Lecture 16 - String Processing

Packages

```
# none! we will use base R
```

String Processing

- We have already worked with the `character` data type on several occasions throughout this course
- Often times, if the `character` variable has consistent categories, we covert the variable into a `factor` data structure (i.e. categorical variable)

- But what if a `character` variable doesn't have consistent categories?
- What if the formatting of categories is inconsistent?
- What if the `character` variable consists of long strings of unique text?
- What if we only want a portion of the `character`, as opposed to the entire string?

- To manipulate characters/strings in this way is often referred to as ***string processing***
- There are many functions in base `R` that enable us to process and manipulate characters/strings for a given purpose.

- Here is a "small" cheat sheet:
  - `strsplit(x, split)`
    - splits a string `x` by another string `split`
  - `paste0(x1, x2, x3, ...)`
    - concatenates strings with no characters in between
  - `paste(x1, x2, x3, ..., sep, collapse)`
    - concatenates strings with a string `sep` in between. Can also be used to concatenate strings in a vector using `collapse`
  - `toupper(x)`
    - converts all characters in a string to upper case
  - `tolower(x)`
    - converts all characters in a string to lower case
  - `nchar(x)`
    - outputs the number of characters in a string
  - `substring(text, first, last)`
    - Extracts the `first` though `last` characters from a string `text`
  - `grepl(pattern, x)`
    - Outputs a logical representing if a `pattern` is in a string `x`
  - `grep(pattern, x)`
    - Outputs which elements in a character vector `x` contains a `pattern`
  - `gsub(pattern, replacement, x)`
    - Replaces a `pattern` in string `x` with another string `replacement`
  - `gregexpr(pattern, text)`
    - Outputs the location of the first character of a `pattern` in a string `text`

- Yes, we will cover all of these!

`strsplit()`

- The function `strsplit()` splits a given string by a pattern you provide to the function
- The `strsplit(x, split)` function commonly takes two arguments
  - `x` : character string or character vector to be processed
  - `split` : the pattern over which you are splitting `x`

- For example, let's split a quote by Roman emperor and philosopher, Marcus Aurelius, by the pattern of a space `" "`
  - "Very little is needed to make a happy life; it is all within yourself, in your way of thinking."

```
# assign string
quote <- "Very little is needed to make a happy life; it is all within yourself, in yo
quote
```

'Very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

- The following script uses `strsplit()` to separate any string between a space `" "`
- The output is an unstructured list

```
# split the quote over the pattern " "
quote_split <- strsplit(x = quote, split = " ")
```

```
quote_split
```

1. 'Very' · 'little' · 'is' · 'needed' · 'to' · 'make' · 'a' · 'happy' · 'life;' · 'it' · 'is' · 'all' · 'within' · 'yourself,' · 'in' · 'your' · 'way' · 'of' · 'thinking.'

```
# check the class
class(quote_split)
```

'list'

- We can then process the same way we work with lists in the past
- Remember, if an unstructured list doesn't have list names, we use two brackets for numeric-indexing

```
# indexing the first list item
quote_split[[1]]
```

'Very' · 'little' · 'is' · 'needed' · 'to' · 'make' · 'a' · 'happy' · 'life;' · 'it' · 'is' · 'all' · 'within' · 'yourself,' · 'in' · 'your' · 'way' · 'of' · 'thinking.'

```
# indexing a single word
quote_split[[1]][8]
```

'happy'

```
# Find where happy occurs in quote
which(quote_split[[1]] == "happy")
```

8

`paste()` and `paste0()`

- Both the `paste()` and `paste0()` functions combine/concatenate strings together

- `paste0()` concatenates strings with no delimiter or separation between the strings

```
# paste with no string separation
paste0("Very", "little", "is", "needed", "to", "make", "a", "happy", "life;",
       "it", "is", "all", "within", "yourself,", "in", "your", "way", "of",
       "thinking.")
```

'Verylittleisneededtomakeahappylife;itisallwithinyourself,inyourwayofthinking.'

- `paste()` concatenates strings with specified separation between the strings
- Use the `sep` arugment to specify how you want the strings separated

```
# paste with a space separation
paste("Very", "little", "is", "needed", "to", "make", "a", "happy", "life;",
      "it", "is", "all", "within", "yourself,", "in", "your", "way", "of",
      "thinking.",
      sep = " ")
```

'Very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

```
# paste with a hyphen separation
paste("Very", "little", "is", "needed", "to", "make", "a", "happy", "life;",
      "it", "is", "all", "within", "yourself,", "in", "your", "way", "of",
      "thinking.",
      sep = "-")
```

'Very-little-is-needed-to-make-a-happy-life;-it-is-all-within-yourself,-in-your-way-of-thinking.'

