

The American Statistician



ISSN: 0003-1305 (Print) 1537-2731 (Online) Journal homepage: <http://amstat.tandfonline.com/loi/utas20>

Wrangling Categorical Data in R

Amelia McNamara & Nicholas J. Horton

To cite this article: Amelia McNamara & Nicholas J. Horton (2018) Wrangling Categorical Data in R, *The American Statistician*, 72:1, 97-104, DOI: [10.1080/00031305.2017.1356375](https://doi.org/10.1080/00031305.2017.1356375)

To link to this article: <https://doi.org/10.1080/00031305.2017.1356375>



[View supplementary material](#) 



Accepted author version posted online: 27

Jul 2017.

Published online: 27 Jul 2017.



[Submit your article to this journal](#) 



Article views: 436



[View Crossmark data](#) 

Wrangling Categorical Data in R

Amelia McNamara ^a and Nicholas J. Horton  ^b

^aProgram in Statistical and Data Sciences, Smith College, Northampton, MA; ^bDepartment of Mathematics and Statistics, Amherst College, Amherst, MA

ABSTRACT

Data wrangling is a critical foundation of data science, and wrangling of categorical data is an important component of this process. However, categorical data can introduce unique issues in data wrangling, particularly in real-world settings with collaborators and periodically-updated dynamic data. This article discusses common problems arising from categorical variable transformations in R, demonstrates the use of factors, and suggests approaches to address data wrangling challenges. For each problem, we present at least two strategies for management, one in base R and the other from the “tidyverse.” We consider several motivating examples, suggest defensive coding strategies, and outline principles for data wrangling to help ensure data quality and sound analysis. Supplementary materials for this article are available online.

ARTICLE HISTORY

Received May 2017

Revised June 2017

KEYWORDS

Data derivation; Data management; Data science; Statistical computing

1. Introduction

Wrangling skills provide an intellectual and practical foundation for data science. Careless data cleaning operations can lead to errors or inconsistencies in analysis (Hermans and Murphy-Hill 2015; FitzJohn et al. 2014). The wrangling of categorical data presents particular challenges and is highly relevant because many variables are categorical (e.g., gender, income bracket, U.S. state), and categorical data are often coded with numerical values. It is easy to break the relationship between category numbers and category labels without realizing it, thus losing the information encoded in a variable. If data sources change upstream (e.g., if a domain expert is providing spreadsheet data at regular intervals), code that worked on the initial data may not generate an error message, but could silently produce incorrect results.

Statistical and data science tools need to foster good practice and provide a robust environment for data wrangling and data management. This article focuses on how R deals with categorical data, and showcases best practices for categorical data manipulation in R to produce reproducible workflows. We consider a number of common idioms related to categorical data that arise frequently in data cleaning and preparation, propose some guidelines for defensive coding, and discuss settings where analysts often get tripped up when working with categorical data.

For example, data ingested into R from spreadsheets can lead to problems with categorical data because of the different storage methods possible in both R and the spreadsheets themselves (Wilson et al. 2016). The examples below help flag when these issues arise or avoid them altogether.

To ground our work, we compare and contrast how categorical data are treated in base R and the tidyverse (Wickham 2014, 2016). Tools from the tidyverse, discussed in another article in this special issue (see Ross, Wickham, and Robinson

2018), are designed to make analysis purer, more predictable, and pipeable. Key components of the tidyverse we address in this article include dplyr, tidyr, forcats, and readr. This suite of packages helps facilitate a reproducible workflow, where a new version of the data could be supplied in the code with updated results produced (Broman 2015; Lowndes et al. 2017). While R code written in base syntax can also have this quality, a common tendency is to use row or column numbers in code, which makes the result less reproducible. Wrangling of categorical data can make this task even more complex (e.g., if a new level of a categorical variable is added in an updated dataset or inadvertently introduced by a careless error in a spreadsheet to be ingested into R).

Our goal is to make the case that it is better to work with categorical data using tidyverse packages than with base R. Tidyverse code is more human readable, which can help reduce errors from the start, and the functions we highlight have been designed to make it harder to accidentally remove relationships implicit in categorical data. Because these issues are even more salient for new users, we recommend that instructors teach tidyverse approaches from the start.

2. Categorical Data in R: Factors and Strings

Consider a variable describing gender including categories male, female, and non-binary. In R, there are two ways to store this information. One is to use a series of *character strings*, and the other is to store it as a *factor*.

In early versions of R, storing categorical data as a factor variable was considerably more efficient than storing the same data as strings, because factor variables only store the factor labels once (Peng 2015; Lumley 2015). However, R now uses a global string pool, so each unique string is only stored once, which means storage is now less of an issue (Peng 2015). For historical

(or possibly anachronistic) reasons, many functions store variables by default as factors.

