Wrangling strings of characters is something we will find ourselves doing often as data scientists. 

**R** provides a comprehensive set of tools for handling and processing strings. In this chapter we review the functionality provided by **R** for managing and manipulating strings.

Through this guide new **R** commands will be introduced. The reader is encouraged to review the command’s documentation and understand what the command does. Help is obtained using the `?` command as in:

```
?read.csv
```

Documentation on a particular package can be obtained using the `help=` option of `library()`:

```
library(help=rattle)
```

This chapter is intended to be hands on. To learn effectively you are encouraged to run **R** (e.g., RStudio or Emacs with ESS mode) and to replicate the commands. Check that output is the same and that you understand how it is generated. Try some variations. Explore.
1 Packages Used

Packages used in this chapter include `dplyr` (Wickham et al., 2018), `glue` (Hester, 2017), `magrittr` (Bache and Wickham, 2014), `stringr` (Wickham, 2018), `stringi` (Gagolewski et al., 2018), `scales` (Wickham, 2017), and `rattle.data` (Williams, 2017b).

```r
# Load required packages from local library into R session.
library(dplyr)  # Wrangling: mutate().
library(stringi)  # The string concat operator %s+%.
library(stringr)  # String manipulation.
library(glue)  # Format strings.
library(magrittr)  # Pipelines for data processing: %>% %>% %<>%.
library(rattle.data)  # Weather dataset.
library(scales)  # commas(), percent().
```
## 2 Concatenate Strings

One of the most basic operations in string manipulation is the concatenate operation. R provides alternatives for doing so but a modern favourite is the `stringi::%s+%` operator.

```
"abc" %s+% "def" %s+% "ghi"
## [1] "abcdefghi"

c("abc", "def", "ghi", "jkl") %s+ c("mno")
## [1] "abcdefghijkmno"

c("abc", "def", "ghi", "jkl") %s+ c("mno", "pqr")
## [1] "abcdefghijkmnopqr"

c("abc", "def", "ghi", "jkl") %s+ c("mno", "pqr", "stu", "vwx")
## [1] "abcdefghijkmnopqrstu" "jklvwx"
```

The tidy function for concatenating strings is `stringr::str_c()`. A `sep` can be used to specify a separator for the concatenated strings.

```
str_c("hello", "world")
## [1] "helloworld"

str_c("hello", "world", sep= " ")
## [1] "hello world"
```

We can also concatenate strings using `glue::glue()`.

```
glue("hello", "world")
## helloworld
```

The traditional `base::cat()` function returns the concatenation of the supplied strings. Numeric and other complex objects are converted into character strings.

```
cat("hello", "world")
## hello world
cat ("hello", 123, "world")
## hello 123 world
```

Yet another alternative (and there are many) is the function `base::paste()`. Notice that it separates the concatenated strings with a space.

```
paste("hello", "world")
## [1] "hello world"
```
3 Concatenate Strings Special Cases

Each operator/functions treats NULL differently. Note the convenience for \texttt{base::cat()} to add a space between the strings, and that \texttt{base::paste()} treats NULL as a zero length string, and thus there are two spaces between the words concatenated.

```
"hello" %s+% NULL %s+% "world"
## character(0)
str_c("hello", NULL, "world")
## [1] "helloworld"
glue("hello", NULL, "world")

\texttt{cat("hello", NULL, "world")}
## hello world
paste("hello", NULL, "world")
## [1] "hello world"
```

NA tends to be treated differently too.

```
"hello" %s+% NA %s+% "world"
## [1] NA
str_c("hello", NA, "world")
## [1] NA
glue("hello", NA, "world")
## helloNAworld
\texttt{cat("hello", NA, "world")}
## hello NA world
paste("hello", NA, "world")
## [1] "hello NA world"
```

The examples becomes more interesting in the context that the arguments to the functions might be string returning functions. If that function returns NULL or NA, purposely or accidentally then it is useful to know the consequences.
4 String Length

The tidy way to get the length of a string is \texttt{stringr::str_length()}.

\begin{verbatim}
str_length("hello world")
## [1] 11
str_length(c("hello", "world"))
## [1] 5 5
str_length(NULL)
## integer(0)
str_length(NA)
## [1] NA
\end{verbatim}

The function \texttt{base::nchar()} is the traditional approach.

\begin{verbatim}
nchar("hello world")
## [1] 11
nchar(c("hello", "world"))
## [1] 5 5
nchar(NULL)
## integer(0)
nchar(NA)
## [1] NA
\end{verbatim}
5 Case Conversion

