day of week for d3, and that's day number seven, followed by the day name which is a Sunday, and then we ask is d3 a leap year and the result there is false. And then for the adjuster functions, we ask what is the first day of the week for d4, which is this date right here, and we can see that the first day of that week was July 22nd, and then what's the last day of that month? Well, it's July 31st. So the dates module is pretty comprehensive and provides a lot of great pre-built code for working with dates and times. I suggest maybe taking some time right now and playing around with it a little bit more on your own.

### **Custom types**

Selecting transcript lines in this section will navigate to timestamp in the video

- Although Julia doesn't support creating classes like other object-oriented languages, you can create custom types. And we'll examine how to do that in this example. So let's go ahead and open up the types start file. The struct keyword is used to define a new type. So to create a new custom type called MyType I can write some code like this. So use struct and then call it MyType and then I can give it some fields. So I'll give it a field1 and I'll define that to be an integer. And then field2 and I'll make that one a string. And you get the idea. And then use the end keyword to close off the struct. So to instantiate this new type, I can use it like a constructor and pass values for the fields. So I can make a variable named x and I'll just instantiate my new type and I'll give it some values like 10 and just a sample string. And then I can print my variable to the stand it out and I can access fields using dot notations so I can write x.field1. Alright so, let's go ahead and run what we have. We've defined a type, we've instantiated it, and then we've printed out some information about the field. So I'm going to go ahead and save. And then here in my terminal, I'm just going ahead and run this. So I'll run type\_start and there in the output you can see that we've created the type. Right, so here's MyType. And then the value of field1. Now, by default, types are immutable which means you can't change the fields once you've assigned them. So let's go back to the code and let's try this. So if I write something like x.field1 is equal to 20 and then save. Let's try running that. And you can see that I'm getting an error. So if you want to make a type that is mutable, then you use the mutable keyword to indicate that. So let's go back to the code. And I'm going to copy this code here. And paste it down here. And I'm going to make a mutable version of this. So I'll write mutable struct and I'll just change the name to MyMutableType just to distinguish the two. And then I can instantiate it like I did before. So I'll write x equals MyMutableType and I'll give it some values. And now I'll try changing one of the fields. So I'll write x.field2 and I'll just change the string and then we'll print it out. Alright so now let's save and let's run. Well actually, let me get rid of this line right here that causes the error. Let's clear the terminal and run it again. And now you can see that I'm changing the value from ABC here to DEF. Alright, one more thing to point out is that you can use the built-in function called is a to see if a given variable is an instance of a given type. Let's go back to the code. So let's print and I'm going to use the is a function to see if x is a MyMutableType instance. And this works on built-in types too. You can check to see if a variable is an int, or a float, or a specific kind of float like a float16 or a float32. So let's just go ahead and save and run. And you can see that the result is true because x is an instance of MyMutableType. That's a quick instruction to types in Julia and you can see that even though you don't define methods like you do for classes in other languages, they're still pretty powerful so take a look at the documentation for this and play around with it on your own. You'll see that you can get quite a bit done with these.

### **Sorting data**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] All right, let's wrap up this chapter by taking a closer look at how to sort various kinds of data in Julia. So I'm going to open up my Sort Start file. And you can see here I've already got some data defined. And these numbers are clearly in a random order. So to sort data, we use the sort function. And I can supply a transformation function that gets applied to each item during the sort process. So let's go ahead and see a simple example. I can sort this information by using the sort function. And I'm going use the version of sort that does not have the exclamation point. It's going to return a new sorted array. So I'll sort array one. And then I can use the by parameter to specify that I want to apply a transformation function to each item in the array. In this case, I'll just simply apply absolute value to each one of these guys. So this will become a positive three, this will be positive 15 and so on. And then I can just print the result. Okay, so that's a good point to just try out what we have. Let's jump over to the terminal. And let's run. And you can see that the data items are now sorted by their absolute values. Notice however, that the item itself was not changed into positive three but it was treated as positive three for the purpose of sorting. We can also use the, is sorted function, to see if a set of data is already sorted. So let's go back to the code. And I will just comment this. So here I have an array of data, but this time it's an array of tuple values. So let's try using the is sorted function to determine if the data is already sorted. So first I'll use is sorted on the arrttup array. So I'll write print line. And I'll use the is sorted function. And I want to check the arrtup array to see if it's sorted already. Then I can use the by parameter to specify a lambda inline function. And I'm going to write X such that X returns the first value in the array. So what's going to happen is, this is sorted function is going to look at each one of these items. So each one of these tuples is an item. This little inline expression function is going to say, hey, for each item, take a look at the first item in that tuple. So each one of these first numbers will get looked at. So let's go ahead and run this. And you can see that the array is already sorted and the result is true because you can see that this data is one, two, three and four. So that's clearly already sorted. So let's try a different statement now. And I'm going to look at the second value in each tuple. And let's go ahead and run this again. And now you can see that when I'm looking at the second tuple value, the result is false. Now let's take a look at sorting custom types. So I'll go ahead and comment this out. Here is have a type definition that defines a rectangle. And then I have an array of rectangle structures. How would I sort this data if I wanted to sort based on the area of each rectangle. You can see that the structure doesn't have an area property for me to access that defines an area. Each rectangle just has a width and a height. So to do this, I need to define a custom sorting function. So I'm going to write a function called compare area and that's going to take a rect one and I'm going to define that as a myrect. And a rect two, which will also be a myrect structure. Now inside the function we can use the width and height of each rectangle to calculate the area on the fly. So I can have a calculation for area one, which will be rect ones length times rect ones width. And then I can do the same thing for area two. And I'll just look at the second rectangle there. And then I'm going to return whether area one is smaller than the second one. And then I'll close off my function. So now I need to use the sort function along with my custom function to sort the array of rectangles. What I'm going to do, is use the LT parameter which gives me a way to specify a function that will perform a less than comparison so that data items can be prepared. So for each item in the array, my function will be invoked to see which one is smaller than the next. And for this example, I'll use the sort exclamation point function to sort in place. So I'm going to sort array two in place. And then I'll specify the LT parameter is my compare area function. And then I'll just print out array two. All right, let's go ahead and save that. So let's run this. And now you can see that each of my rectangles are sorted according to the size of their area.

### **Next steps**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Joe] Well, it looks like we've reached the end of the course. Thanks so much for learning about the Julia programming language along with me, and I hope you've enjoyed the course. Julia is a rich and powerful programming language, and we've only just scratched the surface in this course. There's obviously a lot more to learn here. I would suggest staying up to date on Julia by visiting the official julialang.org site, which is the go-to resource for all things related to Julia. This is where you'll find up-to-date documentation and the latest version of the Julia language runtime. There's also a comprehensive Community page full of resources for learning more about Julia and where to get additional packages and examples. Thanks again for watching my course, and I hope to see you again soon in another one of my courses. Until then, happy coding.