stringr: modern, consistent string processing

by Hadley Wickham

Abstract String processing is not glamorous, but it is frequently used in data cleaning and preparation. The existing string functions in R are powerful, but not friendly. To remedy this, the **stringr** package provides string functions that are simpler and more consistent, and also fixes some functionality that R is missing compared to other programming languages.

Introduction

Strings are not glamorous, high-profile components of R, but they do play a big role in many data cleaning and preparations tasks. R provides a solid set of string operations, but because they have grown organically over time, they can be inconsistent and a little hard to learn. Additionally, they lag behind the string operations in other programming languages, so that some things that are easy to do in languages like Ruby or Python are rather hard to do in R. The **stringr** package aims to remedy these problems by providing a clean, modern interface to common string operations.

More concretely, **stringr**:

- Processes factors and characters in the same way.
- Gives functions consistent names and arguments.
- Simplifies string operations by eliminating options that you don't need 95% of the time (the other 5% of the time you can use the base functions).
- Produces outputs than can easily be used as inputs. This includes ensuring that missing inputs result in missing outputs, and zero length inputs result in zero length outputs.
- Completes R's string handling functions with useful functions from other programming languages.

To meet these goals, **stringr** provides two basic families of functions:

- basic string operations, and
- pattern matching functions which use regular expressions to detect, locate, match, replace, extract, and split strings.

These are described in more detail in the following sections.

Basic string operations

There are three string functions that are closely related to their base R equivalents, but with a few enhancements:

- `str_c` is equivalent to `paste`, but it uses the empty string ("") as the default separator and silently removes zero length arguments.
- `str_length` is equivalent to `nchar`, but it preserves NA's (rather than giving them length 2) and converts factors to characters (not integers).
- `str_sub` is equivalent to `substr` but it returns a zero length vector if any of its inputs are zero length, and otherwise expands each argument to match the longest. It also accepts negative positions, which are calculated from the left of the last character. The end position defaults to `-1`, which corresponds to the last character.
- `str_str<-` is equivalent to `substr<-`, but like `str_sub` it understands negative indices, and replacement strings not do need to be the same length as the string they are replacing.

Three functions add new functionality:

- `str_dup` to duplicate the characters within a string.
- `str_trim` to remove leading and trailing whitespace.
- `str_pad` to pad a string with extra whitespace on the left, right, or both sides.

Pattern matching

stringr provides pattern matching functions to **detect**, **locate**, **extract**, **match**, **replace**, and **split** strings:

- `str_detect` detects the presence or absence of a pattern and returns a logical vector. Based on `grepl`.
- `str_locate` locates the first position of a pattern and returns a numeric matrix with columns start and end. `str_locate_all` locates all matches, returning a list of numeric matrices. Based on `regexpr` and `gregexpr`.

- `str_extract` extracts text corresponding to the first match, returning a character vector. `str_extract_all` extracts all matches and returns a list of character vectors.
- `str_match` extracts capture groups formed by `()` from the first match. It returns a character matrix with one column for the complete match and one column for each group. `str_match_all` extracts capture groups from all matches and returns a list of character matrices.
- `str_replace` replaces the first matched pattern and returns a character vector. `str_replace_all` replaces all matches. Based on `sub` and `gsub`.
- `str_split_fixed` splits the string into a fixed number of pieces based on a pattern and returns a character matrix. `str_split` splits a string into a variable number of pieces and returns a list of character vectors.

Figure 1 shows how the simple (single match) form of each of these functions work.

Arguments

Each pattern matching function has the same first two arguments, a character vector of `string`s to process and a single `pattern` (regular expression) to match. The replace functions have an additional argument specifying the replacement string, and the split functions have an argument to specify the number of pieces.

Unlike base string functions, **stringr** only offers limited control over the type of matching. The `fixed()` and `ignore.case()` functions modify the pattern to use fixed matching or to ignore case, but if you want to use perl-style regular expressions or to match on bytes instead of characters, you're out of luck and you'll have to use the base string functions. This is a deliberate choice made to simplify these functions. For example, while `grepl` has six arguments, `str_detect` only has two.

Regular expressions

To be able to use these functions effectively, you'll need a good knowledge of regular expressions (Friedl, 1997), which this paper is not going to teach you. Some useful tools to get you started:

- A good reference sheet[1]
- A tool that allows you to interactively test[2] what a regular expression will match
- A tool to build a regular expression[3] from an input string

When writing regular expressions, I strongly recommend generating a list of positive (pattern should match) and negative (pattern shouldn't match) test cases to ensure that you are matching the correct components.

Functions that return lists

Many of the functions return a list of vectors or matrices. To work with each element of the list there are two strategies: iterate through a common set of indices, or use `mapply` to iterate through the vectors simultaneously. The first approach is usually easier to understand and is illustrated in Figure 2.

Conclusion

stringr provides an opinionated interface to strings in R. It makes string processing simpler by removing uncommon options, and by vigorously enforcing consistency across functions. I have also added new functions that I have found useful from Ruby, and over time, I hope users will suggest useful functions from other programming languages. I will continue to build on the included test suite to ensure that the package behaves as expected and remains bug free.

Bibliography

J. E. Friedl. *Mastering Regular Expressions*. O'Reilly, 1997. URL http://oreilly.com/catalog/9781565922570.

Hadley Wickham
Department of Statistics
Rice University
6100 Main St MS#138
Houston TX 77005-1827
USA
hadley@rice.edu

[1] http://www.regular-expressions.info/reference.html
[2] http://gskinner.com/RegExr/
[3] http://www.txt2re.com

```
library(stringr)
strings <- c(" 219 733 8965", "329-293-8753 ", "banana", "595 794 7569",
 "387 287 6718", "apple", "233.398.9187  ", "482 952 3315", "239 923 8115",
 "842 566 4692", "Work: 579-499-7527", "$1000", "Home: 543.355.3679")
phone <- "([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"

# Which strings contain phone numbers?
str_detect(strings, phone)
strings[str_detect(strings, phone)]

# Where in the string is the phone number located?
loc <- str_locate(strings, phone)
loc
# Extract just the phone numbers
str_sub(strings, loc[, "start"], loc[, "end"])
# Or more conveniently:
str_extract(strings, phone)

# Pull out the three components of the match
str_match(strings, phone)

# Anonymise the data
str_replace(strings, phone, "XXX-XXX-XXXX")
```

Figure 1: Simple string matching functions for processing a character vector containing phone numbers (among other things).

```
library(stringr)
col2hex <- function(col) {
  rgb <- col2rgb(col)
  rgb(rgb["red", ], rgb["green", ], rgb["blue", ], max = 255)
}

# Goal replace colour names in a string with their hex equivalent
strings <- c("Roses are red, violets are blue", "My favourite colour is green")

colours <- str_c("\\b", colors(), "\\b", collapse="|")
# This gets us the colours, but we have no way of replacing them
str_extract_all(strings, colours)

# Instead, let's work with locations
locs <- str_locate_all(strings, colours)
sapply(seq_along(strings), function(i) {
  string <- strings[i]
  loc <- locs[[i]]

  # Convert colours to hex and replace
  hex <- col2hex(str_sub(string, loc[, "start"], loc[, "end"]))
  str_sub(string, loc[, "start"], loc[, "end"]) <- hex
  string
})
```

Figure 2: A more complex situation involving iteration through a string and processing matches with a function.