Handout 09: Strings in R

Taylor Arnold

Basic string manipulation

This handout covers the main aspects of working with raw strings in R using the **stringr** package. This is part of the tidyverse suite, but not loaded by default. To load the package call:

```
library(stringr)
```

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 `str_`
- great support for non-latin character sets and proper UTF-8 handling
- in some cases much faster than alternatives

We will work with two datasets that come pre-installed with **stringr**, 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 = words)
df_sent <- data_frame(sent = sentences)
```

The first function we will look at is `str_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 = str_sub(words, 1, 3))
```

```
## # A tibble: 980 × 2
##       words chars
##       <chr> <chr>
## 1         a     a
## 2      able   abl
## 3     about   abo
## 4  absolute   abs
## 5    accept   acc
## 6   account   acc
## 7   achieve   ach
## 8    across   acr
## 9       act   act
## 10   active   act
## # ... with 970 more rows
```

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 = str_sub(words, -2, -1))
```

```
## # A tibble: 980 × 2
##       words chars
##       <chr> <chr>
## 1         a     a
## 2      able    le
## 3     about    ut
## 4  absolute    te
## 5    accept    pt
## 6   account    nt
## 7   achieve    ve
## 8    across    ss
## 9       act    ct
## 10   active    ve
## # ... with 970 more rows
```

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

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

```
## # A tibble: 980 × 2
##       words num_char
##       <chr>    <int>
## 1         a        1
## 2      able        4
## 3     about        5
## 4  absolute        8
## 5    accept        6
## 6   account        7
## 7   achieve        7
## 8    across        6
## 9       act        3
## 10   active        6
## # ... with 970 more rows
```

And the functions `str_to_lower` and `str_to_upper` do exactly as they describe:

```
mutate(df_words, up = str_to_upper(words), down = str_to_lower(words))
```

```
## # A tibble: 980 × 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
## 7   achieve  ACHIEVE  achieve
## 8    across   ACROSS   across
## 9       act      ACT      act
## 10   active   ACTIVE   active
## # ... with 970 more rows
```

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

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

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

Matching strings

We now move on to some more complex string functions that require matching some type of pattern, starting with very simply examples. Take the function `str_view_all` that visually displays all occurrences of a pattern in RStudio:

```
str_view_all(df_sent$sent, "is")
```

Notice that you can select to *show in new window* the results in the View pane of RStudio to see all of the results. Notice that this pattern reveals the word "is" but also other occurrences where this string occurs within a word.

Another function that finds patterns is the function `str_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, str_detect(sent, "hand"))
```

```
## # A tibble: 4 × 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.
```

Similarly `str_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:

```
mutate(df_sent, th = str_count(sent, "th"),
                sh = str_count(sent, "sh"),
                ch = str_count(sent, "ch"),
                sent = str_sub(sent, 1, 20))
```

```
## # A tibble: 720 × 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
## 7  The box was thrown b     2     0     0
## 8  The hogs were fed ch     0     0     1
## 9  Four hours of steady     0     0     0
## 10 Large size in stocki     0     0     0
## # ... with 710 more rows
```

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

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

```
mutate(df_sent, sent = str_replace_all(sent, "e", "ë"))
```

```
## # A tibble: 720 × 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.
## 7  Thë box was thrown bësidë thë parkëd truck.
## 8  Thë hogs wërë fëd choppëd corn and garbagë.
## 9          Four hours of stëady work facëd us.
## 10    Largë sizë in stockings is hard to sëll.
## # ... with 710 more rows
```

The function `str_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. So, for instance this finds any time that we have the letters "w" and "s" separated by any third character:

```
filter(df_sent, str_detect(sent, "w.s"))
```

```
## # A tibble: 81 × 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.
## 7           Her purse was full of useless trash.
## 8    The wrist was badly strained and hung limp.
## 9      The meal was cooked before the bell rang.
## 10    The ship was torn apart on the sharp reef.
## # ... with 71 more rows
```

Can you find where this occurs in each line? 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, str_detect(words, "w$"))
```

```
## # A tibble: 18 × 1
##       words
##       <chr>
## 1     allow
## 2      blow
## 3      draw
## 4       few
## 5    follow
## 6      grow
## 7       how
## 8      know
## 9       law
## 10      low
## 11      new
## 12      now
## 13     show
## 14     slow
## 15    throw
## 16 tomorrow
## 17     view
## 18   window
```

Or start with "sh"?

```
filter(df_words, str_detect(words, "^sh"))
```

```
## # A tibble: 11 × 1
##     words
##     <chr>
## 1   shall
## 2   share
## 3     she
## 4   sheet
## 5    shoe
## 6   shoot
## 7    shop
## 8   short
## 9  should
## 10   show
## 11   shut
```

How would we actually match a literal period, dollar sign, or other special character? The answer is to put two slashes before it; so this is how to replace all of the periods in the sentences into exclamation marks:

```
mutate(df_sent, sent = str_replace(sent, "\\.", "!"))
```

```
## # A tibble: 720 × 1
##                                            sent
##                                           <chr>
## 1   The birch canoe slid on the smooth planks!
## 2  Glue the sheet to the dark blue background!
## 3       It’s easy to tell the depth of a well!
## 4     These days a chicken leg is a rare dish!
## 5         Rice is often served in round bowls!
## 6        The juice of lemons makes fine punch!
## 7  The box was thrown beside the parked truck!
## 8  The hogs were fed chopped corn and garbage!
## 9          Four hours of steady work faced us!
## 10    Large size in stockings is hard to sell!
## # ... with 710 more rows
```

There is on other **string** function we did not mention earlier: `str_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. For example, what characters follow the pattern "th"?

```
temp <- mutate(df_sent, triple = str_extract(sent, "th."))
table(temp$triple)
```

```
##
## th  th. tha the thi tho thr thu thy
##  47   3  23 378  20   6  12   1   1
```

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

```
str_replace(html, "<[^>]+>", " ")
```

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.