- But what if your strings are in vector form?
- We must specify the `collapse` argument, which collapses each string in the character vector into a single string using a specified string pattern

```
# string defined previously
quote_split <- strsplit(x = quote, split = " ")
quote_split[[1]]
```

'Very' · 'little' · 'is' · 'needed' · 'to' · 'make' · 'a' · 'happy' · 'life;' · 'it' · 'is' · 'all' · 'within' · 'yourself,' · 'in' · 'your' · 'way' · 'of' · 'thinking.'

```
# need collapse to combine strings in a vector
paste(quote_split[[1]], collapse = " ")
```

'Very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

```
# collapse with other patterns
paste(quote_split[[1]], collapse = " : ")
```

'Very : little : is : needed : to : make : a : happy : life; : it : is : all : within : yourself, : in : your : way : of : thinking.'

- Note this does not work use `sep`

```
# does not work without collapse
paste(quote_split[[1]], sep = " ")
```

'Very' · 'little' · 'is' · 'needed' · 'to' · 'make' · 'a' · 'happy' · 'life;' · 'it' · 'is' · 'all' · 'within' · 'yourself,' · 'in' · 'your' · 'way' · 'of' · 'thinking.'

`toupper()` and `tolower()`

- The functions `toupper(x)` and `tolower(x)` convert strings to all upper case or all lower case
  - `x` : The string input

```
# convert to all upper case
toupper(quote)
```

'VERY LITTLE IS NEEDED TO MAKE A HAPPY LIFE; IT IS ALL WITHIN YOURSELF, IN YOUR WAY OF THINKING.'

```
# convert to all lower case
```

```
tolower(toupper(quote))
```

'very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

- These functions are especially useful when the case of the text is not important for the analysis
- For example, if you are searching for the phrase "data science", "Data science", and "Data Science"

`nchar()`

- The function `nchar(x)` outputs the number of single characters in a string
  - `x` : The string input

```
# how many characters are in the following string?
nchar("Data Science")
```

12

```
# original quote
quote
```

'Very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

```
# use nchar to count the number of characters in the quote
nchar(quote)
```

95

- These functions can also be applied to a vector of strings

```
# quote split by " "
quote_split[[1]]
```

'Very' · 'little' · 'is' · 'needed' · 'to' · 'make' · 'a' · 'happy' · 'life;' · 'it' · 'is' · 'all' · 'within' · 'yourself,' · 'in' · 'your' · 'way' · 'of' · 'thinking.'

```
# find number of characters in each vector entry
nchar(quote_split[[1]])
```

4 · 6 · 2 · 6 · 2 · 4 · 1 · 5 · 5 · 2 · 2 · 3 · 6 · 9 · 2 · 4 · 3 · 2 · 9

`substring()`

- The function `substring(text, first, last)` subsets a portion of the string based on a starting and ending index
  - `text` : string to be subsetted
  - `first` : the starting of the string to be subsetted
  - `last` : the last index of the string to be subsetted

- For example, let's extract the first half of the quote using `substring()`

```
# original quote
quote
```

.

'Very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

```
# extract the first 42 characters from the quote
substring(quote, 1, 42)
```

'Very little is needed to make a happy life'

```
# extract the second portion of the quote
substring(quote, 45, nchar(quote))
```

'it is all within yourself, in your way of thinking.'

- `substring()` also enables you to overwrite portions of a string

- Let's replace the semicolon `";"` with a period `"."`

```
# check what we want to overwrite
substring(quote, 43, 45)
```

'; i'

```
# overwrite the ";" with a "."
substring(quote, 43, 45) <- ". I"
quote
```

'Very little is needed to make a happy life. It is all within yourself, in your way of thinking.'

`grepl()` and `grep()`

`grep()`

- The function `grep(pattern, x)` returns the index of entries that contain a specified pattern
  - `pattern` : you are checking if `x` contains this `pattern`
  - `x` : character value or character vector

- Let's apply `grep()` to a version of the quote
- Note that I have stored the quote as a character vector of length 2

```
# assign quote
quote <- c("Very little is needed to make a happy life",
           "it is all within yourself, in your way of thinking")
quote
```

'Very little is needed to make a happy life' · 'it is all within yourself, in your way of thinking'

- The script below using the `grep()` function to determine if the string `"happy"` is contained within the quote.
- The `grep()` function returns the index of the character vector entry that contains the string pattern `"happy"`.
- Since `"happy"` occurs in the first entry of the vector, `grep()` returns `1`

```
# find "happy" in quote
grep(pattern = "happy", x = quote)
```

1

- Since `"thinking"` occurs in the second entry, `grep()` returns `2`

```
# find "happy" in quote
grep("thinking", quote)
```

2

grepl()

- The function `grepl(pattern, x)` returns a logical `TRUE` if the corresponding entry contains a specified pattern
  - `pattern` : you are checking if `x` contains this `pattern`
  - `x` : character value or character vector

```
# show quote
quote
```

'Very little is needed to make a happy life' · 'it is all within yourself, in your way of thinking'

```
# check if "happy" is in character vector quote
grepl("happy", quote)
```

