String Manipulation

http://datascience.tntlab.org

Module 6

Today's Agenda

- Best practices for strings in R
  - Code formatting
  - Escaping
  - Formatting
- Base R string construction
- Importing strings with *stringi*
- Pattern matching and string operations with *stringr*
- Incorporation of string functions into *magrittr* pipes

Best Practices for Representing Strings

- Use " " by default.
- If your string has a " in it, use ' '
- If your string has both " and ' in it, use " " and escape it as a literal using \"

- Examples
  - "This is your default sentence"
  - 'This is for if you need "quotes"'
  - "This is if you need both \"quotes\" and 'quotes"'

- Do not confuse " " and ' ' with “this” and ‘this’
  - Smart quotes are evil

- Use base-R's writeLines() to check what strings really look like

Escape Sequences

- Used any time to tell R "don't interpret this like you normally would"
  - "\"" becomes "
  - "\\" becomes \
  - "\t" becomes a tab
  - "\n" becomes a new line
  - \' becomes a literal single quote
  - \" becomes a literal double quote
- You can see the full list with ?Quotes

- Can also use Unicode code charts (http://www.unicode.org/charts/) but this won't work in R Studio

Number Formatting

- VERY IMPORTANT FOR US, because this is related to how you force an output table to always have the same number of decimal places
- "Digits" option specifies the number of significant digits in the most precise number when displaying output; the precision of all other simultaneously displayed numbers is then matched to that
- Note that R has default options for settings like this
  - To view: getOption("digits")
  - To set: options(digits = 7)
- format() can be used to pre-specify this by converting numbers into strings, but it takes a lot of trial and error
  - Do remember the scientific=FALSE and big.mark="," parameters
- You will usually want formatC(x, format="f", digits=2)

Pasting

- paste() converts its parameters to text and then sticks them together as a vector of strings
  - sep specifies what goes between the parameters
  - collapse specifies 1) you want to combine everything into a single string and 2) what you want to put between the strings

- paste0() is the same as paste(sep="", …)

- Don't confuse c() and paste()
  - Paste combines elements of vectors specified by its parameters
  - Combine creates new vectors using its parameters as elements

- Most useful if you *are not* already using *stringr*

Getting Text Documents into R with *stringi*

- **stri_read_lines**() from *stringi* is best for pure text documents
  - You know there is only text on every line
  - You can probably configure read_delim to figure it out, but don't try
  - Imports as a vector of strings ("large character vector")

- **stri_isempty**()
  - Returns a logical vector
  - Useful to filter cases imported by stri_read_lines()

String Concatenation Using *stringr*

- Part of the *non-core tidyverse*, so you must call *stringr*
  - All *stringr* functions start str_ -- VERY IMPORTANT

- **str_c**() replaces paste0()
  - Will act more like paste() with sep parameter
  - Handles missing values differently (very useful)
  - Like paste, propagates vectors as long as it's possible
    - nums <- c(1:5); paste("Participant", 1:5);
    - nums <- c(1:5); paste0("Participant", 1:5);
    - nums <- c(1:5); str_c("Participant", 1:5);

String Queries

- **str_length**(string)
  - Very different from length(), which returns *vector* length
  - Very similar to nchar(), but will return character length of factor labels

- **str_sub**(string, start, end)
  - Similar to substr(), but understands negative indexes
  - Extracts information from string based upon location within the strength, returning all characters between positions start and end
  - *You can't go backwards*

  - Examples
    - str_sub("ABCDEF", 2, 3)
    - str_sub("ABCDEF", -2, -1)
    - str_sub("ABCDEF", -5, 3)
    - str_sub("ABCDEF", -3, 3)

  - Start defaults to 1 and end to -1, so what does this do? str_sub("ABCDEF", , -2)

String Searches / Pattern Matching

- **str_detect**(string, pattern)
  - Patterns are regular expressions (regex); the simplest regex is plain text
  - Returns logical vector if detected, which is useful for filtering later
  - str_subset() is the same as x[str_detect(x, pattern)]

  - Use str_subset() to subset *vectors*
  - Use filter(tbl, str_detect()) to subset *tbls*

- **str_extract**(string, pattern) and **str_match**(string, pattern)
  - Follows the same rules as str_detect, but returns a matrix or list of matches and captures
  - Also remember **str_extract_all** and **str_match_all**
- **str_count**(string, pattern)
  - Follows the same rules as str_detect, but returns an integer
- **str_view**(string, pattern)
  - Shows a webpage with all strings and how they were matched; useful diagnostically
  - Requires separate install of packages *htmltools* and *htmlwidgets* for R Studio
  - Also remember str_view_all()

String Operations

- **str_split**(string, pattern, n, simplify)
  - Takes a string and creates a list unless simplify = TRUE, in which case a matrix is returned
- **str_replace**(string, pattern, replacement)
  - Takes strings, replaces the **first matching pattern** with replacement, and returns modified original
  - If you need to replace *all matches*, use str_replace_all(string, pattern, replacement)
- **str_trim**(string, side="both")
  - Trim whitespace
- **str_tolower(), str_to_upper, str_to_title()**
  - Changes cases as needed, a common preprocessing step

