A booklet of R factors

Gaston Sanchez
gastonsanchez.com

About this ebook

Abstract

This ebook aims to show you basic tools for handling categorical data with R factors.

About the reader

I am assuming two things about you: 1) You acknowledge the importance of data exploration and visualization; 2) you have some knowledge of the statistical software R.

Citation

Sanchez, G. (2018) **A booklet of R factors**
URL `http://www.gastonsanchez.com/rfactors.pdf`

Source

Github Repository:
https://github.com/gastonstat/rfactors

Revision

November 17, 2018
Version 1.0

Contents

Chapter 1

Categorical Data in R

I'm one of those with the humble opinion that great software for data science and analytics should have a data structure dedicated to handle categorical data. Lucky for us, R is one of the greatest. In case you're not aware, one of the nicest features about R is that it provides a data structure exclusively designed to handle categorical data: **factors**.

The term "factor" as used in R for handling categorical variables, comes from the terminology used in *Analysis of Variance*, commonly referred to as ANOVA. In this statistical method, a categorical variable is commonly referred to as *factor* and its categories are known as *levels*. Perhaps this is not the best terminology but it is the one R uses, which reflects its distinctive statistical origins. Especially for those users without a brackground in statistics, this is one of R's idiosyncracies that seems disconcerning at the beginning. But as long as you keep in mind that a factor is just the object that allows you to handle a qualitative variable you'll be fine. In case you need it, here's a short mantra to remember: *"factors have levels"*.

1.1 Creating Factors

To create a factor in R you use the homonym function `factor()`, which takes a vector as input. The vector can be either numeric, character or logical. Let's see our first example:

```
# numeric vector
num_vector <- c(1, 2, 3, 1, 2, 3, 2)

# creating a factor from num_vector
first_factor <- factor(num_vector)

first_factor
```

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

As you can tell from the previous code snippet, `factor()` converts the numeric vector `num_vector` into a factor (i.e. a categorical variable) with 3 categories —the so called `levels`.

You can also obtain a factor from a string vector:

```
# string vector
str_vector <- c('a', 'b', 'c', 'b', 'c', 'a', 'c', 'b')

str_vector
```

```
## [1] "a" "b" "c" "b" "c" "a" "c" "b"
```

```
# creating a factor from str_vector
second_factor <- factor(str_vector)

second_factor
```

```
## [1] a b c b c a c b
## Levels: a b c
```

Notice how `str_vector` and `second_factor` are displayed. Even though the elements are the same in both the vector and the factor, they are printed in different formats. The letters in the string vector are displayed with quotes, while the letters in the factor are printed without quotes.

And of course, you can use a logical vector to generate a factor as well:

```
# logical vector
log_vector <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

# creating a factor from log_vector
third_factor <- factor(log_vector)

third_factor
```

```
## [1] TRUE  FALSE TRUE  TRUE  FALSE
```

```
## Levels: FALSE TRUE
```

1.1.1 How R treats factors?

If you're curious and check the technical *R Language Definition*, available online https://cran.r-project.org/manuals.html, you'll find that R factors are referred to as *compound objects*. According to the manual: "Factors are currently implemented using an integer array to specify the actual levels and a second array of names that are mapped to the integers."

Essentially, a factor is internally stored using two arrays: one is an integer array containing the values of categories, the other array is the "levels" which has the names of categories which are mapped to the integers.

Under the hood, the way R stores factors is as vectors of integer values. One way to confirm this is using the function `storage.mode()`

```
# storage of factor
storage.mode(first_factor)

## [1] "integer"
```

This means that we can manipulate factors just like we manipulate vectors. In addition, many functions for vectors can be applied to factors. For instance, we can use the function `length()` to get the number of elements in a factor:

```
# factors have length
length(first_factor)

## [1] 7
```

We can also use the square brackets `[ ]` to extract or select elements of a factor. Inside the brackets we specify vectors of indices such as numeric vectors, logical vectors, and sometimes even character vectors.

```
# first element
first_factor[1]

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

# third element
first_factor[3]

## [1] 3
```

```
## Levels: 1 2 3

# second to fourth elements
first_factor[2:4]

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

# last element
first_factor[length(first_factor)]

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

# logical subsetting
first_factor[rep(c(TRUE, FALSE), length.out = 7)]

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

If you have a factor with named elements, you can also specify the names of the elements within the brackets:

```
names(first_factor) <- letters[1:length(first_factor)]
first_factor

## a b c d e f g
## 1 2 3 1 2 3 2
## Levels: 1 2 3

first_factor[c('b', 'd', 'f')]

## b d f
## 2 1 3
## Levels: 1 2 3
```

However, you should know that factors are NOT really vectors. To see this you can check the behavior of the functions `is.factor()` and `is.vector()` on a factor:

```
# factors are not vectors
is.vector(first_factor)

## [1] FALSE

# factors are factors
is.factor(first_factor)

## [1] TRUE
```

Even a single element of a factor is also a factor:

```
class(first_factor[1])

## [1] "factor"
```

So what makes a factor different from a vector? Well, it turns out that factors have an additional attribute that vectors don't: `levels`. And as you can expect, the class of a factor is indeed `"factor"` (not `"vector"`).

```
# attributes of a factor
attributes(first_factor)

## $levels
## [1] "1" "2" "3"
##
## $class
## [1] "factor"
##
## $names
## [1] "a" "b" "c" "d" "e" "f" "g"
```

Another feature that makes factors so special is that their values (the levels) are mapped to a set of character values for displaying purposes. This might seem like a minor feature but it has two important consequences. On the one hand, this implies that factors provide a way to store character values very efficiently. Why? Because each unique character value is stored only once, and the data itself is stored as a vector of integers.

Notice how the numeric value `1` was mapped into the character value `"1"`. And the same happens for the other values `2` and `3` that are mapped into the characters `"2"` and `"3"`.

1.2 Factors and Data Tables

Usually we get data in some file. Rarely is the case when we have to input data manually. The files can be in text format; they can also be plain text format. Typical file extensions are csv, tsv, xml, json, or whatever other format. The most frequent case is either read data as text, or read data in some type of tabular format (spreadsheet-like table).

For better or worse, when reading tabular data in R, the default behavior of

`read.table()` and similar functions (e.g. `read.csv()`, `read.fwf()`, etc), is to convert characters into factors. Unless you are totally comfortable with the way the data file has been structured, I recommend to turn off this behavior by setting the argument `stringsAsFactors = FALSE`. In this way you will have more freedom and flexibility to manipulate data and do string manipulations. Plus, you can always convert those character variables into factors or numbers.

There is a caveat with `stringsAsFactors = FALSE`. The reason why R converts characters into factors is because of memory efficiency. Internally, a `factor` is stored as an integer vector with an attribute `levels`. In general, this storage form is more efficient than storing a `character` vector. A better way to see this is with an example. We'll use the popular `iris` dataset which contains the categorical variable `Species`:

```
# iris data set
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

# Species as factor
class(iris$Species)

## [1] "factor"
```

Let's compare the sizes of apparently the same type of data:

```
# species in two formats
iris_factor <- iris$Species
iris_string <- as.character(iris$Species)

# comparison of memory size
object.size(iris_factor)

## 1192 bytes

object.size(iris_string)

## 1400 bytes
```

Note that the size of the factor `iris_factor` is less than the character

vector `iris_string`.

For objects with a few number of elements, character vectors will be of smaller size than factors. For instance, consider the following example. Here we have a character vector of color names with 5 elements, and a factor derived from the vector of colors.

```
# vector of colros
colrs <- c('blue', 'red', 'blue', 'green', 'red')

# factor
cols <- factor(colrs)
```

If we compare the size of the objects, you'll notice that the vector occupies much less memory than the factor

```
# comparing sizes
object.size(colrs)

## 232 bytes

object.size(cols)