Often during data transformations strings have to be converted from one case to the other. These simple transformations can be achieved by `base::tolower()` and `base::toupper()`. The `base::casefold()` function can also be used as a wrapper to the two functions.

```
toupper("String Manipulation")
## [1] "STRING MANIPULATION"

tolower("String Manipulation")
## [1] "string manipulation"

casefold("String Manipulation")
## [1] "string manipulation"

casefold("String Manipulation", upper=TRUE)
## [1] "STRING MANIPULATION"
```
6 Tidy Sub-String Operations

We will find ourselves often wanting to extract or modify sub-strings within a string. The tidy way to do this is with the `stringr::str_sub()` function. We can specify the `start=` and the `end=` of the string. The indices of the string start from 1.

```r
s <- "string manipulation"
str_sub(s, start=3, end=6)
## [1] "ring"
str_sub(s, 3, 6)
## [1] "ring"
```

A negative is used to count from the end of the string.

```r
str_sub(s, 1, -8)
## [1] "string manip"
```

Replacing a sub-string with another string is straightforward using the assignment operator.

```r
str_sub(s, 1, -8) <- "stip"
s
## [1] "stipulation"
```

The function also operates over a vector of strings.

```r
v <- c("string", "manipulation", "always", "fascinating")
str_sub(v, -4, -1)
## [1] "ring" "tion" "ways" "ting"
str_sub(v, -4, -1) <- "RING"
v
## [1] "stRING" "manipulaRING" "alRING" "fascinaRING"
```
7 Base Sub-String Operations

The base function `base::substr()` can be used to extract and replace parts of a string similar to `stringr::str_sub()`. Note however that it does not handle negative values and that string replacement only replaces the same length as the replacement string, without changing the length of the original string.

```r
s <- "string manipulation"
substr(s, start=3, stop=6)
## [1] "ring"
substr(s, 3, 6)
## [1] "ring"
substr(s, 1, 12) <- "stip"
s
## [1] "stipng manipulation"
```

The `base::substring()` function performs similarly though uses `last=` rather than `stop=.`.

```r
s <- "string manipulation"
substring(s, first=3, last=6)
## [1] "ring"
x <- c("abcd", "aabcd", "abcc", "cabcd")
substring(x, 2, 4)
## [1] "bcd" "abc" "abc" "abc"
substring(x, 2, 4) <- "AB"
x
## [1] "aABd" "aABcb" "bABcc" "cABcd"
```
8 Trim and Pad

One of the major challenges of string parsing is removing and adding whitespaces and wrapping text.

Additional white space can be present on the left, right or both sides of the word. The stringr::\texttt{str_trim()} function offers an effective way to get rid of these whitespaces.

```r
ws <- c(" abc", " def ", " ghi ")
str_trim(ws)
## [1] "abc" "def" "ghi"
str_trim(ws, side="left")
## [1] "abc" "def" "ghi"
str_trim(ws, side="right")
## [1] " abc" "def" " ghi"
str_trim(ws, side="both")
## [1] "abc" "def" "ghi"
```

Conversely we can also pad a string with additional characters for up to a specified width using stringr::\texttt{str_pad()}. The default padding character is a space but we can override that.

```r
str_pad("abc", width=7)
## [1] " abc"
str_pad("abc", width=7, side="left")
## [1] "abc"
str_pad("abc", width=7, side="right")
## [1] "abc 
str_pad("abc", width=7, side="both", pad="#")
## [1] "##abc##"
```
9 Wrapping and Words

Formatting a text string into a neat paragraph of defined maximum width is another operation we often find ourselves wanting. The stringr::str_wrap() function will do this for us.

```r
st <- "All the Worlds a stage, All men are merely players"
cat(str_wrap(st, width=25))
## All the Worlds a stage,
## All men are merely
## players
```

Words of course form the basis for wrapping a sentence. We may wish to extract words from a sentence ourselves for further processing. Here we use stringr::word() to do so. We specify the positions of the word to be extracted from the sentence. The default separator value is space.

```r
st <- c("The quick brown fox", "jumps on the brown dog")
word(st, start=1, end=2)
## [1] "The quick" "jumps on"
word(st, start=1, end=-2)
## [1] "The quick brown" "jumps on the brown"
```
10 Glue Strings Together

The `glue` package provides a mechanism for building output strings from a collection of strings and variables. The basic use of `glue::glue()` will concatenate its string arguments with variable substitution identified using curly braces. In this example we use the `rattle.data::weatherAUS` and format large numbers using `scales::comma()`.

```r
dsname <- "weatherAUS"
nobs <- nrow(weatherAUS)
starts <- min(weatherAUS$Date)

## The weatherAUS dataset has just less than 145,461 observations, starting f...
```

We can manually wrap the sentence.

```r
## The weatherAUS dataset has just less than 145,461 observations starting from 1 November 2007.
```

Notice how the initial and last empty lines are handled “as expected”, and the line split is maintained.

