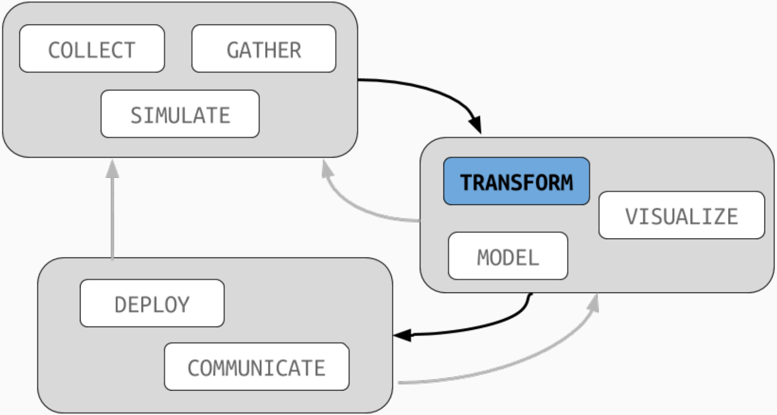
Lecture 18: Strings in R

Taylor Arnold


COLLECT
GATHER
SIMULATE
TRANSFORM
VISUALIZE
MODEL
DEPLOY
COMMUNICATE

Today we will cover the main aspects of working with raw strings in R using the **stringi** package. To load the package call:

```
library(stringi)
```

stringi

The main advantages of this package over this package compared to those in base-R are:

- consistent syntax - the string you are operating on is always the first element and functions all start with `stri_`
- great support for non-latin character sets and proper UTF-8 handling
- in some cases much faster than alternatives

stringi

We will work with two datasets that come pre-installed with **stringr** (a wrapper around **stringi**), a list of common English tokens named `words` and a list of short sentences named `sentences`. We will wrap these up as data frames in order to make them usable by the **dplyr** verbs we have been learning:

```
df_words <- data_frame(words = stringr::words)
df_sent <- data_frame(sent = stringr::sentences)
```

stri_sub

The first function we will look at is `stri_sub` that takes a substring of each input by position; for example the following finds the first three characters of every string in the data set of words:

```
mutate(df_words, chars = stri_sub(words, 1, 3))
```

```
## # A tibble: 980 x 2
##      words chars
##      <chr> <chr>
## 1        a     a
## 2     able   abl
## 3    about   abo
## 4 absolute   abs
## 5   accept   acc
## 6  account   acc
## # ... with 974 more rows
```

stri_sub

Notice that R silently ignores the fact that the first word that has only one letter (it is returned as-is). We can use negative values to begin at the end of the string (-1 is the last character, -2 the second to last and so on). So the last two characters can be grabbed with this:

```
mutate(df_words, chars = stri_sub(words, -2, -1))
```

```
## # A tibble: 980 x 2
##      words chars
##      <chr> <chr>
## 1        a     a
## 2     able    le
## 3    about    ut
## 4 absolute    te
## 5   accept    pt
## 6  account    nt
## # ... with 974 more rows
```

stri_length

The function `stri_length` describes how many characters are in a string:

```
mutate(df_words, num_char = stri_length(words))
```

```
## # A tibble: 980 x 2
##       words num_char
##       <chr>    <int>
## 1         a        1
## 2      able        4
## 3     about        5
## 4  absolute        8
## 5    accept        6
## 6   account        7
## # ... with 974 more rows
```

stri_trans_toupper

And the functions `stri_trans_toupper` and `stri_trans_tolower` do exactly as they describe:

```
mutate(df_words, up = stri_trans_toupper(words), down = stri_trans_tolo
```

```
## # A tibble: 980 x 3
##      words       up     down
##      <chr>    <chr>    <chr>
## 1        a        A        a
## 2     able     ABLE     able
## 3    about    ABOUT    about
## 4 absolute ABSOLUTE absolute
## 5   accept   ACCEPT   accept
## 6  account  ACCOUNT  account
## # ... with 974 more rows
```

stri_trans_totitle

We even have `stri_trans_totitle` to convert to title case:

```
stri_trans_totitle("The birch canoe slid on the smooth planks.")
```

```
## [1] "The Birch Canoe Slid On The Smooth Planks."
```

matching fixed strings

stri_detect

A function that finds patterns is the function `stri_detect`, which returns either `TRUE` or `FALSE` for whether an element has a string withing in. We can use this conjunction with the `filter` command to find examples with a particular string in it:

```
filter(df_sent, stri_detect(sent, fixed = "hand"))
```

```
## # A tibble: 4 x 1
##                                           sent
##                                          <chr>
## 1 Weave the carpet on the right hand side.
## 2 Hedge apples may stain your hands green.
## 3    Shake hands with this friendly child.
## 4        Many hands help get the job done.
```