TRUE · FALSE

```
# check if "thinking" is in character vector quote
grepl("thinking", quote)
```

FALSE · TRUE

```
# check if "Data Science" is in character vector quote
grepl("Data Science", quote)
```

FALSE · FALSE

---

gsub()

- The function `gsub(pattern, replacement, x)` replaces a `pattern` with another specified string `replacement` in a string `x`
  - `pattern` : you are checking if `x` contains this `pattern`
  - `replacement` : the string with which you are replacing `pattern`
  - `x` : character value or character vector being processed

- The script below uses the `gsub()` function to replace the string `"happy"` with the string `"fulfilling"`

```
quote
```

'Very little is needed to make a happy life' · 'it is all within yourself, in your way of thinking'

```
# replace "happy" with "fulfilling"
gsub(pattern     = "happy",
     replacement = "fulfilling",
     x           = quote)
```

'Very little is needed to make a fulfilling life' · 'it is all within yourself, in your way of thinking'

`gregexpr()`

- The function `gregexpr(pattern, text)` returns the position of all the occurrences of a `pattern` in a string `text`
  - `pattern` : you are checking if `text` contains this `pattern`
  - `text` : character value or character vector being processed

- The followig script uses the `gregexpr()` function to location the first character of the pattern `"happy"` in the quote
- The `"h"` in `"happy"` is the 33rd character in the first entry of the quote vector
- Note that `gregexpr()` returns a `-1` if the string does not contain the pattern

`quote`

'Very little is needed to make a happy life' · 'it is all within yourself, in your way of thinking'

```
# find the location of "happy" in the quote
gregexpr("happy", quote)
```

1. 33
2. -1

- Let's check if this is correct!

```
quote[1]
```

'Very little is needed to make a happy life'

```
substring(quote[1], 33, 33 + nchar("happy") - 1)
```

'happy'

Escape Characters

- We have always used quotes `""` or apostrophes `''` to define strings in `R`
- But what if the string itself has a quote?
- For example, what if we are working with height measurements, which include both apotrophes and quotations?

- `5'8"`

- Let's attempt to define this measurement as a string

```
# define height measurement as a string
"5'8""
```

```
Error in parse(text = input): <text>:2:6: unexpected INCOMPLETE_STRING
1: # define height measurement as a string
2: "5'8""
        ^
Traceback:
```

Étapes suivantes : Expliquer l'erreur

- We encounter an error since the quotation representing the inches measurement conflicts with using quotations to define a string!
  - That is, `R` thinks we are ending a string after `8` and doesn't know how to interpret the remaining quotation

- Therefore, we must explicitly tell `R` that the quotation defining the inches measurement is inside the string, NOT ending the string
- We can do this by including a backslash `\` before the quotation
- This is known as "escaping" since the `\` escapes the default usage of the quotation

```
# with escape character \
"5'8\""
```

```
'5\'8"'
```

- But what if we want to include a backlash `\` in the string itself?
- We would need to escape the escape `\`!

```
# without escape character for \
"5'8\"
```

```
Error in parse(text = input): <text>:2:1: unexpected INCOMPLETE_STRING
1: # without escape character for \
2: "5'8\"
   ^
Traceback:
```

Étapes suivantes : Expliquer l'erreur

```
"5'8\\"
```

```
'5\'8\\'
```

---

cat()

- The function `cat()` is similar to the `print()` function, except the strings are printed as you would read them in text

- Notice how the escape characters are visible when using `print()`

```
# using print
print("5'8\"")
```

```
[1] "5'8\""
```

```
print("5'8\\")
```

```
[1] "5'8\\"
```

- When using `cat()`, the string is printed with its formatting

```
# using cat
cat("5'8\"")
```

```
     5'8"
```

```
cat("5'8\\")
```

```
     5'8\
```

Common Escapes

- `\n` : Newline (line break)
- `\t` : Tab
- `\"` : Double quote (")
- `\'` : Single quote (')

```
# assign string
quote <- "Very little is needed to make a happy life; it is all within yourself, in yo
quote
```

'Very little is needed to make a happy life; it is all within yourself, in your way of thinking.'

```
# split string by " "
quote_split <- strsplit(quote, split = " ")
quote_split
```

1. 'Very' · 'little' · 'is' · 'needed' · 'to' · 'make' · 'a' · 'happy' · 'life;' · 'it' · 'is' · 'all' · 'within' · 'yourself,' · 'in' · 'your' · 'way' · 'of' · 'thinking.'

```
# insert new lines between each word
new_quote <- paste(quote_split[[1]], collapse = "\n")
new_quote
```

'Very\nlittle\nis\nneeded\nto\nmake\na\nhappy\nlife;\nit\nis\nall\nwithin\nyourself,
\nin\nyour\nway\nof\nthinking.'

```
print(new_quote)
```

```
     [1] "Very\nlittle\nis\nneeded\nto\nmake\na\nhappy\nlife;\nit\nis\nall\nwithin\nyou
```

```
# print using cat()
cat(new_quote)
```

```
     Very
     little
     is
     needed
     to
     make
     a
     happy
     life;
     it
     is
     all
     within
     yourself,
     in
     your
     way
     of
     thinking.
```

Hi class. So welcome to today's lecture on string processing. What's nice is all of the functions we're going to use today will be in base. R, so no packages needed.

