Introduction to Strings

MSDS 597 Data Wrangling & Husbandry

2/10/2020

Strings

- Boehmke and Wickham & Grolemund both have good chapters on handling strings and on regular expressions.
- There are good functions in base R and the tidyverse package `stringr` to handle strings.
- The `stringr` package expands the base R functions, but also tries to give a uniform syntax.

`strsplit()` and `str_split()`

These functions will split a character string (first argument) into a list of pieces, based on the pattern (second argument) for where to split. Notice the result is a list.

```
example <- "Call me Ishmael"
strsplit(example, " ")
```

```
## [[1]]
## [1] "Call"    "me"      "Ishmael"
```

```
library(stringr)
str_split(example, " ")
```

```
## [[1]]
## [1] "Call"    "me"      "Ishmael"
```

- `tolower()` and `toupper()` change the case, as do `str_to_lower()`, `str_to_upper()` and `str_to_title()`
- `nchar()` and `str_length()` count the number of characters, but the former handles gives `nchar(NA) = 2` in R <= 3.2.0

```
example2 <- c(strsplit(example, " ")[[1]], NA)
tolower(example2)
```

```
## [1] "call"    "me"      "ishmael" NA
```

```
toupper(example2)
```

```
## [1] "CALL"    "ME"      "ISHMAEL" NA
```

```
example2
```

```
## [1] "Call"    "me"      "Ishmael" NA
```

```
nchar(example2)
```

```
## [1]  4  2  7 NA
```

```
str_length(example2)
```

```
## [1]  4  2  7 NA
```

Substrings can be selected with `substr()`, `substring()`, or `str_sub()`.

```
substr(example2, 2, 4)
```

```
## [1] "all" "e"   "shm" NA
```

```
str_sub(example2, 2, 4)
```

```
## [1] "all" "e"   "shm" NA
```

```
x <- "BBCDEF"
str_sub(x, 1, 1) <- "A"
x
```

```
## [1] "ABCDEF"
```

`paste()` and `str_c()`

These two functions concatenate one or more character strings. The `sep` argument gives the separator (default `sep = " "` for `paste()` and `sep = ""` for `str_c()` and `paste0()`) while the `collapse` argument indicates how strings are joined.

```
paste("Call", "me")
```

```
## [1] "Call me"
```

```
paste(example2[-4], collapse = " ")
```

```
## [1] "Call me Ishmael"
```

“To understand how str_c works, you need to imagine that you are building up a matrix of strings. . . .”

```
str_c(LETTERS[1:3], letters[1:3])
```

```
## [1] "Aa" "Bb" "Cc"
```

```
str_c(LETTERS[1:3], letters[1:3], sep = " ")
```

```
## [1] "A a" "B b" "C c"
```

```
str_c(LETTERS[1:6], letters[1:3])
```

```
## [1] "Aa" "Bb" "Cc" "Da" "Eb" "Fc"
```

```
str_c(LETTERS[1:6], letters[1:3], collapse = "+")
```

```
## [1] "Aa+Bb+Cc+Da+Eb+Fc"
```

trimming and padding

`str_trim()` removes leading and trailing white space

```
str_trim(" Ishmael  ")
```

```
## [1] "Ishmael"
```

while `str_pad()` will add characters to the beginning, end, or both.

```
str_pad(c("7", "321", "42"), width = 3, side = "left", pad = "0")
```

```
## [1] "007" "321" "042"
```

```
str_pad(c(7, 321, 42), width = 3, side = "left", pad = "0")
```

```
## [1] "007" "321" "042"
```

Pattern matching and replacement

The functions `gsub(pattern, replacement, string)` and `str_replace(string, pattern, replacement)` will replace substrings with other substrings.

```
sub("our", "or", c("colour", "flavour", "red is a colour not a flavour"))
```

```
## [1] "color"                        "flavor"
## [3] "red is a color not a flavour"
```

```
gsub("our", "or", c("colour", "flavour", "red is a colour not a flavour"))
```

```
## [1] "color"                       "flavor"
## [3] "red is a color not a flavor"
```

```
str_replace(
  c("colour", "flavour", "red is a colour not a flavour"),
  "our",
  "or"
)
```

```
## [1] "color"                        "flavor"
## [3] "red is a color not a flavour"
```

```
str_replace_all(
  c("colour", "flavour", "red is a colour not a flavour"),
  "our",
  "or"
)
```

```
## [1] "color"                       "flavor"
## [3] "red is a color not a flavor"
```

The patterns in these functions can be regular expressions, which provide a power tool for pattern matching.