Video #9: New Things About Strings

In this second video on strings, you'll learn about some new things you can do with strings and explore additional applications of string processing. First, we have the ability to convert all the letters in a string to upper or lower case using the built-in functions upper and lower. <run section> There are two other functions that I'd like to mention here, which can be helpful when processing text. The first looks like "strep" - it's short for "string replace" and it takes three inputs: an extended string and two substrings that are typically both shorter. It creates a new string that it returns, and in this new string, all occurrences of the first substring are replaced by the last string. In this case, all occurrences of 'Be' in the shakespeare string are replaced with 'Play' - let's see what we end up with, 'To Play Or Not To Play' and we assigned this to newString. We can see that our original shakespeare string is left intact. There's a second new function here, strfind, that has an input string and a second input that's a substring, and based on the result here, what did it do? It searched for occurrences of this substring ay in our string 'To Play Or Not To Play', found two places where 'ay' occurred, and returned the index of the start of each occurrence (the first started at index 6 and the second started at index 21).

Now let's explore some examples. In the first example, suppose we want to write a function that tests whether an input string is a palindrome - the same string forwards and backwards. Here are some sample palindromes (apparently Napoleon didn't really say this last line). These examples tell us two things - the locations of the spaces in the first two examples are different when we move forwards and backwards through the text, so we probably want to get rid of the spaces, and we know how to do that from the last video. Also, capitalization doesn't matter - the capital M in Murder and lower case m in rum are considered the same, so we should convert the letters to the same case before checking whether there's a palindrome. This gives us three steps to solve this problem - remove the spaces, convert the letters to lower case, and then check if the string is the same forwards and backwards. How can we translate this to MATLAB? We saw two approaches to removing spaces in the last video - here I collect the non-space characters and re-assign str to the new string with just the letters. Then we convert to lower case. And finally, we can compare the string as is (str), to what happens if we refer to the characters in reverse order, starting with the end location and going down to the index 1. If these two versions of the string are equal in every location, then we must have a palindrome. Let's test this function, and while we're at it, note that the last statement of the function provides a hint about how we might create a reversed version of a string in general, by referring to the locations of the string in reverse order from end down to one. Actually this is a useful enough operation that MATLAB provides a built-in function for this, called reverse. <run section and view results>

So we know that MATLAB provides a reverse function, but I'm going to define a new function to do this, to plant an idea that we're going to use in our next application of string processing. We'll call the function reverseStr and it will take an input string and return the reverse of it, which I call newString in the function definition. We'll use a for loop to construct newString, starting from an initial empty string. The input str is a vector, and the way the for loop works is

that we set up a control variable for the loop (here that variable is called letter) and we specify a vector of values that the control variable will be assigned to, one by one, as we repeat the loop. So letter will be assigned to the first character in str and the body of the loop will be executed, then it will be assigned to the second character of str, and so on. There's just one statement in the body of the loop that reassigns newString to the result of tacking the new letter onto the left of whatever we have already in newString. To understand the effect of this operation, let's hand simulate what happens in the loop when we call the function with the string 'april'. First letter is assigned to 'a' and this letter is added to the initial empty string, giving 'a'. Then letter is assigned to 'p' which is added to the left of the current newString, giving the new value 'pa', then letter is assigned to 'r' and newString becomes 'rpa', and a similar thing happens with the 'i' and 'l', giving us 'lirpa' as the final string that's returned. The idea here is that we're stepping through an input string, and as we go, we're performing an operation that builds up a new string using concatenation.