All right. So we've already worked with this character data type quite a few times throughout the semester. So far, right. And oftentimes this character data type or this character type of variable, has consistent categories and will often convert it into a factor, because we want to analyze that variable as a factor, data structure or categorical, variable

right? But the thing is text data can take many forms right? What if that character data, type or character vector doesn't have consistent categories. What if there are different cases, different punctuations? Right? What? What if the formatting of these categories is inconsistent? We can't simply just convert it into a factor. Right? We need to preprocess it in some way.

Another thing is what if the character variable consists of long strings of unique text.

So it's not necessarily a categorical variable. It's just a variable of text that we'd like to data mine or analyze right?

And what if we only want, say, a portion of that character, we just want to extract some tiny bit of meaningful information from that long string of text.

Well, this brings us to this topic of string processing, where we can process or manipulate strings to extract useful information for our analysis. Okay.

now, there are many, and I mean many, many different functions. Not just in base. R, but if you expand that out to say the tidyverse library, there are so many functions out there that can be used to process and manipulate these character data types or just strings of text

right?

And today, I've created this small cheat sheet. But today we're gonna cover all of these functions. And this doesn't even really scratch the surface of all of the functions you can use to process text.

but it's a great start. All with all of these functions you will be able to do quite a few different things with strings within. R. Now, I've created this little cheat sheet list here, because in the practice assignment you'll be able to quickly refer to that when you want to use a specific function. But if not, you can obviously refer back to the entire lecture. And again, we're going to cover all of these today.

Alright. So the 1st function that I'm gonna cover within this

topic of string processing is str split.

And as you might imagine, this stands for string split. So we're going to split strings based on something. Right? So this function str split or string split splits a given character data type or splits a given string by some pattern or some string you provide to the function. All right. So these are the arguments. It takes an argument. X and something called split.

Now, the X argument is just some character string or character vector. That we want to process. So this is the data you have. This is your actual character. String

now split is a pattern over which you are splitting.

So if you like to split, say, a string by its spaces, or the letters T or H. You can absolutely do that.

So to illustrate this, let's look at this example. Let's split a quote by this famed, you know Roman emperor and philosopher, Marcus Aurelius, you might know him from that one movie gladiator. Right? So let's split a quote by this person, Marcus Aurelius by the pattern of just a space. All right. And we're going to denote a space with, quote space and then quote.

All right. So the quote is very little is needed to make a happy life.

It is all within yourself in your way of thinking. Okay. So we all know how to create or define character data types. Right? We use these double quotes, or we use an apostrophe. And I'm going to assign it to this variable quote.

So all we've done here is assign our quote. And now we can process this quote using the function string split or Str split. Okay, so this is the script here that separates any pieces within the string, any substrings within the string using a space. Okay?

So I'm gonna run this. Remember, X is the quote or the string that I'm processing, and I want to split it by the space. So I specify the quotes space quote here.

and you can see the output.

Now, the output is essentially an unstructured list. And if you look at this 1st item in the list

Now, the output is essentially an unstructured list. And if you look at this 1st item in the list, it's essentially a character. Vector

so what it did was, look at the string here, and then identified anywhere. There was a space.

and then anywhere. There was a space. It removed the space and extracted everything in between as its own object.

So because vary is its own string. Before a space it extracts the word vary.

then it removes the space and then extracts the next item here between 2 spaces, which is the word little. Okay.

Now, just to show you this is indeed an unstructured list, right?

And because it's an unstructured list. We can process this unstructured list in the same way we process all unstructured lists, so we can pull the 1st item. Note that this unstructured list only has a single item here. Right? So we'll pull the 1st item and remember with unstructured lists you use numeric indexing with 2 brackets. Not one.

So I'm going to pull the 1st list item, using the 2 brackets

that provides us with this character vector here, I can even show you

the class of this character vector, just to show you that the item, the 1st item in that unstructured list is a character vector.

And then within this 1st character vector, I'm going to pull the 8th item of that character. Vector, right? And it's the word happy. So if you go through and count 1, 2, 3, 4, 5, 6, 7, 8.

Happy. The string is the

8th word in this list, after splitting by the space, and then you could manipulate the strings using any other operations like we're familiar with. Right? So if I take the splitted quote by the space, I could say, Well, where in this character vector. Do I see the word happy? It provides the 8th index. Okay.

Now, the utility of this function will become more apparent as we move forward, especially when say you have a string

that has a delimiter right? And the delimiter is like a tab within a single string, or the delimiter is a space or a vertical bar. Right? You can use string split to split all of these mini strings based on that separator, or you could consider it like a delimiter.

