Strings Basics

STAT 133

Gaston Sanchez

`github.com/ucb-stat133/stat133-fall-2016`

Character Vectors Reminder

Character Basics

We express character strings using single or double quotes:

```
# string with single quotes
'a character string using single quotes'
```

```
# string with double quotes
"a character string using double quotes"
```

Character Basics

We can insert single quotes in a string with double quotes, and vice versa:

```
# single quotes within double quotes
"The 'R' project for statistical computing"
```

```
# double quotes within single quotes
'The "R" project for statistical computing'
```

Character Basics

We cannot insert single quotes in a string with single quotes, neither we can insert double quotes in a string with double quotes (Don't do this!):

```
# don't do this!
"This "is" totally unacceptable"
```

```
# don't do this!
'This 'is' absolutely wrong'
```

Function `character()`

Besides the single quotes or double quotes, R provides the function `character()` to create vectors of type character.

```
# character vector of 5 elements
a <- character(5)

a

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

Empty string

The most basic string is the **empty string** produced by consecutive quotation marks: `""`.

```
# empty string
empty_str <- ""

empty_str

## [1] ""
```

Technically, `""` is a string with no characters in it, hence the name *empty string*.

Empty character vector

Another basic string structure is the **empty character vector** produced by `character(0)`:

```
# empty character vector
empty_chr <- character(0)

empty_chr

## character(0)
```

Empty character vector

Do not to confuse the empty character vector `character(0)` with the empty string `""`; they have different lengths:

```
# length of empty string
length(empty_str)

## [1] 1

# length of empty character vector
length(empty_chr)

## [1] 0
```

Character Vectors

You can use the concatenate function `c()` to create character vectors:

```
strings <- c('one', '2', 'III', 'four')
strings

## [1] "one"  "2"    "III"  "four"

example <- c('mon', 'tue', 'wed', 'thu', 'fri')
example

## [1] "mon" "tue" "wed" "thu" "fri"
```

Replicate elements

You can also use the function `rep()` to create character vectors of replicated elements:

```
rep("a", times = 5)
rep(c("a", "b", "c"), times = 2)
rep(c("a", "b", "c"), times = c(3, 2, 1))
rep(c("a", "b", "c"), each = 2)
rep(c("a", "b", "c"), length.out = 5)
rep(c("a", "b", "c"), each = 2, times = 2)
```

Function `paste()`

The function `paste()` is perhaps one of the most important functions that we can use to create and build strings.

```
paste(..., sep = " ", collapse = NULL)
```

`paste()` takes one or more R objects, converts them to `"character"`, and then it concatenates (pastes) them to form one or several character strings.

Function paste()

Simple example using paste():

```
# paste
PI <- paste("The life of", pi)

PI

## [1] "The life of 3.14159265358979"
```

Function paste()

The default separator is a blank space (`sep = " "`). But you can select another character, for example `sep = "-"`:

```
# paste
tobe <- paste("to", "be", "or", "not", "to", "be", sep = "-")

tobe

## [1] "to-be-or-not-to-be"
```

Function paste()

If we give paste() objects of different length, then the recycling rule is applied:

```
# paste with objects of different lengths
paste("X", 1:5, sep = ".")

## [1] "X.1" "X.2" "X.3" "X.4" "X.5"
```

Function paste()

To see the effect of the collapse argument, let's compare the difference with collapsing and without it:

```
# paste with collapsing
paste(1:3, c("!", "?", "+"), sep = '', collapse = "")

## [1] "1!2?3+"

# paste without collapsing
paste(1:3, c("!", "?", "+"), sep = '')

## [1] "1!" "2?" "3+"
```

Printing Strings

Printing Methods

Functions for printing strings can be very useful when creating our own functions. They help us have more control on the way the output gets printed either on screen or in a file.

Example `str()`

Many functions print output to the console. Some examples are `summary()` and `str()`:

```
# str
str(mtcars, vec.len = 1)


## 'data.frame': 32 obs. of  11 variables:
##  $ mpg : num  21 21 ...
##  $ cyl : num  6 6 ...
##  $ disp: num  160 160 ...
##  $ hp  : num  110 110 ...
##  $ drat: num  3.9 3.9 ...
##  $ wt  : num  2.62 ...
##  $ qsec: num  16.5 ...
##  $ vs  : num  0 0 ...
##  $ am  : num  1 1 ...
##  $ gear: num  4 4 ...
##  $ carb: num  4 4 ...
```

Printing Characters

R provides a series of functions for printing strings.

Printing functions

Function	Description
`print()`	generic printing
`noquote()`	print with no quotes
`cat()`	concatenation
`format()`	special formats
`toString()`	convert to string
`sprintf()`	C-style printing

Method `print()`

The *workhorse* printing function in R is `print()`, which prints its argument on the console:

```
# text string
my_string <- "programming with data is fun"

# print string
print(my_string)

## [1] "programming with data is fun"
```

To be more precise, `print()` is a generic function, which means that you should use this function when creating printing methods for programmed classes.

Method `print()`

If we want to print character strings with no quotes we can set the argument `quote = FALSE`

```
# print without quotes
print(my_string, quote = FALSE)

## [1] programming with data is fun
```

Function noquote()

An alternative option for achieving a similar output is by using noquote()

```
# print without quotes
noquote(my_string)

## [1] programming with data is fun

# similar to:
print(my_string, quote = FALSE)

## [1] programming with data is fun
```

Function `cat()`

Another very useful function is `cat()` which allows us to concatenate objects and print them either on screen or to a file. Its usage has the following structure:

```
cat(..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
```

Function cat()

If we use cat() with only one single string, you get a similar (although not identical) result as noquote():

```
# simply print with 'cat()'
cat(my_string)

## programming with data is fun
```

cat() prints its arguments without quotes. In essence, cat() simply displays its content (on screen or in a file).

Function cat()

When we pass vectors to cat(), each of the elements are treated as though they were separate arguments:

```
# first four months
cat(month.name[1:4], sep = " ")

## January February March April
```

Function cat()

The argument `fill` allows us to break long strings; this is achieved when we specify the string width with an integer number:

```
# fill = 30
cat("Looooooooooong strings", "can be displayed",
    "in a nice format",
    "by using the 'fill' argument", fill = 30)

## Looooooooooong strings
## can be displayed
## in a nice format
## by using the 'fill' argument
```

Function `cat()`

Last but not least, we can specify a file output in `cat()`. For instance, to save the output in the file `output.txt` located in your working directory:

```
# cat with output in a given file
cat(my_string, "with R", file = "output.txt")
```

Function `format()`

The function `format()` allows us to format an R object for pretty printing. This is especially useful when printing numbers and quantities under different formats.

```
# default usage
format(13.7)

## [1] "13.7"

# another example
format(13.12345678)

## [1] "13.12346"
```

Function `format()`

Some useful arguments of `format()`:

- `width` the (minimum) width of strings produced
- `trim` if set to `TRUE` there is no padding with spaces
- `justify` controls how padding takes place for strings. Takes the values `"left"`, `"right"`, `"centre"`, `"none"`

For controling the printing of numbers, use these arguments:

- `digits` The number of digits to the right of the decimal place.
- `scientific` use `TRUE` for scientific notation, `FALSE` for standard notation

Function format()

```r
# justify options
format(c("A", "BB", "CCC"), width = 5, justify = "centre")
```

```
## [1] "  A  " " BB  " " CCC "
```

```r
format(c("A", "BB", "CCC"), width = 5, justify = "left")
```

```
## [1] "A    " "BB   " "CCC  "
```

```r
format(c("A", "BB", "CCC"), width = 5, justify = "right")
```

```
## [1] "    A" "   BB" "  CCC"
```

```r
format(c("A", "BB", "CCC"), width = 5, justify = "none")
```

```
## [1] "A"   "BB"  "CCC"
```

Function `format()`

```
# digits
format(1/1:5, digits = 2)

## [1] "1.00" "0.50" "0.33" "0.25" "0.20"

# use of 'digits', widths and justify
format(format(1/1:5, digits = 2), width = 6, justify = "c")

## [1] " 1.00 " " 0.50 " " 0.33 " " 0.25 " " 0.20 "
```

string formatting with `sprintf()`

The function `sprintf()` is a wrapper for the `C` function `sprintf()` that returns a formatted string combining text and variable values. Its usage has the following form:

```
sprintf(fmt, ...)
```

The nice feature about `sprintf()` is that it provides us a very flexible way of formatting vector elements as character strings.

Using `sprintf()`

Several ways in which the number `pi` can be formatted:

```
# "%f" indicates 'fixed point' decimal notation
sprintf("%f", pi)

