Video #8: Working with Strings - Things We Know

In the next two videos, we'll build on your experience working with strings. Since the start of the semester, we've used strings in many ways - prompting the user to enter information for a program, displaying information to the user with disp, specifying how to plot something or providing labels for our visual displays. We stored multiple strings in cell arrays like in this example of Red Sox players, and used strings to specify file names, for example, when loading an image into MATLAB. Recently, you've had a taste of processing strings, translating a string like 'Stella' into ubbidubbi, 'Stubelluba'. In this video, we'll touch on things you actually know already, but maybe don't know that you know. In the next video, we'll learn some new things we can do with strings in MATLAB.

We'll start by taking a closer look at how MATLAB and other programming languages actually represent a string of characters. Computers use a numerical code to represent characters like letters, digits, and punctuation. The first numerical code established as a standard for this was the ASCII code, from the American Society for Communication and Information Interchange. The original code could represent 128 different characters that you can see in this table, along with a set of control signals that were used in old teletype machines for entering programs into a computer. These days, computer systems use a coding called Unicode that represents thousands of characters and includes many different alphabets.

You can view this numerical encoding in MATLAB. Strings are stored in a special type of numerical vector that contains the ASCII values. Earlier, in the Command Window, I created a string myString = 'spring break' and if I list the variables in my current workspace with whos, we see that my string is stored in a 1x12 vector of values of type char. There are functions, like double and char that convert between the numerical code and the characters - if I run this section, we see the numerical codes for the letters in the word violet, and can convert a vector of numerical codes back to characters to see the string that this corresponds to.

A key point here is that strings are stored in vectors, which we can visualize like myString here, where each character is stored in a separate location of the vector. Because they're stored in a vector, there's lots of things we can do with strings, for free, things we can do with any vector. Already we've been getting the length of strings, and using an index to refer to the character at a particular location within a string. We can refer to an extended substring within a string using a contiguous set of locations that we specify with colon notation, or we can give a vector of specific indices. What would this give us? 'grab'. We can also change a set of locations to some other string, e.g. change 'break' to 'fling', which should give us 'spring fling'. We know that when we have a numerical vector and try to add a new value at a location that's beyond the length of the vector, MATLAB automatically expands the vector to accommodate that distant location and pads intervening locations with zeros. Let's see what happens when we try to add a character someplace beyond the length of myString <run section, see earlier results> - it again auto-expands the string and now pads intervening locations with spaces.

Another thing we've done with vectors is apply conditional expressions to entire vectors all at once - this is also a freebie we get with strings. Let's start with the 'spring fling' string and create a logical expression that compares the whole string to a single character 'g' using ==. This will generate a logical vector the same length as the extended string (12 chars), and this logical vector will have a true value wherever the character 'g' appears. If we want to know the actual indices where 'g' occurred, we can call the find function. We can also use a logical vector as an index. In this statement, the inner expression (myString == 'f') again gives us a logical vector that's true wherever an 'f' appears in the string, but here, we're using this logical vector as an index - wherever it's true, we're going to change the character to 'b'. So this statement replaces f's with b's, giving us spring bling. <now run section to check results>

We can also compare two extended strings with ==, but take a moment to think about what happens in these two cases before I run this section <see results> 'ellen' == 'silly' is a valid expression - what did it do? It compared the two strings element-by-element to check whether the characters at the same locations are the same character - there's an l in the same location, so the expression gives a logical vector with one true value and the rest false values. To use == with two strings, they need to be the same length. How do we compare two strings that have different lengths? Like we've been doing, we use strcmp('ellen', 'serious'), which gives us a single logical value that's true if they're the same string, otherwise false.

There's one final example of the application of conditional expressions to a string that I'd like to touch on here. We'll take advantage of this example in the next video. Suppose we create a string like 'to be or not to be' that I'll call hamlet, and we want to remove the spaces in the string and just preserve the letters. We can do this a couple different ways that have different consequences. In one approach, we'll collect all the characters that are not spaces and put them into a new string that I'll call letters. The inner expression (hamlet ~= ' ') gives us a logical vector that's true at locations that are not spaces (locations of the letters), then we'll supply that logical vector as an index to hamlet. This selects the content of hamlet wherever we have true values in the logical vector, which selects all the letters in this case. As I describe the second approach, think about whether the hamlet string itself was changed by this action. In the second case, remember that we can remove an element from a vector by assigning a location to an empty vector. Here we figure out where the spaces are, the places where the expression hamlet == ' ' is true, and those are the places that we remove from hamlet. Here also, the question arises, is hamlet changed by this action? Let's run this section and see - the first approach copies non-space characters to the letters string without altering hamlet, while the second approach does alter hamlet by ripping out the spaces from the existing vector. If you want to perform an operation like removing spaces from a string, think about whether it's important to preserve the original string in the process.

So far we explored operations on strings that you really already knew about, because strings are stored in vectors, so you can do many things with strings that you can do with any vector. In the next video, we'll add a few new things that you can do with strings.