Alright. Next up are a couple of functions which essentially concatenate strings together. And we're actually somewhat familiar with the paste function. It just paste 2 strings together. And there's also the paste 0 function shown here, which is essentially the same thing, except it just doesn't put anything between the strings, and I'll show you what I mean in a second. Right?

So both the paste and the paste 0 functions are just 2 functions in R and base r that combine

or concatenate strings. So it just places strings together right

now more specifically, pace 0 concatenate strings with no delimiter right? What I mean is, there's no separation between the strings. It simply places them right next to each other.

So here I have just

I shouldn't call it a list. But basically a sequence of words. Right? Notice, there's no C here. This is not a vector. Pace. 0 basically takes an itemized sort of listing of each of these words, so I vary comma little comma, and so on.

And you can see here that paste 0 concatenates each of these individual strings.

but with no separation in between.

Because Paste 0 says, there's no characters 0 characters in between the strings that I'm concatenating.

And this does have its use. I really want you to know. This exists mainly in the past. What I've used this for is, say, looping through a set of file names, and then I just want to paste. Say, one file one piece of a file name to another piece of a file name. Okay? So, for example, sometimes I'll say, well, I want the patient data set.

0. So I'll put 0 here and then dot Csv, and then, now I have a patient 0. 0, I'm using pay 0

   patient 0 Csv file. And then I can loop through this index here and then load files.

   Okay, so that's page 0.

   Now, paste does the same thing. It combines or concatenate strings, but with a pre-specified separation between the string

and by default the separation is a space. Okay? So here I have the same set of character data types. And I'm going to use paste. So notice, I don't have a 0 here.

and if I run the script you could see that it places a space using this Sep argument, it places a space between each of those characters. Okay.

so here, I've specifically specified this delimiter, I want a space in between each of these characters

now by default, paste already does this. So I can actually erase this and run that.

And it does the same thing, or

I can erase a space and say, I want nothing in between.

And that goes back to paste 0. Right? So very similar functions here.

Now, the separator, you can specify any sort of string or separator you'd like here I have a dash

which also can be useful for looping through file name. So here it concatenated each of these strings, but with a dash in between. Okay.

strings, but what a dash in between. Okay?

now, the thing is, what if your strings are in vector form, right? We need to specify this collapse argument.

So I'll show you what I mean here.

So this is our quote. Remember, we used string split to split this quote into a character vector where each element of the character vector is a word separated by this splitter, which is a space. Okay?

Now, if we wanted to recombine right? If this is a character vector and we wanted to recombine each of these character data types into a single string, but separate them by a space. This means we need to use the collapse argument. So we're going to collapse this character vector. Into a single character string

using the space argument.

So if I run this, I say paste

this character vector which is my quote, I'm collapsing it. using this specified string, which is just a space here

so essentially, it took every element of this character, vector concatenated them together. But putting spaces in between each word.

Now, just to illustrate other collapsing characters say you wanted a colon between each word. For some reason you could run that, and it puts a colon between each word.

So note I just this doesn't work using separation right? So if I were to try and do the same thing, it won't collapse. The quote right? It just outputs the same string. And here's the reason why

collapse specifically operates on character vectors.

Okay, character vectors.

Sep wants to concatenate 2 items or 2 characters separately. Right? And this is why, when we go to Sep, they're just listed here. This is not a character. Vector

So going down, what ends up happening is paste just paste each element with a space. But there are no other elements. Right? So it just ends up outputting the same. Vector

so long story short, if you're trying to collapse a vector of character data types into a single string.

use collapse if you're trying to concatenate multiple strings together that aren't in a character vector, right use, sep.

Alright. The next 2 functions are quite easy. It's just converting text to all uppercase and all lowercase. All right.

So there is a function called toupper which takes X as an argument. And that's our string that

So there is a function called toupper, which takes x as an argument. And that's our string that we're going to process. And there's another function called to lower, which also takes a string as an argument. Okay.

Now, if I take our original quote just to remind you what that quote looks like.

this is our quote by Marcus Aurelius.

Now, if I state to Upper

for that quote, all it does is just uppercase every single letter in that string. Okay, now notice here.

this is a better example. I think.

So I'm going to 1st make it to Upper

and then. Now, I'm going to convert it back to all lower.

Okay, so now it's all lowercase.

right? That's essentially it for the these 2 functions.

What's more important is, why would we use something like this?

Now I'll tend to use to lower when I'm processing text, and I don't want to distinguish between 2 words that have different cases. Right? So say you have a string of text. And you're searching for all of the occurrences of the phrase data science. But you don't care about punctuation. You don't care about uppercase versus lowercase. Right? You're considering all lowercase to be the same as

the 1st letter being uppercase versus the 1st letters of each word being uppercase.

So in this situation. What I would do is convert all of my text to lowercase and then search for the phrase data science. Because then there's no distinction between this phrase, this phrase, and this phrase, All right. And this is often done in natural language, processing and text processing.