Named arguments within the function call can be used to assign values to variables that only exist in the scope of the function call.

```r
dname = "weather",
nobs = nrow(weather),
starts = min(weather$Date))

## The weather dataset has just less than 367 observations starting from 1 November 2007.
```

We can also see the effect of indenting lines in this example, where the indentation is retained.
### 11 Pipeline Glue

We can use `glue::glue_data()` within pipes and operate over the rows of the data that is piped into the operator.

```r
weatherAUS %>%
  sample_n(6) %>%
  glue_data("Observation",
    " {rownames(.) %>% as.integer() %>% comma() %>% sprintf('%7s', .)}",
    " location {Location %>% sprintf('%-14s', .)}",
    " max temp {MaxTemp %>% sprintf('%5.1f', .)}")
```

```
## Observation 133,068 location Launceston max temp 23.8
## Observation 136,307 location AliceSprings max temp 23.2
## Observation 41,622 location Williamtown max temp 23.2
## Observation 120,795 location Perth max temp 24.2
## Observation 93,346 location Townsville max temp 29.6
## Observation 75,506 location Portland max temp 14.3
....
```

It can also be useful with the tidyverse workflow.

```r
weatherAUS %>%
  sample_n(6) %>%
  mutate(TempRange = glue("{MinTemp}-{MaxTemp}")) %>%
  glue_data("Observed temperature range at {Location} of {TempRange}")
```

```
## Observed temperature range at Woomera of 6.5-17.6
## Observed temperature range at NorahHead of 17.7-26.3
## Observed temperature range at Townsville of 19.8-26.7
## Observed temperature range at Nuriootpa of 9.8-33.1
## Observed temperature range at MelbourneAirport of 15.3-26.6
## Observed temperature range at Nuriootpa of 3.6-22.1
....
```
12 Pattern Matching with Regular Expressions

One of the most powerful string processing concepts is the concept of regular expressions. A regular expression is a sequence of characters that describe a pattern. The concept was formalized by American mathematician Stephen Cole Kleene. A regular expression pattern can contain a combination of alphanumeric and special characters. It is a complex topic and we take an introductory look at it here to craft regular expressions in R.

An important concept is that of metacharacters which have special meaning within a regular expression. Unlike other characters that are used to match themselves, metacharacters have a specific meaning. The following table shows a list of metacharacters used in regular expressions.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches at the start of the string</td>
</tr>
<tr>
<td>$</td>
<td>Matches at the end of the string</td>
</tr>
<tr>
<td>()</td>
<td>Define a subexpression to be matched and retrieved later.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>Matches a single character that is contained within bracket</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character</td>
</tr>
</tbody>
</table>

Such metacharacters are used to match different patterns.

```r
s <- c("hands", "data", "on", "data$ience", "hands$data$ience", "hand$son")
grep(pattern="^data", s, value=TRUE)
## [1] "data"  "data$ience"
grep(pattern="on$", s, value=TRUE)
## [1] "on"    "hand$son"
grep(pattern="(nd)..(nd)", s, value=TRUE)
## [1] "hand$son$data$ience"
```

In order to match a metacharacter in R we need to escape it with `\` (double backslash).

```r
grep(pattern="\\$", s, value=TRUE)
## [1] "data$ience"  "hands$data$ience"
```
13 Regular Expressions: Quantifiers

Quantifiers are used to match repetition of a pattern within a string. The following table shows a list of quantifiers.