## [1] "3.141593"

# decimal notation with 3 decimal digits
sprintf("%.3f", pi)

## [1] "3.142"

# 1 integer and 0 decimal digits
sprintf("%1.0f", pi)

## [1] "3"
```

Using `sprintf()`

Several ways in which the number `pi` can be formatted:

```
# more options
sprintf("%5.1f", pi)

## [1] "  3.1"

sprintf("%05.1f", pi)

## [1] "003.1"
```

Using sprintf()

```
# print with sign (positive)
sprintf("%+f", pi)

## [1] "+3.141593"

# prefix a space
sprintf("% f", pi)

## [1] " 3.141593"

# left adjustment
sprintf("%-10f", pi) # left justified

## [1] "3.141593  "
```

Using sprintf()

```
# exponential decimal notation "e"
sprintf("%e", pi)

## [1] "3.141593e+00"

# exponential decimal notation "E"
sprintf("%E", pi)

## [1] "3.141593E+00"

# number of significant digits (6 by default)
sprintf("%g", pi)

## [1] "3.14159"
```

Using sprintf()

```
# more sprintf examples
sprintf("Harry's age is %s", 12)

## [1] "Harry's age is 12"

sprintf("five is %s, six is %s", 5, 6)

## [1] "five is 5, six is 6"
```

Comparing printing methods

```r
# printing method
print(1:5)
# convert to character
as.character(1:5)
# concatenation
cat(1:5, sep="-")
# default pasting
paste(1:5)
# paste with collapsing
paste(1:5, collapse = "")
# convert to a single string
toString(1:5)
# unquoted output
noquote(as.character(1:5))
```

ggplot2 object

```
library(ggplot2)

gg <- ggplot(data = mtcars, aes(x = mpg, y = hp)) +
  geom_point()

summary(gg)


## data: mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb [32x11]
## mapping:  x = mpg, y = hp
## faceting: facet_null()
## -----------------------------------
## geom_point: na.rm = FALSE
## stat_identity: na.rm = FALSE
## position_identity
```

ggplot2 summary()

https://github.com/hadley/ggplot2/blob/master/R/summary.r

```
summary.ggplot <- function(object, ...) {
  wrap <- function(x) paste(
    paste(strwrap(x, exdent = 2), collapse = "\n"), "\n", sep = "")

  if (!is.null(object$data)) {
    output <- paste(
      "data:      ", paste(names(object$data), collapse = ", "),
      " [", nrow(object$data), "x", ncol(object$data), "] ",
      "\n", sep = "")
    cat(wrap(output))
  }
  if (length(object$mapping) > 0) {
    cat("mapping:  ", clist(object$mapping), "\n", sep = "")
  }
  if (object$scales$n() > 0) {
    cat("scales:  ", paste(object$scales$input(), collapse = ", "), "\n")
  }
  cat("faceting: ")
  print(object$facet)
  if (length(object$layers) > 0)
    cat("-----------------------------------\n")
  invisible(lapply(object$layers, function(x) {
    print(x)
    cat("\n")
  }))

}
```

Reading Raw Text

Reading Text with `readlines()`

- `readLines()` allows us to import text *as is* (i.e. we want to read raw text)
- Use `readLines()` if you don't want R to assume that the data is any particular form
- `readLines()` takes the name of a file or the name of a URL that we want to read
- The output is a character vector with one element for each line of the file or url

Reading Text with `readlines()`

For instance, here's how to read the file located at:
http://www.textfiles.com/music/ktop100.txt

```
# read 'ktop100.txt' file
ktop <- "http://www.textfiles.com/music/ktop100.txt"

top105 <- readLines(ktop)
```

Reading Text with `readlines()`

```
head(top105, n = 5)

## [1] "From: ed@wente.llnl.gov (Ed Suranyi)"
## [2] "Date: 12 Jan 92 21:23:55 GMT"
## [3] "Newsgroups: rec.music.misc"
## [4] "Subject: KITS' year end countdown"
## [5] ""
```

Basic String Manipulation

String Manipulation

There are a number of very handy functions in R for doing some basic manipulation of strings:

Manipulation of strings

Function	Description
`nchar()`	number of characters
`tolower()`	convert to lower case
`toupper()`	convert to upper case
`casefold()`	case folding
`chartr()`	character translation
`abbreviate()`	abbreviation
`substring()`	substrings of a character vector
`substr()`	substrings of a character vector

Counting number of characters

`nchar()` counts the number of characters in a string, that is, the "length" of a string:

```
# how many characters?
nchar(c("How", "many", "characters?"))