Alright. Next up is our function called N. Char.

You could think of this as the number of characters, characters being a single, say, letter, symbol, and so on. All right. Now the function nchar outputs the number of single characters in a string and takes a single argument, which is the string itself represented by this argument x.

So say I were to feed it this string here: “data science,” as its argument. And I run the nchar function.

You could see here that it outputs a single number: 12. Now, if we were to count the number of characters,

there’s the D, which is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12. We have 12 characters. So this can be

quite a valuable function, showing the lengths of characters. Notice how it includes punctuation as well as spaces, and so on.

Just showing the original quote,

I'm just applying the nchar function here. Now to that original quote from earlier, we see that it has 95 characters.

Now, what's really nice about this function is it can be applied to vectors of strings. So remember, after we split the quote

after using strsplit, it outputs an unstructured list. So I'm going to pull the first item from this unstructured list using the two-bracket syntax.

This provides our character vector after splitting by the space from that quote, right?

And say I run the nchar on this character vector here, right?

What is the output?

Well, it's now a numeric vector. It's not a single value of 95, right? Because each of these elements or each of these strings is a separate element or entry within this character vector. nchar goes through almost like it's looping through each word, and it's counting the number of characters within each word.

So the word "very" has 4 characters in it, and the word "little" has 6 characters in it.

Alright, and that's nchar.

Alright. Now, the next function we're going to cover is substring

alright. And I've used this quite a bit, especially in grad school. So substring takes 3 primary arguments.

There's text which is our string that we're going to process. There's 1st and there's last.

and what the substring what the substring function does is essentially subset a string right? It subsets a portion of a string based on some starting and ending index. So it's kind of like a numeric based subsetting in a way right, except instead of using that bracket syntax like we use to subset vectors, we're explicitly sending our indexing as arguments to the function. All right.

So again, text, this 1st argument is the string that we'd like to subset.

then the 1st argument literally, 1st is the start

index of the string that you'd like to subset. And last is where you'd like to end that subset. Okay?

So if I were to go to quote

right. This is our original quote here and say, I wanted to extract this 1st portion of the string. I don't want all of this string, I just want right up until that semicolon. Well, that's 42

I don't want all of this string. I just want right up until that semicolon. Well, that's 42 characters in

right. So if I say substring this quote.

and I want to extract the 1st character, which is the V up until the 40 second character, which is the E in the word light, right?

If I were to change that to a 3, so 43. Right? That 43rd character is that semicolon. And I'm going to avoid that

right? Then I can extract the second portion of that quote

right. So if this E here is the 40 second character. Well, this colon is a 43, rd and the space in between the semicolon and the I is a 44.th So I want to extract the 45th element. Okay, so I'm going to subset this quote, and I want to take the 45th

character in this long string, and I want to end it at this period here.

Now, what is that period? Well, that's the number of characters in the string.

Right?

So that would be, we're scrolling up the 95th character is the period.

So if I were to run this subset, it pulls it all the way until the end of the quote which ends with the period. Now, if I don't want the period, you could just do a minus one here, right? Subtract one off the number of characters in the quote, and it removes the period because it goes to the 94th character in that string.

Okay.

Now, substring also enables you to overwrite portions of a string which is really nice, similar to how we would overwrite the variable names in a data frame or the row names in a data frame. Right?

So here I'm going to extract the 43, rd 44, th and 45th

characters of the string, which is the colon space I

and say, I want these as 2 separate sentences. I don't want this as a semicolon, a space and then lowercase. I I want to replace a semicolon with a period, and this I with the uppercase, I to reflect the start of a new sentence.

Well, what I can do is say, I'm going to extract these characters. And I want to replace these characters with a period, a space, and an I.

Okay. So when I do that. If I look at the original quote, I've now converted this into 2 separate sentences, where you have here a period space in uppercase, I.

All right. The next 2 functions are very powerful, and you see this re well, that stands, in a sense for regular expressions, and this is a whole other way of processing text. We're not

going to cover regular expressions today, but we can use these functions to still process text without, say formal regular expression, all right. So the 1st is called Grep, L. Right L. Stands for logical, and then just a regular grep.

So let's cover grep first.st

Now this function, Grep

takes 2 arguments, pattern and x, so it kind of reverses in syntax. Remember, in the prior functions we usually have the string 1st and then a pattern. Well, in this graph function we have our pattern 1st and then X, so just be aware of that.

And what the Gret function does is it? Returns the index of entries that contain some pattern that you specify to the function. And obviously this is best illustrated by example. So just to reiterate this pattern argument

is, you're checking if X contains that pattern. So you provide that pattern that you're searching for, and X is just the character or the string that you're processing.

So let's apply Grep to a version of that quote. We were working with earlier. And what I'm doing here is, I'm converting this quote into a, vector all right. So I'm just going to have a vector of 2 elements

with the vector of length 2.