While factors are important when including categorical variables in regression models and when plotting data, they can be tricky to deal with, since many operations applied to them return different values than when applied to character vectors. As an example, consider a set of decades,

```
x1 <- c(10, 10, 20, 20, 40)
x1f <- factor(x1)
ds <- data.frame(x1, x1f)
library(dplyr)
ds <- ds %>%
  mutate(x1recover = as.numeric(x1f))
ds

##   x1 x1f x1recover
## 1 10 10      1
## 2 10 10      1
## 3 20 20      2
## 4 20 20      2
## 5 40 40      3
```

Instead of creating a new variable with a numeric version of the value of the factor variable `x1f`, the variable is created with a factor number (i.e., 10 is mapped to 1, 20 is mapped to 2, and 40 is mapped to 3). This result is unexpected because `base::as.numeric()` is intended to recover numeric information by coercing a character variable. Compare the following:

```
as.numeric(c("hello"))

## [1] NA

as.numeric(factor(c("hello")))

## [1] 1
```

The `factor` function has other behavior that feels unexpected. For example, the following code silently makes a missing value, because the values in the data and the levels do not match.

```
factor("a", levels="c")

## [1] <NA>

## Levels: c
```

The unfortunate behavior of factors in R has led to an online movement against the default behavior of many data import functions to make factors out of any variable composed as strings (Peng 2015; Wickham, Hester, and Francois 2017). The tidyverse is part of this movement, with functions from the `readr` package defaulting to leaving strings as-is. (Others have chosen to add `options(stringAsFactors=FALSE)` into their startup commands.)

Although the storage issues have been solved, and there are problems with defaulting strings to factors, factors are still necessary for some data analytic tasks. The most salient case is in modeling. When you pass a factor variable into `lm()` or `glm()`, R automatically creates indicator (or more colloquially “dummy”) variables for each of the levels and picks one as a reference group.

For simple cases, this behavior can also be achieved with a character vector. However, to choose which level to use as a reference level or to order classes, factors must be used. For example, if a factor encodes income levels as low, medium, high, it might make sense to use the lowest income level (`low`) as the reference class so that all the other coefficients can be interpreted in comparison to it. However, R would use `high` as the reference by default because “h” comes before “l” in the alphabet.

While ordering is particularly important when doing ordinal logistic regression and multinomial logistic regression, the use of alphabetic ordering by default means even simple linear regression can be affected.

In the context of visualizing data, factors are also relevant because they allow categorical variables to be mapped to aesthetic attributes.

While factors are important, they can often be hard to deal with. Because of the way the group numbers are stored separately from the factor labels, it can be easy to overwrite data in such a way that the original data are lost. They present a steep learning curve for new users. In this article, we will suggest best practices for working with factor data.

To motivate this process, we will consider data from the General Social Survey (Smith et al. 2015). The General Social Survey is a product of the National Data Program for the Social Sciences, and the survey has been conducted since 1972 by NORC at the University of Chicago (2016). It contains data on many aspects of social life, and is widely used by social scientists. (In this article, we consider data from 2014.)

There are some import issues inherent to the data which are not particular to categorical data (see Supplementary Appendix A for details). We will work with the data with slightly cleaned up variable names.

```
library(dplyr)
GSS <- read.csv("../data/GSScleaned.csv")
glimpse(GSS)

## Observations: 2,540
## Variables: 16
## $ Year           <int> 2014, 2014, 2014, 2014, 2014, 2014, ...
## $ ID             <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1...
## $ LaborStatus    <fctr> Working fulltime, Working fulltime, ...
## $ OccupationalPrestigeScore <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ MaritalStatus  <fctr> Divorced, Married, Divorced, Marrie...
## $ NumChildren    <int> 0, 0, 1, 2, 3, 1, 2, 2, 4, 3, 2, 0, ...
## $ Age            <fctr> 53.000000, 26.000000, 59.000000, 56...
## $ HighestSchoolCompleted <int> 16, 16, 13, 16, 17, 17, 12, 17, 10, ...
## $ Sex             <fctr> Male, Female, Male, Female, Female, ...
## $ Race            <fctr> White, White, White, White, White, ...
## $ ChildhoodFamilyIncome <fctr> Below average, Average, Below avera...
## $ TotalFamilyIncome <fctr> $25000 or more, $25000 or more, $25...
## $ RespondentIncome <fctr> $25000 or more, $25000 or more, Not...
## $ PoliticalParty  <fctr> Not str republican, Not str republi...
## $ OpinionOfIncome <fctr> Above average, Above average, Below...
## $ SexualOrientation <fctr> Heterosexual or straight, Heterosex...
```

The remainder of this article is organized around case studies (examples) to carry out four specific and useful tasks:

1. Changing the labels of factor levels,

2. Reordering factor levels,
3. Combining several levels into one (both string-like labels and numeric), and
4. Making derived factor variables.