## [1]  3  4 11

# how many characters?
nchar("How many characters?")

## [1] 20
```

Notice that the white spaces between words in the second example are also counted as characters.

Counting number of characters

Do not confuse `nchar()` with `length()`. The former gives us the **number of characters**, the later only gives the **number of elements** in a vector:

```
# how many elements?
length(c("How", "many", "characters?"))

## [1] 3

# how many elements?
length("How many characters?")

## [1] 1
```

Convert to lower case with `tolower()`

R comes with three functions for text casefolding. The first function we'll discuss is `tolower()` which converts any upper case characters into lower case:

```
# to lower case
tolower(c("aLL ChaRacterS in LoweR caSe", "ABCDE"))

## [1] "all characters in lower case" "abcde"
```

Convert to upper case with `toupper()`

The opposite function of `tolower()` is `toupper`. As you may guess, this function converts any lower case characters into upper case:

```
# to upper case
toupper(c("All ChaRacterS in Upper Case", "abcde"))

## [1] "ALL CHARACTERS IN UPPER CASE" "ABCDE"
```

Case conversion with `casefold()`

`casefold()` converts all characters to lower case, but we can use the argument `upper = TRUE` to indicate the opposite (characters in upper case):

```
# lower case folding
casefold("aLL ChaRacterS in LoweR caSe")

## [1] "all characters in lower case"

# upper case folding
casefold("All ChaRacterS in Upper Case", upper = TRUE)

## [1] "ALL CHARACTERS IN UPPER CASE"
```

Character translation with `chartr()`

There's also the function `chartr()` which stands for *character translation*.

```
# character translation
chartr(old, new, x)
```

`chartr()` takes three arguments: an `old` string, a `new` string, and a character vector `x`

Character translation with `chartr()`

The way `chartr()` works is by replacing the characters in `old` that appear in `x` by those indicated in `new`. For example, suppose we want to translate the letter `"a"` (lower case) with `"A"` (upper case) in the sentence `x`:

```
# replace 'a' by 'A'
chartr("a", "A", "This is a boring string")

## [1] "This is A boring string"
```

Character translation with `chartr()`

```
# multiple replacements
crazy <- c("Here's to the crazy ones",
           "The misfits",
           "The rebels")

chartr("aei", "#!?", crazy)

## [1] "H!r!'s to th! cr#zy on!s" "Th! m?sf?ts"
## [3] "Th! r!b!ls"
```

Abbreviate strings with `abbreviate()`

Another useful function for basic manipulation of character strings is `abbreviate()`. Its usage has the following structure:

```
abbreviate(names.org, minlength = 4, dot = FALSE,
           strict =FALSE,
           method = c("left.keep", "both.sides"))
```

Although there are several arguments, the main parameter is the character vector (`names.org`) which will contain the names that we want to abbreviate

Abbreviate strings with abbreviate()

```
# some color names
some_colors <- colors()[1:4]

# abbreviate (default usage)
colors1 <- abbreviate(some_colors)
colors1

##         white     aliceblue  antiquewhite antiquewhite1
##        "whit"        "alcb"        "antq"        "ant1"
```

Abbreviate strings with abbreviate()

```
# abbreviate with 'minlength'
colors2 <- abbreviate(some_colors, minlength = 5)
colors2

##         white     aliceblue  antiquewhite antiquewhite1
##       "white"       "alcbl"       "antqw"       "antq1"

# abbreviate
colors3 <- abbreviate(some_colors, minlength = 3,
                      method = "both.sides")
colors3

##         white     aliceblue  antiquewhite antiquewhite1
##         "wht"         "alc"         "ant"         "an1"
```

Replace substrings with `substr()`

The function `substr()` extracts or replaces substrings in a character vector. Its usage has the following form:

```
# replace
substr(x, start, stop)
```

`x` is a character vector, `start` indicates the first element to be extracted (or replaced), and `stop` indicates the last element to be extracted (or replaced)

Extracting substrings with `substr()`

```
# extract characters in positions 1, 2, 3
substr("abcdef", 1, 3)