The 1st element is just this 1st portion of the quote. So I can do that just extract. The 1st element, which is that 1st half of the quote.

second element is the second half of the quote. Okay.

now, the script below uses this graph function, this regular expressions function

to determine if the string happy right? This is the pattern I'm searching for.

If the string happy is contained within the quote. In fact, I can put

pattern here just to make sure we know that

that is the pattern we're searching for.

Okay, I'll go ahead and run this. And again.

this function I'm using to determine if this pattern string. Happy is contained within the quote.

Okay, now, the output of this function is one. Why? Because the Gret function returns the index

of the character vector entry that contains the string pattern happy.

So if we scroll up the 1st entry in this, vector

is the 1st half of the sentence, right happy is contained within this 1st entry of the vector.

Now, if I were to say, put happy in the second, one.

Grep will output one and 2, because the string pattern, Happy is in both the 1st element of

Grep will output one and 2, because the string pattern. Happy is in both the 1st element of that vector and the second element of that. Vector okay, I'm just going to undo that

perfect.

Now, what do you think the output is here? Right? If I'm searching for thinking, well, we know thinking is in the second

element of this vector right? It's the second index of that vector right here.

But we can run that. And it simply outputs 2.

Okay, very powerful function. Right? So if you have a vector of strings. You can use graph to identify which elements of that vector contain a certain pattern.

And you could get pretty fancy with the types of patterns you specify.

Now, following up with that there's Grep. L, which is just essentially the same function, except it outputs a logical, a true, based on. If that string you're processing actually contains the pattern you're searching for. All right. So again, we have 2 arguments here pattern, which is, you are checking. If X contains this pattern.

and then X is the actual string that you're processing. Okay.

so this is the original quote.

then I'm going to use graph L to determine

if the 1st vector contains the string happy. And if the second vector contains the string happy.

So if I run that

it outputs a logical data type or a logical vector where the 1st element of this logical vector is a true why? Because the 1st element of this character vector contains the word happy.

The second element is a false, because the second element of this, vector this character vector does not contain the word happy right? And because of this, you can actually use subsetting to extract what you need from the quote right? So based on what we know about subsetting, I want to extract every element of this character vector, that's true. But I could say, quote.

and it will extract that 1st item.

Okay.

now, going down right, we're going to get a false, because the word thinking is not in the 1st element of that character. Vector however, it is in the second element of this character. Vector right? So we end up getting a true here.

And if I check if the phrase data science, which wasn't a thing back in ancient Rome, right? So it's false in both quotes because the phrase data science does not appear in either entry.

neither the 1st or second entry of that character vector.

Alright. So that's plenty of functions there. And the last 2 functions that I'd like to cover for string processing are gsub and gregexpr — so this is like G regular expression.

First, gsub.

Now, gsub is quite easy. It's just replacing some pattern in a string x with your other string, which we call a replacement. Right? So this function gsub takes 3 arguments.

And it replaces a pattern with another specified pattern or string we call replacement in some overall string x. Right? So the pattern is what you're checking for in the string x.

The replacement is what you're replacing the pattern with.

And x is the string you're processing. Alright.

So here I'll just show you the quote again.

And the quote contains "happy" here.

So what gsub (sub standing for substitute) wants to do is search this quote for the pattern, right?

So let me just write here: pattern, replacement, and then here x, just so we know which arguments go where.

All right.

So what it's going to do is look through this quote, find wherever it says "happy" like here, and it's going to replace that word with the word "fulfilling."

Okay?

So it basically removed the word "happy," which is our pattern, and then replaced it with our replacement argument, which is the word "fulfilling." And now we have this new quote: "Very little is needed to make a fulfilling life."

Okay, and that's pretty much it for gsub. It's just used to replace portions of text within your strings.

Now, moving to this other gregexpr — for G regular expression.

This takes 2 arguments.

And it's quite powerful. It returns the position in the string of all occurrences of a certain pattern. Right? So you have this text argument,

You have this text argument, and you're searching through this text argument for this pattern, and then it returns the location of the pattern within the strings.

Okay, so if I run this,

the word "happy" appears in the 33rd position here.

Alright. So if I run this, I want to run gregexpr. I specify my pattern as "happy" — so I'm

searching for "happy", right? Then I have text here,

which is my quote. Now, where does "happy" occur? Well, it occurs in this first entry of this character vector.

And the "h" occurs in the 33rd character. So if you were to count from "V" all the way to the "h", the "h" is the 33rd character here.

All right.

Now, the second entry of the vector does not contain the string pattern "happy." So it outputs a -1, because it's basically telling us it doesn't occur anywhere in the second string here.

Now, how can we use this? Well, one way is you can extract certain words, right? So say we have our quote. This is our first entry vector, because the word "happy" occurs in the 33rd

character.

So you can see we have the string here, and we want to extract the 33rd character, which is the "h",

using substring all the way to the 33rd plus the nchar of "happy". All right.