Each case study begins with a problem and presents several solutions. Typically, we contrast a method that uses the functionality of base R with an approach from the tidyverse along with some annotations of the code as needed. We will argue that while both approaches can solve the problem, the tidyverse solution tends to be simpler, easier to learn, and less fragile.

3. Changing the Labels of Factor Levels

Our first example works with the `LaborStatus` variable. It is a categorical variable with nine levels. Most of the labels are spelled out fully, but a few are strangely formatted. We want to change this.

This is a specific case of the more general problem of changing the text of factor labels, so they appear more nicely formatted in a plot, for example.

There are two typical approaches in base R. One is more compact, but depends on the levels of the factor not changing in the data being fed in, and the other is more robust, but extremely verbose. In contrast, the `dplyr` package offers a more human readable method, while also supporting reproducibility.

3.1. Compact but Fragile (base R)

To begin this example, we will create a new copy of the variable in question so as to leave the original data for comparison.

```
GSS$BaseLaborStatus <- GSS$LaborStatus
levels(GSS$BaseLaborStatus)

## [1] "Keeping house"      "No answer"        "Other"
## [4] "Retired"           "School"           "Temp not working"
## [7] "Unempl, laid off"  "Working fulltime"  "Working parttime"

summary(GSS$BaseLaborStatus)

##   Keeping house      No answer       Other     Retired
## 263                  2                 76          460
##   School Temp not working Unempl, laid off Working fulltime
## 90                  40                104         1230
## Working parttime    NA's
## 273                  2
```

Almost all of our code examples start with some examination of the `levels()` and `summary()` of the variable, to keep track of what the expected results are. With the counts in mind, the labels can be rephrased for a few categories.

```
levels(GSS$BaseLaborStatus) <- c(levels(GSS$BaseLaborStatus)[1:5],
  "Temporarily not working",
  "Unemployed, laid off",
  "Working full time",
  "Working part time")

summary(GSS$BaseLaborStatus)

##   Keeping house      No answer       Other
## 263                  2                 76
##   Retired            School Temporarily not working
## 460                  90                40
## Unemployed, laid off Working full time Working part time
## 104                 1230               273
## NA's
## 2
```

This method is less than ideal, because it depends on the data coming in with the factor levels ordered in a particular way. The first five levels are left the same, and the last four are overwritten.

We call this a *fragile* process since future datasets may cause a workflow to break (a related concept in computer science is *software brittleness*, where a small change can lead to an error). Why is this fragile? By default, R orders factor levels alphabetically. So, `Keeping house` is first not because it is the most common response, but simply because “k” comes first in the alphabet. If the data get changed outside of R, for example so responses currently labeled `Working full time` get labeled `Full time work`, the code above will not generate an error message, but will mislabel all the data such that the `BaseLaborStatus` variable is essentially meaningless.

The issue of alphabetic ordering becomes even more relevant when considering strings that include non-ASCII characters, where the default order levels may vary from locale to locale. This means that code could create different results based on where it was run.

The workflow will also fail if additional factor levels are added after the fact. In our experience, both with students and scientific collaborators, spreadsheet data can be easily changed in these ways. Others have noted this concern (Leek 2016).

3.2. Robust but Verbose (base R)

Another (more robust method) to recode this variable in base R is to use subsetting to overwrite particular values in the data.

```
GSS$BaseLaborStatus <- GSS$LaborStatus
summary(GSS$BaseLaborStatus)

##   Keeping house      No answer       Other     Retired
## 263                  2                 76          460
##   School Temp not working Unempl, laid off Working fulltime
## 90                  40                104         1230
## Working parttime    NA's
## 273                  2

GSS$BaseLaborStatus <- as.character(GSS$BaseLaborStatus)
GSS$BaseLaborStatus[GSS$BaseLaborStatus == "Temp not working"] <-
  "Temporarily not working"
GSS$BaseLaborStatus[GSS$BaseLaborStatus == "Unempl, laid off"] <-
  "Unemployed, laid off"
GSS$BaseLaborStatus[GSS$BaseLaborStatus == "Working fulltime"] <-
  "Working full time"
GSS$BaseLaborStatus[GSS$BaseLaborStatus == "Working parttime"] <-
  "Working part time"
GSS$BaseLaborStatus <- factor(GSS$BaseLaborStatus)
summary(GSS$BaseLaborStatus)

##   Keeping house      No answer       Other
## 263                  2                 76
##   Retired            School Temporarily not working
## 460                  90                40
## Unemployed, laid off Working full time Working part time
## 104                 1230               273
## NA's
## 2
```

This second approach is more robust, because if the labels or ordering of levels changes before this code is run it will not overwrite labels on the incorrect data. However, this approach has a number of limitations in addition to being tedious and error prone. It is possible to miss cases, and misspelling and cut-and-paste errors can mean pieces of the code do not actually do anything.