- Remember that all *stringr* functions *operate on strings,* not tibbles

Regex

- Made easier in some circumstances by *rebus*, but this requires you to learn *rebus* syntax, which only works in the R package *rebus*
  - *rebus* constructs a regex expression, which you can look at to help learn
  - https://www.rstudio.com/wp-content/uploads/2016/09/RegExCheatsheet.pdf
  - Type rebus:: in R Studio to get a quick list of constants and functions

- When creating regex, scalability is critical
  - When I test your regexes, I will do so with strings you don't have access to but following the same data generation techniques
  - As long as you match what you're supposed to and don't match things you're not supposed to, that's correct
  - Recommend you create your own mini-datasets for testing

- **WARNING:** ALWAYS CASE SENSITIVE.

Metacharacters

- From regexone.com

Lesson Notes

abc…	*Letters*
123…	*Digits*
\d	*Any Digit*
\D	*Any Non-digit character*
.	*Any Character*
\.	*Period*
[abc]	*Only a, b, or c*
[^abc]	*Not a, b, nor c*
[a-z]	*Characters a to z*
[0-9]	*Numbers 0 to 9*
\w	*Any Alphanumeric character*
\W	*Any Non-alphanumeric character*
{m}	*m Repetitions*
{m,n}	*m to n Repetitions*
*	*Zero or more repetitions*
+	*One or more repetitions*
?	*Optional character*
\s	*Any Whitespace*
\S	*Any Non-whitespace character*
^…$	*Starts and ends*
(…)	*Capture Group*
(a(bc))	*Capture Sub-group*
(.*)	*Capture all*
(abc|def)	*Matches abc or def*

Common Rebus Constants and Functions

- ALPHA
- ALNUM
- BLANK
- DIGIT (DGT) and NOT_DGT
- LOWER and UPPER
- PUNCT
- SPACE (SPC) and NOT SPC
- ANY_CHAR
- WRD and NOT_WRD

- Don't worry about UNICODE unless you need it

- Any standard *regex* character has its own *rebus* constant, e.g.,
  - DOLLAR and CARET
  - BACKSLASH
  - OPEN_PAREN and CLOSED_PAREN
  - Check "NAMESPACE" on github

- And functions, e.g.,
  - char_class()
  - capture()
  - repeated()
  - optional()
  - zero_or_more()
  - one_or_more()
  - negated_char_class() (same as ^)
  - exactly() (same as START, END)

Common Problems with *rebus*

- When you specify a repeating function (i.e., one_or_more(), zero_or_more()), *rebus* usually assumes you want to specify it as a character class: [ ]

- However, when you put a metacharacter into a character class, it becomes a literal
  - By default, **one_or_more(ANY_CHAR)** becomes [.]+ instead of .+
  - .+ is interpreted as "one or more of any character"
  - [.]+ is interpreted as "one or more dots"

- You can override this with char_class=FALSE, e.g.,
  - one_or_more(ANY_CHAR, char_class=FALSE)

- Another common error is forgetting that *rebus* (and *regex*) are greedy:
  - You probably don't mean: one_or_more(ANY_CHAR, char_class=FALSE) %R% DOT
  - You probably want the inverse: one_or_more(negated_char_class(DOT))

Captures and Backreferences

- Capturing returns pieces of matched regexes, e.g.,
  - "I want four-digit numbers, but only when they occur after the letters "Yr: "
  - *rebus*: "Yr: " %R% capture(DGT %R% DGT %R% DGT %R% DGT)
  - *regex:* "Yr: (\d\d\d\d)"
  - *regex:* "Yr: (\d{4})"

- **str_match** will return both the full match and each capture match as a matrix (contrast with str_extract, which only returns the full match)

- Backreferences uses a capture as a term in the regex
  - Uses REF1 – REF9 constants in rebus or \1 - \9 in regex
  - Useful to *detect and return* repeated information

Common Problems with *regex*

- When passing a string as the pattern for a regex, the R string processor processes the pattern *as a string* before the regex processor ever sees it.
- This means that any characters that are special in *both* R *and* regex may need double-escaping: $ * + . ? [ ] ^ { } | ( ) \
- *rebus* does help here, as long as you remember to use *rebus* constants

- Example
  - str_detect("This is a sentence.", ".") # does not do what you think it does
  - str_detect("This is a sentence.", "\.") # gets to regex, but regex sees a literal
  - str_detect("This is a sentence.", "\\.") # correct form (\ is escaped by R)
  - str_detect("This is a sentence.", fixed(".")) # also correct but doesn't work if # mixing fixed and non-fixed

Incorporating into *dplyr/magrittr*

- Remember that *stringr* functions (str_) use strings as input and produce strings as output
- In a *magrittr* pipe, you should generally only pass tibbles
- Therefore, in a *magrittr* pipe, stringr functions should be parameters of a *dplyr* verb

- Example
  - library(magrittr)
  - start_df %<>%
    mutate(lowerText = str_to_lower(existing_text)) %>%
    mutate(verbFound = str_match(lowerText, pattern)[,2])