So let me break this down. nchar extracts the number of characters — the number of characters in the word "happy" is 5, right? So we basically want to extract the 33rd character

to the 33 + 5, which is 38, minus one. So characters 33 to 37 would extract this word here — 33 being the "h", 37 being the "y".

And if I use substring to do that, we use many of these functions to extract, right? And now we have our original quote "happy" here. So this is just one way you can use it.

Alright. So those are all the functions that I pretty much want you to know from base R. That can be used very powerfully, I might add, for string manipulation and string processing, right? But that's not everything.

There's regular expressions, but there's also something called escape characters which are important to know.

So thus far to define any sort of character, data type or character vector, we've always used quotes. Right? So the way R processes this is, it sees a quote and says, “Okay, this person's trying to start or define a character,” and then it sees the end quote and says, “Alright. They're done defining the character. I'm going to move on in the program.”

And the same goes for apostrophes. Right?

But what if the string itself has a quote in it?

What if we want to create a string that shows the height: 5 foot 8, and it has a quote. But we want this whole thing to be in quotes. Alright. So let's attempt this. It looks a little weird.

Here we have a quote.

and then we have 5 and an apostrophe which

resembles or represents the foot. Right? So it's 5 feet and 8 inches, the inch mark represented by the quote,

and then we have close quotes. We close the quotes because that's the end of the quote, "5 foot 8 inches." So I'm going to run this,

and notice, here we encounter an error,

and it says "incomplete string." Right? So I'm going to do the same with a single quote.

A single quote — this is an incomplete string.

Alright. So what's happening here is R sees the first quote and thinks, “Alright. They're starting a character.”

"5'8" is part of the character, and then it closes the character using a quote. And then R thinks, “Well, there's this other quote here. This person wants to start another character,” but we never ended it. It doesn't know that this quotation mark

representing inches is something that we want to explicitly include as a quotation.

Okay, so how do we reconcile this?

Well, we need to explicitly tell R: this quotation is not defining a string. This quotation is defining something that I want in my text. It's defining our inches measurement. It's not ending the string.

The way we can tell R this is by using escaping — and we use an escape character.

We do so by including a backslash before the quotation. This is known as escaping, since this backslash escapes the default usage of that quote.

Alright. So here, remember we had an error.

I can put a backslash right behind that quote to tell R, "Whatever comes after this backslash is a character I want inside the quote."

And it ends up running that. So there's no error there. The notation looks a little weird — you see this backslash here — but it at least puts the quote there. Let me just check one more thing.

Okay, yeah.

So now it runs without error, right? But we still have some weird formatting issues.

Now, what if we want to include a backslash in the string itself? Well, we can escape the backslash.

So if I run this,

R doesn't know what to do because the backslash is a loaded character. The backslash

means something following that is going to be something I'm trying to escape. Right? So to escape the escape character — to escape the backslash — you put another backslash. So this says, “I'm going to escape

the proper use of a backslash.”

And if I run that, it ends up putting the two backslashes there.

Now, again, the formatting is a little awkward,

so I'll cover the cat function now.

Now the cat function stands for concatenate, and it's very similar to the print function, except when it prints strings, it prints them using the formatting specified using these escape characters.

So notice how, if I print "5 foot 8,"

it prints that backslash with it.

But in actual text, we don't need the backslash there, right? We don't want these two slashes.

So if you instead use cat instead of print,

it'll remove the quotes on the outside and just print it as if it were text. Right? So we're concatenating this text together. We have "5 foot 8."

It knows that everything in between the outer quotes is a quote, and then it uses the escape to just represent this backslash quote as a single quote.

Same thing with the two backslashes here.

Now, more common escapes are \n — you'll see this a lot when you load text data into R. This means new line.

There's \t, which stands for tab.

And again, something we've already covered: " for quotes, and it also works for a single quote or apostrophe. Alright. So let me just show you an example of this.

I'm going to redefine my quote here. This is a quote from the original earlier video.

I'm going to use strsplit because I want to split this quote

by the space so that I have a single vector or an unstructured list where the first element is a character vector — where each element of that character vector

is a separate string.

Right?

I'm going to then combine this string using paste. But remember, this is a character vector.

quote_split[[1]] is a character vector, so I'm going to use collapse, and I'm going to collapse them —

not with a space. Right? If I collapse them with a space, I end up getting the original quote. But here I'm going to collapse them with an escape new line — so this new line break.

Okay?

And it doesn't look very nice. It basically places a \n in between every single word.

But remember that backslash is an escape character. It's telling R, “After this word 'very' go to a new line and then write the word 'little'.”

Now, if I were to print this new quote,

you'll see that it just preserves that \n. Doesn't really look very good. But here we can use cat to print the text as we would read it.

And you can see here it prints "very" then goes to a new line based on this \n escape,

prints the word "little", goes to a new line, and so on. So essentially, it allows you to incorporate formatting in your strings.