3.3. Direct and Robust (*tidyverse*)

The `recode()` function in the `dplyr` package is a vectorized function, which combines the robustness of the second base R approach while also reducing the verbosity. It still suffers from the problem of misspelling and cut-and-paste errors, because it will not generate an error message if you try to recode a nonexistent level.

```
GSS <- GSS %>%
  mutate(tidyLaborStatus =
    recode(LaborStatus,
      `Temp not working` = "Temporarily not working",
      `Unempl, laid off` = "Unemployed, laid off",
      `Working fulltime` = "Working full time",
      `Working parttime` = "Working part time"))
summary(GSS$tidyLaborStatus)

## #> #> Keeping house           No answer        Other
## #> 263                      2                76
## #> Retired                  School Temporarily not working
## #> 460                      90               40
## #> Unemployed, laid off     Working full time   Working parttime
## #> 104                      1230              273
## #> NA's
## #> 2
```

In the above example, notice the trailing space in 'Working parttime' in the `recode()` call. Because of this typo (the original factor level is actually 'Working parttime'), the original factor level persists after the recode.

3.4. Aside—Editing Whitespace out of Levels

A more general problem sometimes arises due to extra spaces included when data are ingested. Such whitespace can be dealt with when data are read, or addressed later using string operations. This latter approach can be carried out using the `trimws()` function in base R.

```
gender <- factor(c("male ", "male ", "male ", "male "))
levels(gender)

## [1] "male"    "male"    "male"    "male"    "

gender <- factor(trimws(gender))
levels(gender)

## [1] "male"
```

4. Reordering Factor Levels

Often, factor levels have a natural ordering to them. However, the default in base R is to order levels alphabetically. So, users must have a way to impose order on their factor variables.

Again, there is a fragile way to reorder the factor levels in base R, and a more robust method in the `tidyverse`.

4.1. Fragile Method (base R)

One common way to make this sort of change is to pass an argument to `levels` within the `factor()` function. However, this is fragile with respect to spelling issues and trailing whitespace.

```
GSS$BaseOpinionOfIncome <- GSS$OpinionOfIncome
summary(GSS$BaseOpinionOfIncome)

## #> Above average          Average       Below average      Don't know
## #> 483                   1118          666                 21
## #> Far above average     Far below average      No answer      NA's
## #> 65                     179            6                 2
GSS$BaseOpinionOfIncome <-
  factor(GSS$BaseOpinionOfIncome,
    levels = c("Far above average", "Above average", "Average ", "Below Average",
               "Far below average", "Don't know", "No answer"))

summary(GSS$BaseOpinionOfIncome)

## #> Far above average     Above average       Average       Below Average
## #> 65                   483             0                 0
## #> Far below average     Don't know      No answer      NA's
## #> 179                  21              6                1786
```

Note that many of the category totals come through appropriately, but several totals get set to 0 (Average because of the trailing whitespace and Below Average because of the mistaken capitalization). These errors can be exceedingly frustrating to troubleshoot.

An approach that looks similar upon inspection but actually does not work is to overwrite the `levels()` of the factor outside the `factor()` command. It is tempting for new analysts to write code such as the following, which completely breaks the association between rows and factor labels the dataset.

```
badApproach <- GSS$OpinionOfIncome
summary(badApproach)

## #> Above average          Average       Below average      Don't know
## #> 483                   1118          666                 21
## #> Far above average     Far below average      No answer      NA's
## #> 65                     179            6                 2

levels(badApproach) <- c("Far above average", "Above average",
                           "Average", "Below Average", "Far below average",
                           "Don't know", "No answer")
summary(badApproach)

## #> Far above average     Above average       Average       Below Average
## #> 483                  1118             666                21
## #> Far below average     Don't know      No answer      NA's
## #> 65                   179              6                 2
```

Notice that no errors were generated, but the labels have been clobbered and the counts do not match up anymore. Instead of Far above average having 65 observations, it has 483.