Our last example applies this idea to a problem from biology that involves translating an RNA sequence to a string of amino acids. A DNA molecule is a sequence of nucleotides, and DNA is transcribed to RNA, also a sequence of nucleotides, and RNA is then translated into a protein. RNA sequences contain four nucleotides shown in this picture. They're abbreviated A, G, C and U, which stand for Adenine, Guanine, Cytosine, and Uracil. So an RNA sequence might look like this - CAGACU… Each set of three contiguous RNA nucleotides is called a codon, and it codes for a single amino acid. So in this example, the triplet CAG codes for the amino acid Glutamine (abbreviated Gln), the next triplet ACU codes for Threonine, and so on. A protein is made of a chain of amino acids hooked together. Our task is to write a function to translate an RNA sequence into an amino acid sequence, and to accomplish this, we obviously need to process strings of text. I started by writing a function rna2amino which translates each codon, or triplet of nucleotides in an input string. The details of translating the codon to a specific amino acid are tucked inside a function translateCodon that we'll look at later. The input string is called nucleotides, and this picture shows an example of an input string stored in a vector. The function returns a string that I call aminoAcids that looks like the string we saw earlier. We start by assigning aminoAcids to an empty string, and in this for loop, we want to step through the codons in the input string and translate each codon to an amino acid that we tack onto the aminoAcids string as we go. Here, we'll set up an index i that will refer to the starting index of each codon. So i will first be assigned to 1, the start of the first codon, but then each time we repeat the loop, we increase i by 3, so the second time through the loop, i is 4, at the start of the second codon, and the third time through the loop it's assigned to 7, where the third codon begins, and so on. The last value it will be assigned to is the length of the input string minus 2, which is 12-2 in this case, the value 10, which is the starting index for the last codon. Inside the for loop, we assign aminoAcids to the result of concatenating what we have so far with a space and with the amino acid name that's returned by the translateCodon function. This translateCodon function takes a single codon, a substring of 3 nucleotides starting at index i and going up to index i + 2 (for the first codon, this would be indices 1 to 3).

So how do we translate this codon into an amino acid? First, we'll return to the biology for a moment. Suppose we're given a codon like AGU - this codes the amino acid Serine, but how can we determine this? We're going to use this table here. In the central region is the amino acids, and you can see that many are repeated in multiple locations in this middle area, because there are multiple combinations of the three nucleotides in the codon that code for the same amino acid. To use this table, we first consider the nucleotide that's given in the first position and look at this column on the left that's labeled first position at the top. Here, the first position is A so we're going to restrict the region of the table that we consider to the four rows labeled with A (highlighted in green). Then we look at the second position in the codon, and in the central region labeled second position at the top, we select the column corresponding to the nucleotide in the second position. It's G in this case, so we're going to consider just this column labeled G at the top (blue). Given the combination of A in the first position and G in the second position, our possibilities are narrowed down to the four where these two regions intersect. Finally, we consider the nucleotide in the third position, which is either U, C, A or G, and that tells us the particular row of the corresponding amino acid. The third position is U in our example, so taking all three nucleotides together, the amino acid is Serine, circled in red.

How do we incorporate this table into our MATLAB code? First we're going to take these columns of amino acids and place them inside one long cell array, listing the first column of amino acids first, then the second column, and so on, This is done in a script that I called createAminoTable <look at script in editor>, the cell array is named aminoNames, and there are 64 names here. Back to our table, it turns out that if we associate the 4 nucleotides with particular integers, 0, 1, 2, and 3, we can perform a simple calculation using this numerical representation that tells us where in our long list of amino acids, to find the correct amino acid for this codon. I don't want to take the time to derive the formula that we'll use for this calculation, but you'll see from the code, that it's pretty simple. So let's look at the code. The function translateCodon has one input that's a string of three nucleotide letters that I call codon, and it returns the corresponding amino acid abbreviation that I call aminoAcid. We first call our script createAminoTable that creates that cell array named aminoNames. The next step is to convert the nucleotide letters to the integers 0-3. I get the letter in each of the three positions and see where it appears in a string of possibilities UCAG. find gives the index, which will be from 1 to 4, so I subtract one to get an integer from 0 to 3. So n1, n2, and n3 are the numerical encoding of the nucleotides in the three positions of the codon. This is the simple calculation that I told you about, which gives an index between 1 and 64, and we use this index to pull out the correct amino acid in the aminoNames cell array. A bit magical, but I'm using this example to illustrate what we've learned about processing strings. We can run some examples to check the results. <run section>

So that's it for the solution to some problems that involve processing textual information, and how we can use strings to store this textual information and use MATLAB to process these strings of text.