## 608 bytes
```

1.2.1 What is the advantage of R factors?

Every time I teach about factors, there is inevitably one student who asks a very pertinent question: Why do we want to use factors? Isn't it redundant to have a `factor` object when there are already character or integer vectors?

I have two answers to this question.

The first has to do with the storage of factors. Storing a factor as integers will usually be more efficient than storing a character vector. As we've seen, this is an important issue especially when factors are of considerable size. The second reason has to do with *ordinal* variables. Qualitative data can be classified into nominal and ordinal variables. Nominal variables could be easily handled with character vectors. In fact, nominal means name (values are just names or labels), and there's no natural order among the categories. A different case is when we have ordinal variables, like sizes `"small", "medium", "large"` or college years `"freshman", "sophomore", "junior", "senior"`. In these cases we are still using names of categories, but they can be arranged in increasing or decreasing order. In other words, we can rank the categories since they

have a natural order: small is less than medium which is less than large. Likewise, freshman comes first, then sophomore, followed by junior, and finally senior.

So here's an important question: How do we keep the order of categories in an ordinal variable? We can use a character vector to store the values. But a character vector does not allow us to store the ranking of categories. The solution in R comes via factors. We can use factors to define ordinal variables, like the following example:

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

sizes
```

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

We'll take in more detail about ordinal factors in the next chapter. For now, just keep in mind the `sizes` example. As you can tell, `sizes` has ordered levels, clearly identifying the first category `"sm"`, the second one `"md"`, and the third one `"lg"`

1.2.2 When to factor?

As mentioned above, all the reading table functions—e.g. `read.table()`, `read.csv()`, `read.delim()`, `read.fwf()`, etc.—import data tables as objects of class `data.frame`. In turn, the creation of data frames (e.g. via `data.frame()`) converts, by default, character strings into factors. We've talked about the `stringsAsFactors` argument of the function `data.frame()`. This is the argument that allows us to turn on or turn off the conversion of character strings into factors.

So here's another question: **When do we want characters to become factors?** There is no universal answer to this question. The decision of whether to convert strings into factors is going to depend on various aspects. For instance, the purpose of the analysis, or the type of variable that contains the strings. Sometimes it will make sense to have a variable as `factor`, like *gender* (e.g. male, female) or *ethnicty* (e.g. hispanic, african-american, native-american). In other cases it does not make much sense to create a factor from a variable containing addresses or telephone numbers

or names of individuals.

There is also a another aspect related to the question of whether to convert strings as factors or not. In practice, most of the times we'll be working with some data set. The data we'll be provided with is what I call the **raw data**. Pretty much all real-life data analysis projects require the analyst to process, clean, and transform the raw data into a clean data version, also called *tidy* data. Since the processing-cleaning phase involves manipulating text, formatting, grouping, changing scales, splitting strings, and a wide variety of operations, it is better to import the raw data and leave strings as characters. Once the clean data set has been created, then we can work with this version and convert some of the strings into factors.

Chapter 2

A closer look at `factor()`

Since working with categorical data in R typically involves working with factors, you should become familiar with the variety of functions related with them. In the following sections we'll cover a bunch of topics and details about factors so you can be better prepared to deal with any type of categorical data.

2.1 Function `factor()`

Given the fundamental role played by the function `factor()` we need to pay a closer look at its arguments. If you check the documentation — `help(factor)`— you'll see that the usage of the function `factor()` is:

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax = NA)
```

with the following arguments:

- `x` a vector of data
- `levels` an optional vector for the categories
- `labels` an optional character vector of labels for the levels
- `exclude` a vector of values to be excluded when forming the set of levels

- `ordered` logical value to indicate if the levels should be regarded as ordered
- `nmax` an upper bound on the number of levels

The main argument of `factor()` is the input vector `x`. The next argument is `levels`, followed by `labels`, both of which are optional arguments. Although you won't always be providing values for `levels` and `labels`, it is important to understand how R handles these arguments by default.

Argument `levels`. If `levels` is not provided (which is what happens in most cases), then R assigns the unique values in `x` as the category levels.

For example, consider our numeric vector from the first example: `num_vector` contains unique values 1, 2, and 3.

```
# numeric vector
num_vector <- c(1, 2, 3, 1, 2, 3, 2)

# creating a factor from num_vector
first_factor <- factor(num_vector)

first_factor
```

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

Now imagine we want to have `levels` 1, 2, 3, 4, and 5. This is how you can define the factor with an extended set of levels:

```
# numeric vector
num_vector
```

```
## [1] 1 2 3 1 2 3 2
```

```
# defining levels
one_factor <- factor(num_vector, levels = 1:5)
one_factor
```

```
## [1] 1 2 3 1 2 3 2
## Levels: 1 2 3 4 5
```

Although the created factor only has values between 1 and 3, the `levels` range from 1 to 5. This can be useful if we plan to add elements whose values are not in the input vector `num_vector`. For instance, you can append two more elements to `one_factor` with values `4` and `5` like this:

```
# adding values 4 and 5
one_factor[c(8, 9)] <- c(4, 5)
one_factor
```

```
## [1] 1 2 3 1 2 3 2 4 5
## Levels: 1 2 3 4 5
```

If you attempt to insert an element having a value that is not in the predefined set of levels, R will insert a missing value (`<NA>`) instead, and you'll get a warning message like the one below:

```
# attempting to add value 6 (not in levels)
one_factor[1] <- 6
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = 6):  invalid factor level,
NA generated
```

```
one_factor
```

```
## [1] <NA> 2    3    1    2    3    2    4    5
## Levels: 1 2 3 4 5
```

Argument `labels`. Another very useful argument is `labels`, which allows you to provide a string vector for naming the `levels` in a different way from the values in `x`. Let's take the numeric vector `num_vector` again, and say we want to use words as labels instead of numeric values. Here's how you can create a factor with predefined `labels`:

```
# defining labels
num_word_vector <- factor(num_vector, labels = c("one", "two", "three"))

num_word_vector
```

```
## [1] one   two   three one   two   three two
## Levels: one two three
```

Argument `exclude`. If you want to ignore some values of the input vector `x`, you can use the `exclude` argument. You just need to provide those values which will be removed from the set of `levels`.

```
# excluding level 3
factor(num_vector, exclude = 3)
```

```
## [1] 1    2    <NA> 1    2    <NA> 2
## Levels: 1 2
```

```
# excluding levels 1 and 3
factor(num_vector, exclude = c(1,3))

## [1] <NA> 2    <NA> <NA> 2    <NA> 2
## Levels: 2
```

The side effect of `exclude` is that it returns a missing value (`<NA>`) for each element that was excluded, which is not always what we want. Here's one way to remove the missing values when excluding 3:

```
# excluding level 3
num_fac12 <- factor(num_vector, exclude = 3)

# oops, we have some missing values
num_fac12

## [1] 1    2    <NA> 1    2    <NA> 2
## Levels: 1 2

# removing missing values
num_fac12[!is.na(num_fac12)]

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

2.1.1 Manipulating Categories or Levels

We've seen how to work with the argument `levels` inside the function `factor()`. But that's not the only way in which you can manipulate the levels of a factor. Closely related to `factor()` there are two other important sibling functions: `levels()` and `nlevels()`.

The function `levels()` lets you have access to the `levels` attribute of a factor. This means that you can use `levels()` for both: **getting** the categories, and **setting** the categories.

Getting levels. To get the different values for the categories in a factor you just need to apply `levels()` on a factor:

```
# levels()
levels(first_factor)

## [1] "1" "2" "3"

levels(third_factor)
```

```
## [1] "FALSE" "TRUE"
```

Setting levels. If what you want is to specify the `levels` attribute, you must use the function `levels()` followed by the assignment operator `<-`. Suppose that we want to change the levels of `first_factor` and express them in roman numerals. You can achieve this with:

```
# copy of first factor
first_factor_copy <- first_factor
first_factor_copy
```

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

```
# setting new levels
levels(first_factor_copy) <- c("I", "II", "III")
first_factor_copy
```

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

Number of Levels. Besides the function `levels()` there's another very handy function: `nlevels()`. This function allows you to return the number of levels of a factor. In other words, `nlevels()` returns the length of the attribute `levels`:

```
# nlevels()
nlevels(first_factor)
```

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

```
# equivalent to
length(levels(first_factor))
```

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

Don't confuse `length()` with `nlevels()`. The former returns the number of elements in a factor, while the latter returns the number of levels.

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()` and assigning a new vector of levels containing repeated values for those categories that we wish to merge.

For example, say we want to combine categories `I` and `III` into a new level `I+III`. Here's how to do it:

```
# nlevels()
levels(first_factor) <- c("I+III", "II", "I+III")

# equivalent to
first_factor
```

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

Note that the length of the vector specifying the merged categories will be the same as the number of levels.