Another base approach that will not suffer from spelling mistakes is to use numeric indexing to reorder the levels. Again, the indexing must take place within a `factor()` call.

```
GSS$BaseOpinionOfIncome <- GSS$OpinionOfIncome
summary(GSS$BaseOpinionOfIncome)

## #> Above average          Average       Below average      Don't know
## #> 483                   1118          666                 21
## #> Far above average     Far below average      No answer      NA's
## #> 65                     179            6                 2

GSS$BaseOpinionOfIncome <-
  factor(GSS$BaseOpinionOfIncome,
    levels=levels(GSS$BaseOpinionOfIncome)[c(5,1:3,6,4,7)])
summary(GSS$BaseOpinionOfIncome)

## #> Far above average     Above average       Average       Below average
## #> 65                   483             1118              666
## #> Far below average     Don't know      No answer      NA's
## #> 179                  21              6                 2
```

This is both verbose and depends on the number and order of the levels staying the same. If another factor level is added to the dataset, the above code will generate an error message because

the number of levels differs. This example illustrates why it is sometimes dangerous to replace an old version of a data frame with a new version.

Again, if you try this approach outside of a `factor()` call, no errors are generated but the levels get clobbered.

```
badApproach <- GSS$OpinionOfIncome
summary(badApproach)

##   Above average      Average    Below average     Don't know
##       483           1118          666             21
## Far above average Far below average      No answer      NA's
##       65            179            6              2
levels(badApproach) <- levels(badApproach)[c(5,1:3,6,4,7)]
summary(badApproach)

## Far above average      Above average      Average    Below average
##       483           1118          666             21
## Far below average      Don't know      No answer      NA's
##       65            179            6              2
```

Notice that once again, `Far above average` has been given the wrong number of observations. Here, base methods for reordering factor levels are very fragile. Approaches that appear functional and do not generate error messages can easily lead to garbled data.

4.2. Robust Method (tidyverse)

Because of the fragility and potential for frustration and mistakes associated with reordering levels in base R, we recommend the use of a tidyverse package.

The package `forcats` (where the name is an anagram of the word `factors!`) is included in the tidyverse (Wickham 2017). It includes a `fct_relevel()` function that allows for robust reordering of factor levels. It takes a specification of the order of factor levels (either completely or partially) and is robust to rerunning code in an interactive session.

```
library(forcats)
summary(GSS$OpinionOfIncome)

##   Above average      Average    Below average     Don't know
##       483           1118          666             21
## Far above average Far below average      No answer      NA's
##       65            179            6              2

GSS <- GSS %>%
  mutate(tidyOpinionOfIncome =
    fct_relevel(OpinionOfIncome,
                "Far above average",
                "Above average",
                "Average",
                "Below average",
                "Far below average"))
summary(GSS$tidyOpinionOfIncome)

## Far above average      Above average      Average    Below average
##       65            483           1118          666
## Far below average      Don't know      No answer      NA's
##      179            21              6              2
```

Notice the levels unmentioned in the function call end up at the back end of the ordering, but all the counts are appropriate.

5. Combining Several Levels into One

5.1. Combining Discrete Levels

This is another common task. Maybe you want fewer coefficients in your model, or the data-generating process makes a

finer distinction between categories than your research. For whatever the reason, you want to group together levels that are currently separate.

5.1.1. Fragile Method (base R)

This method overwrites the labels of factor levels with repeated labels to group levels together.

```
GSS$BaseMarital <- GSS$MaritalStatus
summary(GSS$BaseMarital)

##   Divorced      Married Never married      No answer     Separated
##       411          1158          675             4            81
## Widowed      NA's
##       209          2

levels(GSS$BaseMarital) <- c("Not married", "Married",
                             "Not married", "No answer",
                             "Not married", "Not married", NA)
summary(GSS$BaseMarital)

## Not married      Married      No answer      NA's
##      1376          1158          4            2
```

As before, this is fragile because it depends on the order of the factor levels not changing, and on a human accurately counting the indices of all the levels they wish to change.

5.1.2. Robust Method (tidyverse)

In the tidyverse, the `recode()` function performs recoding more robustly.

```
summary(GSS$MaritalStatus)

##   Divorced      Married Never married      No answer     Separated
##       411          1158          675             4            81
## Widowed      NA's
##       209          2

GSS <- GSS %>%
  mutate(tidyMaritalStatus = recode(MaritalStatus,
                                    Divorced = "Not married",
                                    `Never married` = "Not married",
                                    Widowed = "Not married",
                                    Separated = "Not married"))
summary(GSS$tidyMaritalStatus)

## Not married      Married      No answer      NA's
##      1376          1158          4            2
```

In contrast to the base approach, the tidyverse approach allows the analyst to only mention the levels that need to be recoded. The levels do not need to be presented in the order they originally appeared (note that `Widowed` appears earlier in the list than it does in the `summary()`).

5.2. Combining Numeric-Type Levels

Combining numeric-type levels is a frequently-occurring problem even when `stringsAsFactors = FALSE`. Often variables like age or income are right-censored, so there is a final category that lumps the remainder of people into one group. This means the data are necessarily at least a character string if not a factor. However, it may be more natural to work with numeric expressions when recoding these data.

