Character Vectors and Factors

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat/stat133
Course web: gastonsanchez.com/teaching/stat133

Character Vectors

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:

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

```r
# 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)
```

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
```

More on character

Once an empty character object has been created, new components may be added to it simply by giving it an index value outside its previous range:

```
# another example
example <- character(0)
example

## character(0)

# add first element
example[1] <- "first"
example

## [1] "first"
```

Empty character vector

We can add more elements without the need to follow a consecutive index range:

```
example[4] <- "fourth"
example

## [1] "first"  NA        NA        "fourth"

length(example)

## [1] 4
```

R fills the missing indices with missing values `NA`.

Function `is.character()`

To test if an object is of type `"character"` you use the function `is.character()`:

```
# define two objects 'a' and 'b'
a <- "test me"
b <- 8 + 9

# are 'a' and 'b' characters?
is.character(a)

## [1] TRUE

is.character(b)

## [1] FALSE
```

Function as.character()

R allows you to convert non-character objects into character strings with the function `as.character()`:

```
b
```

```
## [1] 17
```

```
# converting 'b' into character
as.character(b)
```

```
## [1] "17"
```

Replicate elements

You can 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()`

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:

```r
# 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+"
```

Function paste0()

There's also the function `paste0()` which is the equivalent of

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

```
# collapsing with paste0
paste0("let's", "collapse", "all", "these", "words")

## [1] "let'scollapseallthesewords"
```

More coming soon

We'll talk more about handling character strings in a couple of weeks

Factors

Factors

- A similar structure to vectors are **factors**
- factors are used for handling categorial (i.e. qualitative) variables
- they are represented as objects of class `"factor"`
- internally, factors are stored as integers
- factors behave much like vectors (but they are not vectors)

Types of Categorical (qualitative) variables

Categorical Variables and Factors

Types of Categorical (qualitative) variables

- Binary (2 categories)
- Nominal (there's no order of categories)
- Ordinal (categories have an order)

Factors

To create a factor we use the function `factor()`

```
#
cols <- c("blue", "red", "blue", "gray", "red")
cols <- factor(cols)
cols

## [1] blue red  blue gray red
## Levels: blue gray red
```

The different values in a factor are called **levels**

Binary Factors

Since factors represent categorical variables, we can have binary, nominal and ordinal factors

```
# binary factors have two levels
yes_no <- factor(c("yes", "yes", "no", "yes", "no"))
yes_no

## [1] yes yes no  yes no
## Levels: no yes
```

Nominal Factors

Nominal factors have unordered categories

```
# nominal factor
food <- factor(c("burger", "pizza", "burrito",
                 "pizza", "burrito", "pizza"))
food

## [1] burger  pizza   burrito pizza   burrito pizza
## Levels: burger burrito pizza
```

Ordinal Factors

Ordinal factors have ordered categories or levels; to create an ordered factor we need to specify the levels in the desired order

```
# ordinal factor
sizes <- factor(c("md", "sm", "md", "lg", "sm", "lg"),
                levels = c("sm", "md", "lg"),
                ordered = TRUE)
sizes

## [1] md sm md lg sm lg
## Levels: sm < md < lg
```

Note that the levels are ordered

Ordinal Factors

When creating ordinal factors, always specify the desired order of the `levels`, otherwise R will arrange them in alphanumeric order

```
# ordinal factor
bad_sizes <- factor(c("md", "sm", "md", "lg", "sm", "lg"),
                    ordered = TRUE)

bad_sizes

## [1] md sm md lg sm lg
## Levels: lg < md < sm
```

Note that the levels are arranged in alphanumeric order (not really what we want)

About Factors

We can use various functions to get information about a factor:

```
length(sizes)

## [1] 6

nlevels(sizes)

## [1] 3

levels(sizes)

## [1] "sm" "md" "lg"

is.ordered(sizes)

## [1] TRUE
```

Function `levels()`

- besides the argument `levels` of `factor()`, there is also the function `levels()`
- `levels()` lets you have access to the categories
- you can use `levels()` to **get** the categories
- you can use `levels()` to **set** the categorie

Function levels()

```
# size levels
levels(sizes)

## [1] "sm" "md" "lg"

# setting new levels
levels(sizes) <- c("Small", "Medium", "Large")
sizes

## [1] Medium Small  Medium Large  Small  Large
## Levels: Small < Medium < Large
```

Function `nlevels()`

`nlevels()` returns the number of levels of a factor. In other words, `nlevels()` returns the length of the attribute `levels`:

```r
# nlevels()
nlevels(food)

## [1] 3

# equivalent to
length(levels(food))

## [1] 3
```

Merging levels

- Sometimes we may need to "merge" or collapse two or more different levels into one single level
- We can achieve this by using the function `levels()`
- Assign a new vector of levels containing repeated values for those categories that we wish to merge

Merging levels

Combine categories `Small` and `Medium` into a new level `Sm-Md`. Here's how to do it:

```
# merging some levels
levels(sizes) <- c("Sm-Md", "Sm-Md", "Large")

sizes

## [1] Sm-Md Sm-Md Sm-Md Large Sm-Md Large
## Levels: Sm-Md < Large
```

Merging levels

Here's another example using a list:

```
set.seed(222)
y <- sample(letters[1:5], 15, rep = TRUE)
v <- as.factor(y)

new_levels <- list(I = c("a", "e"), II = c("b", "c", "d"))
levels(v) <- new_levels

v

##  [1] I  I  II I  I  I  II II II I  II I  I  II I
## Levels: I II
```

Unclassing factors

- Factors are stored as vectors of integers (for efficiency reasons)
- Even though a factor may be displayed with string labels, the way it is stored internally is as integers
- Sometimes you'll ~~nee~~ to know what numbers are associated to each level values

Unclassing factors

```r
# some factor
xfactor <- factor(c(22, 11, 44, 33, 11, 22, 44))
xfactor

## [1] 22 11 44 33 11 22 44
## Levels: 11 22 33 44

# unclassing a factor
unclass(xfactor)

## [1] 2 1 4 3 1 2 4
## attr(,"levels")
## [1] "11" "22" "33" "44"
```

Note that the levels `("11" "22" "33" "44")` were mapped to the vector of integers `(1 2 3 4)`

Unclassing factors

Although rarely used, there can be some cases in which what you need to do is revert the integer values in order to get the original factor levels. This is only possible when the levels of the factor are themselves numeric. To accomplish this use the following command:

```
# recovering numeric levels
as.numeric(levels(xfactor))[xfactor]

## [1] 22 11 44 33 11 22 44
```

Dropping Levels

- It is common to get a sample or subset of a factor
- The obtained factor may have less levels that the original factor
- When this happens, we may want to drop unused levels
- This can be achieved with the function `droplevels()`

Dropping Levels with droplevels()

```r
# original factor
vowels <- factor(c('a', 'a', 'e', 'i', 'o', 'u', 'i'))

# subset
subvowels <- vowels[1:4]
subvowels

## [1] a a e i
## Levels: a e i o u

# drop unused levels
droplevels(subvowels)

## [1] a a e i
## Levels: a e i
```

Categorizing Quantitative Variables

- A common task is getting a categorical variable from a quantitative variable
- In other words, discretize or categorize a quantitative variable
- For this task R provides the function `cut()`
- The idea is to *cut* values into intervals

Function cut()

Continuous values are converted into intervals, which in turn will be the levels of the generated factor

```
cut(x, breaks, labels = NULL, include.lowest = FALSE,
    right = TRUE, dig.lab = 3, ordered_result = FALSE
```

Function `cut()`

Arguments of `cut()`

- `x` a numeric vector which is to be converted to a factor by cutting.
- `breaks` numeric vector giving the number of intervals into which `x` is to be cut.
- `labels` labels for the levels of the resulting category.
- `include.lowest` logical indicating if values equal to the lowest 'breaks' point should be included.
- `right` logical, indicating if the intervals should be closed on the right.
- `dig.lab` integer which is used when labels are not given.
- `ordered_result` logical: should the result be an ordered factor?

Function cut()

```
# cutting a quantitative variable
set.seed(321)
income <- round(runif(n = 1000, min = 100, max = 500), 2)

# cut income in 5 classes
income_level <- cut(x = income, breaks = 5)
table(income_level)

## income_level
## (99.7,180]  (180,260]  (260,340]  (340,420]  (420,500]
##        191        197        184        218        210
```

Function cut()

By default, `cut()` has its argument `right` set to `TRUE`. This means that the intervals are open on the left (and closed on the right):

```
# using other cutting break points
income_breaks <- seq(from = 100, to = 500, by = 100)
income_a <- cut(x = income, breaks = income_breaks)

table(income_a)

## income_a
## (100,200] (200,300] (300,400] (400,500]
##       250       225       259       266

sum(table(income_a))

## [1] 1000
```

Function cut()

To change the default way in which intervals are open and closed you can set `right = FALSE`. This option produces intervals closed on the left and open on the right:

```
# using other cutting break points
income_b <- cut(x = income, breaks = income_breaks,
                right = FALSE)

table(income_b)

## income_b
## [100,200) [200,300) [300,400) [400,500)
##       250       225       259       266

sum(table(income_b))

## [1] 1000
```

Function `gl()`

In addition to the function `factor()`, there's `gl()`. This function generates factors by specifying a pattern of levels:

```
gl(n, k, length = n*k, labels = seq_len(x),
   ordered = FALSE)
```

Function gl()

```r
# factor with gl()
num_levs <- 4
num_reps <- 3

simple_factor <- gl(num_levs, num_reps)
simple_factor

##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4
```

The main inputs of `gl()` are `n` and `k`, that is, the number of levels and the number of replications of each level.

Function gl()

Here's another example setting the arguments labels and length:

```
# another factor with gl()
girl_boy <- gl(2, 4, labels = c("girl", "boy"), length = 7)
girl_boy

## [1] girl girl girl girl boy  boy  boy
## Levels: girl boy
```

By default, the total number of elements is 8 (n=2 $\times$ k=4). Four girl's and four boy's. But since we set the argument length = 7, we only got three boy's.

Function gl()

```r
# frequencies (counts)
table(girl_boy)

## girl_boy
## girl  boy
##    4    3

# frequencies (percents)
prop.table(table(girl_boy))

## girl_boy
##      girl       boy
## 0.5714286 0.4285714
```

Dates

Dates

- Dates and times are very common in data analysis
- We can distinguish between dates, and date-times
- Dates consist of year, month and day: `2015-06-11`
- Date-times consist of both a date and a specific time: `2015-06-11 09:35:24`
- R provides various options and packages for dealing with date and time data
- There are 3 date and times classes that come with R: `POSIXct`, `POSIXlt`, `Date`

POSIX Dates

- POSIX stands for **Portable Operating System Interface**
- POSIX is a family of standards for the design of operating systems
- It is especially used for operating systems that are compatible with Unix
- There is a POSIX format for date-times
- POSIX date-times allow modification of time zones

POSIX Dates

- There are two POSIX classes to store date-times: `POSIXct` (calendar time) and `POSIXlt` (list)
- `POSIXct` date-times values are given as number of seconds since January 1, 1970, in the Coordinated Universal Time (UTC) zone
- `POSIXlt` date-times values are stored as a list with elements for second, minute, hour, day, month, and year
- the `POSIXct` is the usual choice for storing date-times in R

POSIX Dates

The function `Sys.time()` gives the current date and time in `POSIXct` format

```
# current date and time
Sys.time()

## [1] "2015-06-12 08:10:21 PDT"
```

POSIXct Dates

```r
# current date and time
now_ct <- Sys.time()

now_ct

## [1] "2015-06-12 08:10:21 PDT"

class(now_ct)

## [1] "POSIXct" "POSIXt"
```

Even though the date is displayed like a `character`, the class `POSIXct` is not stored as characters.

POSIXct Dates

```r
# unclass 'now_ct'
unclass(now_ct)

## [1] 1434121822
```

Unclassing a `POSIXct` date-time gives you the number of seconds from January 1, 1970

POSIXct Dates

```
# from POSIXct to POSIXlt
now_lt <- as.POSIXlt(now_ct)

now_lt

## [1] "2015-06-12 08:10:21 PDT"
```

The print display of a `POSIXlt` date-time is similar to `POSIXct`

POSIXlt Dates

```
# POSIXlt
nlt <- unclass(now_lt)
class(nlt)

## [1] "list"

nlt[1:5]

## $sec
## [1] 21.50593
##
## $min
## [1] 10
##
## $hour
## [1] 8
##
## $mday
## [1] 12
##
```

POSIXlt Dates

You can access indivual components of a POSIXlt object using list indexing:

```
# POSIXlt
now_lt$sec

## [1] 21.50593

now_lt$min

## [1] 10

now_lt$year

## [1] 115
```

Date class

- Besides `POSIXct` and `POSIXlt`, there is a third class called `Date`
- `Date` stored dates as the number of days since January 1, 1970
- `Date` class only contains date (no times)
- If you don't care about times, then `Date` is a good option to use
- See the documentation in `help(Date)`

Date class

The function `as.Date()` allows a variety of input formats. For instance, we can convert a `POSIXct` date-time into a `Date`

```
# from POSIXlt to Date
now_date <- as.Date(now_ct)

now_date

## [1] "2015-06-12"

class(now_date)

## [1] "Date"
```

Function as.Date()

```r
as.Date("2015-6-8")

## [1] "2015-06-08"

as.Date("2015-06-8")

## [1] "2015-06-08"

as.Date("2015/6/8")

## [1] "2015-06-08"
```

Function as.Date()

```r
as.Date("6/8/2015", format = "%m/%d/%Y")

## [1] "2015-06-08"

as.Date("6/8/2015", format = "%m/%d/%Y")

## [1] "2015-06-08"

as.Date("June 8, 2015", format = "%B %d, %Y")

## [1] "2015-06-08"

as.Date("8JUNE15", format = "%d%b%y")

## [1] "2015-06-08"
```

Format codes for dates

code	value
%d	day of the month (decimal number)
%m	month (decimal number)
%b	month (abbrevaited)
%B	month (full name)
%y	year (2 digit)
%Y	year (4 digit)

Useful Packages

- There are many other date and time classes provided in various R packages
- Two common packages for working with dates are `"chron"` and `"lubridate"`
- http://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html