Factors with missing values. Missing values are ubiquitous and they can appear in any data set. This means that we can have categorical variables with missing values. For instance, let's say we have a vector *drinks* with the type of drink ordered by a group of 7 individuals:

```
# vector of drinks
drink_type <- c('water', 'water', 'beer', 'wine', 'soda', 'water', NA)
drink_type
```

```
## [1] "water" "water" "beer"  "wine"  "soda"  "water" NA
```

As you can tell from the vector `drink_type`, there is a missing value for the last element. Now let's convert this vector into a factor:

```
# drinks factor
drinks <- factor(drink_type)
drinks
```

```
## [1] water water beer  wine  soda  water <NA>
## Levels: beer soda water wine
```

Missing values in factors are displayed as `<NA>` instead of just `NA`.

What if you want to consider a missing value as another level? To do this, first you need to add a new level to the factor. For instance, you can take the current levels and append a string `"NA"` that will be the level of the missing values:

```
# add extra level 'NA'
levels(drinks) <- c(levels(drinks), "NA")

drinks
```

```
## [1] water water beer  wine  soda  water <NA>
## Levels: beer soda water wine NA
```

Now that there is a level dedicated to missing values, you can assign a string `"NA"` to the actual missing values:

```
drinks[is.na(drinks)] <- "NA"

drinks
```

```
## [1] water water beer  wine  soda  water NA
## Levels: beer soda water wine NA
```

Notice that `drinks` has now a level for `NA`. Notice also that this label is not anymore displayed as `<NA>`. In other words, this is not an R missing value anymore. It is just another category or level:

```
is.na(drinks)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

2.1.2 Ordinal factors

By default, `factor()` creates a *nominal* categorical variable, not an ordinal. One way to check that you have a nominal factor is to use the function `is.ordered()`, which returns `TRUE` if its argument is an ordinal factor.

```
# ordinal factor?
is.ordered(num_vector)
```

```
## [1] FALSE
```

If you want to specify an ordinal factor you must use the `ordered` argument of `factor()`. This is how you can generate an ordinal value from `num_vector`:

```
# ordinal factor from numeric vector
ordinal_num <- factor(num_vector, ordered = TRUE)
ordinal_num
```

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

As you can tell from the snippet above, the levels of `ordinal_factor` are displayed with less-than symbols `'<'`, which means that the levels have an increasing order. We can also get an ordinal factor from our string vector:

```
# ordinal factor from character vector
ordinal_str <- factor(str_vector, ordered = TRUE)
ordinal_str
```

```
## [1] a b c b c a c b
## Levels: a < b < c
```

In fact, when you set `ordered = TRUE`, R sorts the provided values in alphanumeric order. If you have the following alphanumeric vector `("a1", "1a", "1b", "b1")`, what do you think will be the generated ordered factor? Let's check the answer:

```
# alphanumeric vector
alphanum <- c("a1", "1a", "1b", "b1")

# ordinal factor from character vector
ordinal_alphanum <- factor(alphanum, ordered = TRUE)
ordinal_alphanum
```

```
## [1] a1 1a 1b b1
## Levels: 1a < 1b < a1 < b1
```

An alternative way to specify an ordinal variable is by using the function `ordered()`, which is just a convenient wrapper for `factor(x, ..., ordered = TRUE)`:

```
# ordinal factor with ordered()
ordered(num_vector)
```

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

```
# same as using 'ordered' argument
factor(num_vector, ordered = TRUE)
```

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

A word of caution. Don't confuse the function `ordered()` with `order()`. They are not equivalent. `order()` arranges a vector into ascending or descending order, and returns the sorted vector. `ordered()`, as we've seen, is used to get ordinal factors.

Of course, you won't always be using the default order provided by the functions `factor(..., ordered = TRUE)` or `ordered()`. Sometimes you want to determine categories according to a different order.

For example, let's take the values of `str_vector` and let's assume that we want them in descending order, that is, `c < b < a`. How can you do that? Easy, you just need to specify the `levels` in the order you want them and set `ordered = TRUE` (or use `ordered()`):

```
# setting levels with specified order
factor(str_vector, levels = c("c", "b", "a"), ordered = TRUE)

## [1] a b c b c a c b
## Levels: c < b < a

# equivalently
ordered(str_vector, levels = c("c", "b", "a"))

## [1] a b c b c a c b
## Levels: c < b < a
```

Here's another example. Consider a set of size values `"xs"` extra-small, `"sm"` small, `"md"` medium, `"lg"` large, and `"xl"` extra-large. If you have a vector with size values you can create an ordinal variable as follows:

```
# vector of sizes
sizes <- c("sm", "xs", "xl", "lg", "xs", "lg")

# setting levels with specified order
ordered(sizes, levels = c("xs", "sm", "md", "lg", "xl"))

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

Notice that when you create an ordinal factor, the given `levels` will always be considered in an increasing order. This means that the first value of `levels` will be the smallest one, then the second one, and so on. The last category, in turn, is taken as the one at the top of the scale.