In these data, age is provided as an integer for respondents 18–88, but also includes the possible answers 89 or older, `No answer` and `NA`.

A common data wrangling task might be to turn this into a factor variable with two levels: 18–65, and over 65. In this case, it would be easier to deal with a conditional statement about the

numeric values, rather than writing out each of the numbers as a character vector.

5.2.1. Fragile Method (base R)

To break these data apart as simply as possible, it needs to be numeric. The first step is to recode the label for 89 or older to read 89.

```
GSS$BaseAge <- GSS$Age
levels(GSS$BaseAge)

## [1] "18.000000" "19.000000" "20.000000" "21.000000" "22.000000"
## [6] "23.000000" "24.000000" "25.000000" "26.000000" "27.000000"
## [11] "28.000000" "29.000000" "30.000000" "31.000000" "32.000000"
## [16] "33.000000" "34.000000" "35.000000" "36.000000" "37.000000"
## [21] "38.000000" "39.000000" "40.000000" "41.000000" "42.000000"
## [26] "43.000000" "44.000000" "45.000000" "46.000000" "47.000000"
## [31] "48.000000" "49.000000" "50.000000" "51.000000" "52.000000"
## [36] "53.000000" "54.000000" "55.000000" "56.000000" "57.000000"
## [41] "58.000000" "59.000000" "60.000000" "61.000000" "62.000000"
## [46] "63.000000" "64.000000" "65.000000" "66.000000" "67.000000"
## [51] "68.000000" "69.000000" "70.000000" "71.000000" "72.000000"
## [56] "73.000000" "74.000000" "75.000000" "76.000000" "77.000000"
## [61] "78.000000" "79.000000" "80.000000" "81.000000" "82.000000"
## [66] "83.000000" "84.000000" "85.000000" "86.000000" "87.000000"
## [71] "88.000000" "89 or older" "No answer"

levels(GSS$BaseAge) <- c(levels(GSS$BaseAge)[1:71], "89", "No answer")
```

This code is already fragile because of its reliance on numeric indexing. From the `levels()` output, the first 71 levels correspond to the ages 18–88, and are in the expected order, so these are left as-is. Then, 89 or older is overwritten with simply 89. Finally, the variable can be converted to a character vector and then to a numeric one.

```
GSS$BaseAge <- as.numeric(as.character(GSS$BaseAge))
summary(GSS$BaseAge)

##   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
## 18.00 34.00 49.00 49.01 62.00 89.00 11
```

This avoids the pitfall from the introduction by not using `as.numeric()` on the factor variables (this would convert 18 to 1, 19 to 2, etc.). This method cheats a little—if the goal were to use this as a numeric variable in an analysis it would not be appropriate to turn all the 89 or older cases into the number 89. In this case, the goal is to create a two-level factor, so those cases would be assigned to the 65 and up category one way or the other.

Once the variable is numeric, some conditional logic can be applied to split into two cases.

```
summary(GSS$BaseAge)

##   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
## 18.00 34.00 49.00 49.01 62.00 89.00 11

GSS$BaseAge <- ifelse(GSS$BaseAge < 65, "18-64", "65 and up")
GSS$BaseAge <- factor(GSS$BaseAge)
summary(GSS$BaseAge)

##   18-64 65 and up NA's
## 2011      518     11
```

5.2.2. Robust Method (tidyverse)

The tidyverse method follows similar logic. However, instead of explicitly overwriting 89 or older with the number 89 using indexing, the tidyverse solution uses the `readr` `parse_number()` function to remove the numbers from

each factor label. This works for the labels that already look numeric, like 18 . 00000 as well as for 89 or older. Then, conditional logic can be used to split the variable within a `mutate` command.

```
library(readr)
GSS <- GSS %>%
  mutate(tidyAge = parse_number(Age)) %>%
  mutate(tidyAge = if_else(tidyAge < 65, "18-65", "65 and up"),
        tidyAge = factor(tidyAge))
summary(GSS$tidyAge)

##   18-65 65 and up NA's
## 2011      518     11
```

Note that this approach requires the analyst to be very sure the strings including a number have a *relevant* number. If one of the levels was labeled 2 or more people in household it would be converted to the number 2. This would accidentally add a number that was not meaningful.

6. Creating Derived Categorical Variables

Challenges often arise when data scientists need to create derived categorical variables. As an example, consider an indicator of moderate drinking status. The National Institutes of Alcohol Abuse and Alcoholism have published guidelines for moderate drinking (NIAAA 2016). These guidelines state that women (or men aged 65 or older) should drink no more than one drink per day on average and no more than three drinks on any single day or at a sitting. Men under age 65 should drink no more than two drinks per day on average and no more than four drinks on any single day.