## [1] "abc"

# extract 'area code'
substr("(510) 987 6543", 2, 4)

## [1] "510"
```

Replace substrings with `substr()`

```
# replace 2nd letter with hash symbol
x <- c("may", "the", "force", "be", "with", "you")

substr(x, 2, 2) <- "#"

x

## [1] "m#y"   "t#e"   "f#rce" "b#"    "w#th"  "y#u"
```

Replace substrings with `substr()`

```
# replace 2nd and 3rd letters with ":)"
y <- c("may", "the", "force", "be", "with", "you")

substr(y, 2, 3) <- ":)"

y

## [1] "m:)"   "t:)"   "f:)ce" "b:"    "w:)h"  "y:)"
```

Replace substrings with `substr()`

```
# replacement with recycling
z <- c("may", "the", "force", "be", "with", "you")

substr(z, 2, 3) <- c("#", "@")

z

## [1] "m#y"   "t@e"   "f#rce" "b@"    "w#th"  "y@u"
```

Replace substrings with `substring()`

Closely related to `substr()`, the function `substring()` extracts or replaces substrings in a character vector. Its usage has the following form:

```
substring(text, first, last = 1000000L)
```

`text` is a character vector, `first` indicates the first element to be replaced, and `last` indicates the last element to be replaced

Replace substrings with `substring()`

```
# same as 'substr'
substring("ABCDEF", 2, 4)

## [1] "BCD"

substr("ABCDEF", 2, 4)

## [1] "BCD"

# extract each letter
substring("ABCDEF", 1:6, 1:6)

## [1] "A" "B" "C" "D" "E" "F"
```

Replace substrings with `substring()`

```
# multiple replacement with recycling
txt <- c("another", "dummy", "example")

substring(txt, 1:3) <- c(" ", "zzz")

txt
```

```
## [1] " nother" "dzzzy"   "ex mple"
```

Set Operations

We can apply functions such as set union, intersection, difference, equality and membership, on `"character"` vectors.

Function	Description
`union()`	set union
`intersect()`	intersection
`setdiff()`	set difference
`setequal()`	equal sets
`identical()`	exact equality
`is.element()`	is element
`%in%()`	contains
`sort()`	sorting

Union

```r
# two character vectors
set1 <- c("some", "random", "words", "some")

set2 <- c("some", "many", "none", "few")

# union of set1 and set2
union(set1, set2)

## [1] "some"   "random" "words"  "many"   "none"   "few"
```

Intersection

```
# two character vectors
set3 <- c("some", "random", "few", "words")

set4 <- c("some", "many", "none", "few")

# intersect of set3 and set4
intersect(set3, set4)

## [1] "some" "few"
```

Set Difference

```
# two character vectors
set5 <- c("some", "random", "few", "words")

set6 <- c("some", "many", "none", "few")

# difference between set5 and set6
setdiff(set5, set6)

## [1] "random" "words"
```

Set Equality

```
# three character vectors
set7 <- c("some", "random", "strings")
set8 <- c("some", "many", "none", "few")
set9 <- c("strings", "random", "some")

# set7 == set8?
setequal(set7, set8)

## [1] FALSE

# set7 == set9?
setequal(set7, set9)

## [1] TRUE
```

Element Membership

```
# three vectors
set10 <- c("some", "stuff", "to", "play", "with")
elem1 <- "play"
elem2 <- "many"

# elem1 in set10?
is.element(elem1, set10)

## [1] TRUE

# elem2 in set10?
is.element(elem2, set10)

## [1] FALSE
```

Element Membership

```
# elem1 in set10?
elem1 %in% set10

## [1] TRUE

# elem2 in set10?
elem2 %in% set10

## [1] FALSE
```

Sorting

`sort()` arranges elements in alphabetical order

```
set11 <- c("random", "words", "multiple")

# sort (decreasingly)
sort(set11)

## [1] "multiple" "random"   "words"

# sort (increasingly)
sort(set11, decreasing = TRUE)

## [1] "words"    "random"   "multiple"
```

Package "stringr"

About "stringr"

- functions are more consistent, simpler and easier to use
- "stringr" ensures that function and argument names (and positions) are consistent
- all functions deal with NA's and zero length character appropriately
- the output data structures from each function matches the input data structures of other functions

About "stringr"

"stringr" provides functions for both:

- basic manipulations and,
- for regular expression operations.

In this set of slides we cover those functions that have to do with basic manipulations.

About "stringr"

```
# installing 'stringr'
install.packages("stringr")