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>The preceding item is matched 0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>The preceding item is matched 1 or more times</td>
</tr>
<tr>
<td>?</td>
<td>The preceding item is matched at most 1 times.</td>
</tr>
<tr>
<td>{n}</td>
<td>The preceding item is matched n times.</td>
</tr>
<tr>
<td>{n,}</td>
<td>The preceding item is matched at least n times.</td>
</tr>
</tbody>
</table>

Some examples will illustrate.

```r
s <- c("aaab", "abb", "bc", "abbcd", "bbbc", "abab", "caa")
grep(pattern="ab*b", s, value=TRUE)  # [1] "aaab" "abb" "abbcd" "abab"
grep(pattern="abbc?", s, value=TRUE)  # [1] "abb" "abbcd"
grep(pattern="b{2,}?", s, value=TRUE)  # [1] "abb" "abbcd" "bbbc"
```

Generated 29th July 2018 6:33pm
14 Regular Expressions: Character Classes

A character class is a collection of characters that are in some way grouped together. We enclose the characters to be grouped within square brackets \[\]. The pattern then matches any one of the characters in the set. For example, the character class \[0-9\] matches any of the digits from 0 to 9.

<table>
<thead>
<tr>
<th>Character Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [0-9]</td>
<td>Digits</td>
</tr>
<tr>
<td>2 [a-z]</td>
<td>Lower-case letters</td>
</tr>
<tr>
<td>3 [A-Z]</td>
<td>Upper-case letters</td>
</tr>
<tr>
<td>4 [a-zA-Z]</td>
<td>Alphabetic characters</td>
</tr>
<tr>
<td>5 [^a-zA-Z]</td>
<td>Non-alphabetic characters</td>
</tr>
<tr>
<td>6 [a-zA-Z0-9]</td>
<td>Alphanumeric characters</td>
</tr>
<tr>
<td>7 [[:space:]]</td>
<td>Space characters</td>
</tr>
<tr>
<td>8 [!,:;'()@#$]^_`~?&gt;&lt;=-*+.?</td>
<td>Punctuation characters</td>
</tr>
</tbody>
</table>

```
s <- c("abc12", "@#$", "345", "ABcd")
grep(pattern="[0-9]+", s, value=TRUE)
  # [1] "abc12" "345"

grep(pattern="[a-z]+", s, value=TRUE)
  # [1] "ABcd"

grep(pattern="[^@#$]+", s, value=TRUE)
  # [1] "abc12" "345" "ABcd"
```

R also supports the use of POSIX character classes which are represented within \[\[\]\] (double braces).

```
grep(pattern="[:alpha:]", s, value=TRUE)
  # [1] "abc12" "ABcd"

grep(pattern="[:upper:]", s, value=TRUE)
  # [1] "ABcd"
```
15 Generate Strings for Testing

It is sometimes very useful to be able to test out some code using some test data. A simple way to generate test strings us with `stringi::stri_rand_lipsum()`.

```r
stri_rand_lipsum(20)
## [1] "Lorem ipsum dolor sit amet, posuere at in in id ligula sodales eget ..."
## [2] "Sapien augue dignissim, vulputate, montes ipsum rutrum eu eu porta f...
## [3] "Ultrices ante in commodo eu id elementum velit ut bibendum. Nisl, la...
## [4] "Dis aptent senectus rhoncus et sed donec, vitae posuere, neque. Arcu...
## [5] "Vestibulum molestie in donec tincidunt, eu sapien. Quam in curae cla...
## [6] "Leo, ac integer sed penatibus. Curabitur, neque habitant quam, dui c...
## [7] "Vulputate elementum in urna. Ut nunc sed, imperdiet. Suscipit eu int...
## [8] "Senectus justo. Lobortis mauris praesent taciti. Massa ultrices in v...
## [9] "Nec at sapien phasellus nec eros quis ligula ac vestibulum, eu lorem...
## [10] "Sit, vestibulum est, velit ultrices nisi porta aliquam non in. Monte...
## [11] "Risus, sit metus augue non. Quisque sed amet, ac libero tempus sed n...
## [12] "Sed porttitor, eu amet ex amet nibh mauris, venenatis sed nec. Netus...
## [13] "Tellus himenaeos in commodo tincidunt sit. Metus sit mauris mus si...
## [14] "Massa in in potenti tellus rutrum orci donec fames. Ut elit in moles...
## [15] "In ac fermentum amat mus. In ridiculus augue elit fermentum, ornare...
## [16] "Sed ultricies vel consequat aliquet magnis nisl tortor. Nisl eu, gra...
## [17] "Scoleresisque sagittis consequat tempor iaculis sociis commodo. Hac ma...
## [18] "Non a interdum per malesuada enim potenti cum. Auctor ut purus egest...
## [19] "Interdum vel ut eros. Sed, dictumst laoreet curabitur nec, cursus ar...
## [20] "Leo sed ut at lacinia lacus enim felis in ultrices. Imperdiet feugia...

stri_rand_lipsum(2)
## [1] "Lorem ipsum dolor sit amet, sem, aliquam duis arcu. Nam magna, non ve...

sapply(stri_rand_lipsum(10), nchar, USE.NAMES=FALSE)
## [1] 514 527 617 740 653 630 690 789 417

sapply(stri_rand_lipsum(10), nchar, USE.NAMES=FALSE)
## [1] 502 473 615 360 813 761 629 547 290 301
```