The HELPmiss dataset from the `mosaicData` package includes baseline data from randomized Health Evaluation and Linkage to Primary Care (HELP) clinical trial (Samet et al. 2003). These subjects for the study were recruited from a detoxification center, hence those that reported alcohol as their primary substance of abuse have extremely high rates of drinking.

Variable	Description
Sex	gender of subject female or male
i1	average number of drinks per day (in last 30 days)
i2	maximum number of drinks per day (in past 30 days)
Age	age (in years)

These guidelines can be used to create a new variable called `abstinent` for those reporting no drinking based on the value of their `i1` variable and `moderate` for those that do not exceed the NIAAA guidelines, with all other non-missing values coded as `highrisk`.

```
library(mosaic)
library(mosaicData)
library(dplyr)
library(readr)
```

Because missing values can become especially problematic in more complex derivations, we will make one value missing so we can ensure our data wrangling accounts for the missing value.

```

data(HELPmiss)
HELPsmall <- HELPmiss %>%
  mutate(i1 = ifelse(id == 1, NA, i1)) %>% # make one value missing
  select(-sex, -i1, -i2, -age)
head(HELPsmall, 2)

##   sex i1 i2 age
## 1 male NA 26 37
## 2 male 56 62 37

```

6.1. Fragile Method (base R)

```

# create empty vector for new variable
drinkstat <- character(length(HELPsmall$i1))
# create abstinent group
drinkstat[HELPsmall$i1 == 0] = "abstinent"
# create moderate group
drinkstat[(HELPsmall$i1 > 0 & HELPsmall$i1 <= 1) & # find those with moderate levels
           HELPsmall$i2 <= 3 & HELPsmall$sex == "female") |
  (HELPsmall$i1 > 0 & HELPsmall$i1 <= 2 &
   HELPsmall$i2 <= 4 & HELPsmall$sex == "male")] = "moderate"
# create highrisk group
drinkstat[((HELPsmall$i1 > 1 | HELPsmall$i2 > 3) & HELPsmall$sex == "female") |
  ((HELPsmall$i1 > 2 | HELPsmall$i2 > 4) & HELPsmall$sex == "male")] = "highrisk"
# account for missing values
is.na(drinkstat) <- is.na(HELPsmall$i1) | is.na(HELPsmall$i2) |
  is.na(HELPsmall$sex)
drinkstat <- factor(drinkstat)
table(drinkstat, useNA = "always")

## drinkstat
## abstinent  highrisk  moderate      <NA>
##       69        372       28          1

```

While this approach works, it is hard to follow, check, or debug. The logical conditions are all correctly coded, but require many repetitions of `HELPsmall$variable`, and the missing value was not handled by default (without the `is.na()` call, the missing value would default to be `highrisk` because of the extreme value for `i2` for that subject).

6.2. Robust Method (tidyverse)

```

HELPsmall <- HELPsmall %>%
  mutate(drink_stat = case_when(
    i1 == 0 ~ "abstinent",
    i1 <= 1 & i2 <= 3 & sex == 'female' ~ "moderate",
    i1 <= 1 & i2 <= 3 & sex == 'male' & age >= 65 ~ "moderate",
    i1 <= 2 & i2 <= 4 & sex == 'male' ~ "moderate",
    is.na(i1) ~ "missing", # can't put NA in place of "missing"
    TRUE ~ "highrisk"
  ))
  
HELPsmall %>%
  group_by(drink_stat) %>%
  dplyr::count()

## # A tibble: 4 x 2
## # Groups: drink_stat [4]
##   drink_stat     n
##   <chr> <int>
## 1 abstinent     69
## 2 highrisk      372
## 3 missing        1
## 4 moderate       28

```

In the robust tidyverse method, the same logic is used, but the conditions are clearer and more comprehensible. Instead of one complex Boolean condition for `moderate`, three separate lines can be used to match the different options. While the end result is the same, this code is more human readable and it is harder to miss special cases.

An additional example is provided in Supplementary Appendix B.

7. Defensive Coding

It is always good practice to code in a defensive manner. Investing a little time up front can help avoid painful errors later. In the context of wrangling categorical data, defensive coding involves running many `summary()` commands to ensure data operations do not mangle relationships, and might involve adding conditional testing statements into code creating or modifying factors. These testing statements (such as those implemented in the `testthat` and `assertthat` packages) can help ensure the data have not changed from one session to another, or as the result of changes to the raw data.