# load 'stringr'
library(stringr)
```

Basic "stringr" functions

Function	Description	Similar to
`str_c()`	string concatenation	`paste()`
`str_length()`	number of characters	`nchar()`
`str_sub()`	extracts substrings	`substring()`
`str_dup()`	duplicates characters	*none*
`str_trim()`	removes leading and trailing whitespace	*none*
`str_pad()`	pads a string	*none*
`str_wrap()`	wraps a string paragraph	`strwrap()`
`str_trim()`	trims a string	*none*

About "stringr"

`stringr` provides functions for both:

- all functions in "stringr" start with `str_`
- some functions are designed to provide a better alternative to already existing functions
- Other functions don't have a corresponding alternative

Function `str_c()`

`str_c()` is equivalent to `paste()` but instead of using the white space as the default separator, `str_c()` uses the empty string `""`

```
# default usage
str_c("May", "The", "Force", "Be", "With", "You")


## [1] "MayTheForceBeWithYou"
```

Function `str_c()`

Another major difference between `str_c()` and `paste()`: zero length arguments like `NULL` and `character(0)` are silently removed by `str_c()`.

```
# removing zero length objects
str_c("May", "The", "Force", NULL, "Be", "With", "You",
      character(0))

## [1] "MayTheForceBeWithYou"
```

Function str_c()

str_c() is equivalent to paste() but instead of using the white space as the default separator, str_c() uses the empty string ""

```
# changing separator
str_c("May", "The", "Force", "Be", "With", "You", sep="_")

## [1] "May_The_Force_Be_With_You"

# synonym function 'str_join'
str_join("May", "The", "Force", "Be", "With", "You", sep="-")

## Warning:  'str_join' is deprecated.
## Use 'str_c' instead.
## See help("Deprecated")

## [1] "May-The-Force-Be-With-You"
```

Function str_length()

str_length() is equivalent to nchar(), returning the number of characters in a string

```
# some text (NA included)
some_text = c("one", "two", "three", NA, "five")

# compare 'str_length' with 'nchar'
nchar(some_text)

## [1]  3  3  5 NA  4

str_length(some_text)

## [1]  3  3  5 NA  4
```

Function str_length()

str_length() has the nice feature that it converts factors to characters, something that nchar() is not able to handle:

```
# some factor
some_factor = factor(c(1, 1, 1, 2, 2, 2),
                     labels = c("good", "bad"))
some_factor

## [1] good good good bad  bad  bad
## Levels: good bad

# 'str_length' on a factor:
str_length(some_factor)

## [1] 4 4 4 3 3 3
```

Function str_length()

Compare str_length() against nchar()

```
# some factor
some_factor = factor(c(1,1,1,2,2,2),
                     labels = c("good", "bad"))

# now try 'nchar' on a factor
nchar(some_factor)

## Error in nchar(some_factor):  'nchar()' requires a
character vector
```

Function str_substr()

```
# some text
lorem = "Lorem Ipsum"

# apply 'str_sub'
str_sub(lorem, start=1, end=5)

## [1] "Lorem"

# equivalent to 'substring'
substring(lorem, first=1, last=5)

## [1] "Lorem"
```

Function str_substr()

str_sub() allows you to work with negative indices in the start and end positions:

```
# some strings
resto = c("brasserie", "bistrot", "creperie", "bouchon")

# 'str_sub' with negative positions
str_sub(resto, start=-4, end=-1)

## [1] "erie" "trot" "erie" "chon"
```

When we use a negative position, str_sub() counts backwards from last character.

Function `str_sub()`

A related function is `str_sub()`; when given a set of positions they will be recycled over the string

```
# extracting sequentially
str_sub(lorem, seq_len(nchar(lorem)))

##  [1] "Lorem Ipsum" "orem Ipsum"  "rem Ipsum"   "em Ipsum"    "
##  [6] " Ipsum"      "Ipsum"       "psum"        "sum"         "
## [11] "m"
```

Function str_sub()

We can also give str_sub() a negative sequence, something that substring() ignores:

```
# reverse substrings with negative positions
str_sub(lorem, -seq_len(nchar(lorem)))

##  [1] "m"           "um"          "sum"         "psum"          "
##  [6] " Ipsum"      "m Ipsum"     "em Ipsum"    "rem Ipsum"     "
## [11] "Lorem Ipsum"
```

Function `str_sub()`

We can use `str_sub()` not only for extracting subtrings but also for replacing substrings:

```
# replacing 'Lorem' with 'Nullam'
lorem <- "Lorem Ipsum"
str_sub(lorem, 1, 5) <- "Nullam"
lorem