Now that we have several nominal and ordinal factors, we can compare the behavior of `is.ordered()` on two factors:

```
# is.ordered() on an ordinal factor
ordinal_str

## [1] a b c b c a c b
## Levels: a < b < c

is.ordered(ordinal_str)

## [1] TRUE

# is.ordered() on a nominal factor
```

```
second_factor
```

```
## [1] a b c b c a c b
## Levels: a b c
```

```
is.ordered(second_factor)
```

```
## [1] FALSE
```

2.1.3 Unclassing factors

We've mentioned that factors are stored as vectors of integers (for efficiency reasons). But we also said that factors are more than vectors. Even though a factor is displayed with string labels, the way it is stored internally is as integers. Why is this important to know? Because there will be occasions in which you'll need to know exactly what numbers are associated to each level values.

Imagine you have a factor with `levels` 11, 22, 33, 44.

```
# 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
```

To obtain the integer vector associated to `xfactor` you can use the function `unclass()`:

```
# unclassing a factor
unclass(xfactor)
```

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

As you can see, the `levels ("11" "22" "33" "44")` were mapped to the vector of integers `(1 2 3 4)`.

An alternative option is to simply apply `as.numeric()` or `as.integer()` instead of using `unclass()`:

```
# equivalent to unclass
as.integer(xfactor)
```

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

```
# equivalent to unclass
as.numeric(xfactor)
```

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

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
```

2.1.4 Reordering factors

Sometimes it's not enough with creating an ordinal variable or setting the order of the categories. Occasionally, you would like to reorder a factor. For this purpose you can use the function `reorder()`

```
vowels <- c('a', 'b', 'c', 'd', 'e')
set.seed(975)
vowels_fac <- factor(sample(vowels, size = 20, replace = TRUE))
#reorder(vowels_fac, count, median)

# reordering a factor
bymedian <- with(InsectSprays, reorder(spray, count, median))
```

Reordering levels. Another useful function to reorder the levels of a factor is `relevel()`. This function has two main arguments: an unordered factor `x`, and a reference level `ref`. What `relevel()` does is to re-order the levels of a factor so that the level specified by `ref` will be the first level, while the others are moved down.

For example, consider `one_factor`, which is an unordered factor with levels 1, 2, 3, 4, 5. We can use `relevel()` to specify `ref=5` as the reference level. This will cause 5 to be the first level, while the rest of the levels will be moved down:

```
one_factor
```

```
## [1] <NA> 2    3    1    2    3    2    4    5
```

```
## Levels: 1 2 3 4 5

# reorder levels of unordered factor
relevel(one_factor, ref = 5)

## [1] <NA> 2    3    1    2    3    2    4    5
## Levels: 5 1 2 3 4
```

2.1.5 Dropping levels

"There are two tasks that are often performed on factors. One is to drop unused levels; this can be achieved by a call to `factor()` since `factor(y)` will drop any unused levels from `y` if `y` is a factor."

"The second task is to coarsen the levels of a factor, that is, group two or more of them together into a single new level."

```
y <- sample(letters[1:5], 20, rep = TRUE)
v <- as.factor(y)
xx <- list(I = c("a", "e"), II = c("b", "c", "d"))
levels(v) <- xx
v

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

Chapter 3

More about Factors

So far we've seen a comprehensive review of the functions `factor()`, and `levels()`. Now we'll talk about other accessory functions for creating and handling factors in R.

3.1 Categorizing a quantitative variable

A common data manipulation task is: how to get a categorical variable from a quantitative variable? In other words, how to discretize or categorize a quantitative variable?

For this kind of common task R provides the handy function `cut()`. The idea is to *cut* values of a numeric input vector into intervals, which in turn will be the levels of the generated factor. The usage of `cut()` is:

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

with the following arguments:

- `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?