stri_count

Similarly `stri_count` tells us how often a sentence uses a particular string. For instance, how many times are the digraphs "th", "ch", and "sh" used in each sentence:

```
temp <-
mutate(df_sent, th = stri_count(sent, fixed = "th"),
                sh = stri_count(sent, fixed = "sh"),
                ch = stri_count(sent, fixed = "ch"),
                sent = stri_sub(sent, 1, 20))
```

I took a substring of the first column to make it fit on the page.

stri_count

```
temp
```

```
## # A tibble: 720 x 4
##                   sent    th    sh    ch
##                  <chr> <int> <int> <int>
## 1 The birch canoe slid     2     0     1
## 2 Glue the sheet to th     2     1     0
## 3 It's easy to tell th     2     0     0
## 4 These days a chicken     0     1     1
## 5 Rice is often served     0     0     0
## 6 The juice of lemons      0     0     1
## # ... with 714 more rows
```

stri_replace_all

The function `stri_replace_all` replaces one pattern with another. Perhaps we want to replace all of those borning “e”’s with “ë”:

```
mutate(df_sent, sent = stri_replace_all(sent, "ë", fixed = "e"))
```

```
## # A tibble: 720 x 1
##                                              sent
##                                             <chr>
## 1  Thë birch canoë slid on thë smooth planks.
## 2 Gluë thë shëët to thë dark bluë background.
## 3      It's ëasy to tëll thë dëpth of a wëll.
## 4    Thësë days a chickën lëg is a rarë dish.
## 5        Ricë is oftën sërvëd in round bowls.
## 6       Thë juicë of lëmons makës finë punch.
## # ... with 714 more rows
```

stri_replace

The function `stri_replace` without the “all” only replaces the first occurrence in each string.

matching patterns

Trying to use the previous functions with a fixed string can be useful, but the true strength of these functions come from their ability to accept a pattern known as a regular expression.

We don't have time to cover these in great detail, but will show a few important examples. The first example we will us is the "." symbol which matches any character.

patterns

So, for instance the following finds any time that we have the letters "w" and "s" separated by any third character. Can you find where this occurs in each line?

```
filter(df_sent, stri_detect(sent, regex = "w.s"))
```

```
## # A tibble: 81 x 1
##                                              sent
##                                             <chr>
## 1           Rice is often served in round bowls.
## 2   The box was thrown beside the parked truck.
## 3           The boy was there when the sun rose.
## 4 The fish twisted and turned on the bent hook.
## 5       The swan dive was far short of perfect.
## 6 The beauty of the view stunned the young boy.
## # ... with 75 more rows
```

anchors

Two other special characters are “^” and “$”, called *anchors*. The first matches the start of a sentence and the second matches the end of a sentence. So, which words end with the letter “w”?

```
filter(df_words, stri_detect(words, regex = "w$"))
```

```
## # A tibble: 18 x 1
##    words
##    <chr>
## 1  allow
## 2   blow
## 3   draw
## 4    few
## 5 follow
## 6   grow
## # ... with 12 more rows
```

anchors

Or start with “sh”?

```
filter(df_words, stri_detect(words, regex = "^sh"))
```

```
## # A tibble: 11 x 1
##   words
##   <chr>
## 1 shall
## 2 share
## 3   she
## 4 sheet
## 5  shoe
## 6 shoot
## # ... with 5 more rows
```

stri_extract

There is on other **stringi** function we did not mention earlier: `stri_extract`. Given a pattern it returns the string that matches it. This is not very useful without regular expression but with them is an invaluable tool.

stri_extract

For example, what characters follow the pattern “th”?

```r
temp <- mutate(df_sent, triple = stri_extract(sent, regex = "th."))
count(temp, triple, sort = TRUE)
```

```
## # A tibble: 10 x 2
##   triple     n
##    <chr> <int>
## 1    the   378
## 2   <NA>   229
## 3    th     47
## 4    tha    23
## 5    thi    20
## 6    thr    12
## # ... with 4 more rows
```

HTML tags

There are many other more complex regular expressions. For example, this one is very useful:

```
stri_replace(html, " ", regex = "<[^>]+>")
```

If `html` is a string, this will replace all of the characters in html tags with a single space. We will use that in our lab today.