The strings generated are of different lengths and each call generates different strings.
16 Read a File as Vector of Strings

There may be occasions where we would like to load a dataset from a file as strings, one line as a string, returning a vector of strings. We can achieve using the function `base::readLines()`.

In the following example we access the system file `weather.csv` that is provided by the `rattle` package.

```r
dsnname <- "weather"  # Dataset name.
ftype <- "csv"  # Source dataset file type.
dsnname %s+% "." %s+% ftype %T>%
print() %>%
system.file(ftype, ., package="rattle") %>%
readLines() ->
ds
## [1] "weather.csv"
```

A sample of the data.

```r
head(ds)
```

Find those strings that contain a specific pattern using `base::grep()`.

```r
grep("ENE", ds)
## [1] 3 10 23 26 28 36 37 42 43 49 50 54 68 69 71 76 86 91
## [37] 222 278 303 304 310 323 341 348 351 357 365
grep("ENE", ds, value=TRUE)
```
17 Command Summary

This chapter has introduced, demonstrated and described the following R packages, functions, commands, operators, and datasets:

%+%  Operator from stringi. Concatenate strings with no separator between the strings.
cat()  Function from base. Concatenate strings with a space separator between the strings by default. Add sep= to specify a different or no separator.
gregexpr()  Function from base. Returns all matches of pattern in string.
grep()  Function from base. Returns index of elements that matched.
grepl()  Function from base. Returns boolean values indicating if a pattern exist in the string.
gsub()  Function from base. Replaces all matches of pattern with replacement.
paste()  Function from base. Concatenate strings by pasting them together.
regexec()  Function from base. Combines results of regexpr() and gregexpr().
regexpr()  Function from base. Returns the first match of the pattern in string.
str_c()  Function from stringr. Tidy version of concatenate strings.
strsplit()  Function from data.table. Split string in to vector according to pattern match.
str_detect()  Function from stringr. Detect a presence or absence of a pattern in a string.
str_extract()  Function from stringr. Extracts first occurrence of pattern in string.
str_extract_all()  Function from stringr. Extracts all occurrences of pattern in string.
str_match()  Function from stringr. Extract first matched group from a string.
str_match_all()  Function from stringr. Extract all matched groups from a string.
str_locate()  Function from stringr. Locate the position of the first occurrence of a pattern in a string.
str_locate_all()  Function from stringr. Locate the position of all occurrences of a pattern in a string.
str_replace()  Function from stringr. Returns the first match of the pattern in string.
str_replace_all()  Function from stringr. Returns all matches of pattern in string.
str_split()  Function from stringr. Split up a string into a variable number of pieces.
str_split_fixed()  Function from stringr. Split up a string into a fixed number of pieces.
sub()  Function from base. Replaces the first match of pattern with replacement.
18 Further Reading and Acknowledgements

The Rattle book (Williams, 2011), published by Springer, provides a comprehensive introduction to data mining and analytics using Rattle and R. It is available from Amazon. Rattle provides a graphical user interface through which the user is able to load, explore, visualise, and transform data, and to build, evaluate, and export models. Through its Log tab it specifically aims to provide an R template which can be exported and serve as the starting point for further programming with data in R.

The Essentials of Data Science book (Williams, 2017a), published by CRC Press, provides a comprehensive introduction to data science through programming with data using R. It is available from Amazon. The book provides a template based approach to doing data science and knowledge discovery. Templates are provided for data wrangling and model building. These serve as generic starting points for programming with data, and are designed to require minimal effort to get started. Visit https://essentials.togaware.com for further guides and templates.

Other resources include:

- Handling and Processing Strings in R, a freely available ebook by Gaston Sanchez from 2013.


Some of the material has been updated from material collected by Karthik Bharadwaj.
19 References


This document, sourced from StringsO.Rnw bitbucket revision 276, was processed by KnitR version 1.20 of 2018-02-20 10:11:46 UTC and took 7.4 seconds to process. It was generated by gjw on Ubuntu 18.04 LTS.