Example Here's an example. The following code creates a numeric vector, `income`, that generates some fake values of a hypothetical variable income.

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

To convert `income` into a factor we use `cut()`. The first argument is the input vector (`income` in this case). The argument `breaks` is used to indicate the number of categories or levels of the output factor (e.g. 10)

```
# cutting a quantitative variable
income_level <- cut(x = income, breaks = 10)

levels(income_level)
```

```
##  [1] "(99.7,140]" "(140,180]"  "(180,220]"  "(220,260]"  "(260,300]"
##  [6] "(300,340]"  "(340,380]"  "(380,420]"  "(420,460]"  "(460,500]"
```

As you can tell, `income_level` has 10 levels; each level formed by an interval. Moreover, the intervals are all of the same form: a range of values with the lower bound surrounded by a parenthesis, and the upper bound surrounded by a bracket.

You can inspect the produced factor `income_level` and check the frequencies with `table()`

```
table(income_level)
```

```
## income_level
## (99.7,140]  (140,180]  (180,220]  (220,260]  (260,300]  (300,340]
##         85        106        113         84         87         97
##  (340,380]  (380,420]  (420,460]  (460,500]
##        115        103         98        112
```

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 = 50)
income_a <- cut(x = income, breaks = income_breaks)
table(income_a)
```

```
## income_a
## (100,150] (150,200] (200,250] (250,300] (300,350] (350,400] (400,450]
##       111       139       122       103       124       135       131
## (450,500]
##       135
```

```
sum(table(income_a))
```

```
## [1] 1000
```

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
## [100,150) [150,200) [200,250) [250,300) [300,350) [350,400) [400,450)
##       111       139       122       103       124       135       131
## [450,500)
##       135
```

```
sum(table(income_b))
```

```
## [1] 1000
```

You can change the labels of the levels using the argument `labels`. For example, let's say we want to name the resulting levels with letters. The first level [100,150) will be changed to `"a"`, the second level [150,200) will be changed to `"b"`, and so on.

```
income_c <- cut(x = income, breaks = income_breaks,
                labels = letters[1:(length(income_breaks)-1)])

table(income_c)
```

```
## income_c
##   a   b   c   d   e   f   g   h
## 111 139 122 103 124 135 131 135
```

3.1.1 Factor into indicators

Another frequent operation is to decompose a categorical variable into indicators, also known as dummy variables. The idea is to create a table (rectangular array or matrix) with as many columns as levels. Each column is a binary vairbale (0, or 1).

You can think of this as "unfolding" a factor. Other authors call it creating a disjunctive table. Each row has only one 1, and the rest of values are zeros. The sum of values in a column equals the number of elements in that particular category.

Say you have a factor with category temperatures `hot` and `cold`. One way to obtain dummy indicators for each temperatur level is to construct a matrix with as many columns as categories to binarize:

```
# example
hot_cold = gl(n = 2, k = 3, labels = c('hot', 'cold'))

hot_cold_mat = matrix(0, nrow = length(hot_cold), ncol = nlevels(hot_cold))
hot_cold_mat[hot_cold == 'hot', 1] = 1
hot_cold_mat[hot_cold == 'cold', 2] = 1
dimnames(hot_cold_mat) = list(1:length(hot_cold), c('hot', 'cold'))
hot_cold_mat
```

```
##   hot cold
## 1   1    0
## 2   1    0
## 3   1    0
## 4   0    1
## 5   0    1
## 6   0    1
```

```
# sum of columns equals elements in each category
colSums(hot_cold_mat)
```

```
##  hot cold
##    3    3
```

3.1.2 Generating Factors Levels with `gl()`

In addition to the function `factor()`, there's a secondary function that you can use to create factors with a simple structure: `gl()`. This function generates factors by specifying a pattern of levels. Here's its usage:

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

with the following arguments:

- `n` an integer giving the number of levels.
- `k` an integer giving the number of replications.
- `length` an integer giving the length of the result.
- `labels` an optional vector of labels for the resulting factor levels.
- `ordered` logical indicating whether the result should be ordered or not.

Here's an example on how to use `gl()`:

```
# 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. Especially for working with data under the approach of *Design of Experiments* (DoE), `gl()` can be very useful.

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.