## [1] "Nullam Ipsum"
```

Function str_sub()

```
# replacing with negative positions
lorem = "Lorem Ipsum"
str_sub(lorem, -1) <- "Nullam"
lorem

## [1] "Lorem IpsuNullam"

# multiple replacements
lorem = "Lorem Ipsum"
str_sub(lorem, c(1,7), c(5,8)) <- c("Nullam", "Enim")
lorem

## [1] "Nullam Ipsum"  "Lorem Enimsum"
```

Duplication with `str_dup()`

`str_dup()` duplicates and concatenates strings within a character vector:

```
# default usage
str_dup("hola", 3)

## [1] "holaholahola"

# use with differetn 'times'
str_dup("adios", 1:3)

## [1] "adios"             "adiosadios"       "adiosadiosadios"
```

Duplication with `str_dup()`

```
# use with a string vector
words <- c("lorem", "ipsum", "dolor")
str_dup(words, 2)

## [1] "loremlorem" "ipsumipsum" "dolordolor"

str_dup(words, 1:3)

## [1] "lorem"           "ipsumipsum"      "dolordolordolor"
```

Padding with `str_pad()`

Another handy function that we can find in `stringr` is `str_pad()` for *padding* a string. Its default usage has the following form:

```
str_pad(string, width, side = "left", pad = " ")
```

The idea of `str_pad()` is to take a string and pad it with leading or trailing characters to a specified total `width`.

Padding with `str_pad()`

```
# default usage
str_pad("hola", width=7)

## [1] "   hola"

# pad both sides
str_pad("adios", width=7, side="both")

## [1] " adios "
```

Padding with str_pad()

```
# left padding with '#'
str_pad("hashtag", width=8, pad="#")

## [1] "#hashtag"

# pad both sides with '-'
str_pad("hashtag", width=9, side="both", pad="-")

## [1] "-hashtag-"
```

Wrapping with `str_wrap()`

The function `str_wrap()` is equivalent to `strwrap()` which can be used to *wrap* a string to format paragraphs. Its default usage has the following form:

```
str_wrap(string, width = 80, indent = 0, exdent = 0)
```

Padding with `str_wrap()`

```
# quote (by Douglas Adams)
some_quote <- c(
  "I may not have gone",
  "where I intended to go,",
  "but I think I have ended up",
  "where I needed to be")

# some_quote in a single paragraph
some_quote <- paste(some_quote, collapse = " ")
```

Padding with `str_wrap()`

Say we want to display the text of `some_quote` within some pre-specified column width (e.g. width of 30):

```
# display paragraph with width=30
cat(str_wrap(some_quote, width = 30))

## I may not have gone where I
## intended to go, but I think I
## have ended up where I needed
## to be
```

Trimming with `str_trim()`

One of the typical tasks of string processing is that of parsing a text into individual words.
Usually, we end up with words that have blank spaces, called *whitespaces*, on either end of the word. In this situation, we can use the `str_trim()` function to remove any number of whitespaces at the ends of a string. Its usage requires only two arguments:

```
str_trim(string, side = "both")
```

Padding with str_trim()

```
# text with whitespaces
bad_text <- c(" several   ", "   whitespaces ")

# remove whitespaces on the left side
str_trim(bad_text, side = "left")

## [1] "several   "   "whitespaces "

# remove whitespaces on the right side
str_trim(bad_text, side = "right")

## [1] " several"        "   whitespaces"

# remove whitespaces on both sides
str_trim(bad_text, side = "both")

## [1] "several"     "whitespaces"
```

Word extraction with `word()`

`word()` function that is designed to extract words from a sentence:

```
word(string, start = 1L, end = start, sep = fixed(" ")
```

The way in which we use `word()` is by passing it a `string`, together with a `start` position of the first word to extract, and an `end` position of the last word to extract. By default, the separator `sep` used between words is a single space.

Word extraction with `word()`

```
# some sentence
change = c("Be the change", "you want to be")

# extract first word
word(change, 1)

## [1] "Be"   "you"

# extract second word
word(change, 2)

## [1] "the"   "want"
```

Word extraction with `word()`

```
# some sentence
change = c("Be the change", "you want to be")

# extract last word
word(change, -1)

## [1] "change" "be"

# extract all but the first words
word(change, 2, -1)

## [1] "the change" "want to be"
```