As an example, we might want to check there are exactly three levels for the drinking status variable in the `HELP` dataset. If there were fewer or more than three levels, something would have gone wrong with our code. The `assertthat` package can help with this.

```

library(assertthat)
levels(drinkstat)

## [1] "abstinent" "highrisk"  "moderate"

assert_that(length(levels(drinkstat)) == 3)

## [1] TRUE

```

We also might want to ensure the factor labels are exactly what we were expecting. Perhaps we want to make sure the `Sex` variable in the `GSS` data has two categories, with particular levels. The `expect_equivalent()` function from the `testthat` package can be used to make this check.

```

library(testthat)
levels(GSS$Sex)

## [1] "Female" "Male"

expect_equivalent(levels(GSS$Sex), c("Female", "Male"))

expect_setequal(levels(GSS$Sex), c("Male", "Female"))

```

This check will only work if the levels are exactly the same as the strings provided, and are in the same order. Ideally, `expect_equivalent()` would do level checking without relying on order, but that is not the current functionality.

```
expect_setequal(levels(GSS$Sex), c("Male", "Female"))
```

While assertions of this sort are most commonly used to provide error-checking within functions, we believe that they can and should be incorporated into working code. In this manner, they may serve as the basis for a function at some point in the future.

8. Conclusion

Categorical variables arise commonly in most datasets. Aspects of data wrangling in R involving categorical variables can be problematic and error-prone, particularly when using base R. In this article, we have outlined some example case studies where analytic tasks can be simplified and made more robust through use of new tools available in the tidyverse. However, these are only some of the issues categorical data presents.

For example, many analysts use testing and training data when working with models. Without careful thought toward levels of categorical variables, there can be a mismatch between the levels present in the training data and those present in the testing data. If a particular level was not present in the training data, the model will not be able to make predictions for the observations in the testing data with that level. Even worse, if the two sets have the same number of levels, the model may produce predictions by matching the order of the levels rather than the labels. Another possible issue can arise when indexing by a factor—the levels get treated as integers, rather than characters.

We believe further work is needed to continue to make it easier to undertake analyses requiring data wrangling (particularly with respect to categorical data). New tools and an increased emphasis on defensive coding may help improve the quality of data science moving forward.

Supplementary Materials

The online supplement contains the appendices for the article.

Acknowledgments

Thanks to Mine Çetinkaya-Rundel, Johanna Hardin, Zev Ross, Colin Rundel, Tam Tran The, and Hadley Wickham for helpful comments and suggestions on an earlier draft.

ORCID

Amelia McNamara  <http://orcid.org/0000-0003-4916-2433>
Nicholas J. Horton  <http://orcid.org/0000-0003-3332-4311>

References

Broman, K. (2015), “Initial Steps Toward Reproducible Research,” available at <http://kbroman.org/steps2rr/>. [97]

- FitzJohn, R., Pennell, M., Zanne, A., and Cornwell, W. (2014), “Reproducible Research is Still a Challenge,” Technical Report rOpenSci. [97]
- Hermans, F., and Murphy-Hill, E. (2015), “Enron’s Spreadsheets and Related Emails: A Dataset and Analysis,” *Proceedings of the 37th International Conference on Software Engineering*, 2, 7–16. [97]
- Leek, J. (2016), “How to Share Data with a Statistician,” <https://github.com/jtleek/datassharing>. [99]
- Lowndes, J. S., Best, B. D., Scarborough, C., Afflerbach, J. C., Frazier, M. R., O’Hara, C. C., Jiang, N., and Halpern, B. S. (2017), “Our Path to Better Science in Less Time using Open Data Science Tools,” *Nature Ecology & Evolution*, 1, 0160. [97]
- Lumley, T. (2015), “stringsAsFactors = <sigh>,” available at <http://notstatschat.tumblr.com/post/124987394001/stringsasfactors-sigh>. [97]
- NIAAA (2016), “Rethinking Drinking: What’s ‘Low-Risk’ Drinking?” available at <http://rethinkingdrinking.niaaa.nih.gov/How-much-is-too-much/Is-your-drinking-pattern-risky/Whats-Low-Risk-Drinking.aspx>. [102]
- NORC at the University of Chicago (2016), “GSS Data Explorer,” available at <https://gssdataexplorer.norc.org/>. [98]
- Peng, R. D. (2015), “stringsAsFactors: An Unauthorized Biography,” available at <http://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/>. [97,98]
- Ross, Z., Wickham, H., and Robinson, D. (2018), “Declutter your R Workflow with Tidy Tools,” *The American Statistician*, submitted. [97]
- Samet, J. H., Larson, M. J., Horton, N. J., Doyle, K., Winter, M., and Saitz, R. (2003), “Linking Alcohol and Drug-Dependent Adults to Primary Medical Care: A Randomized Controlled Trial of a Multidisciplinary Health Intervention in a Detoxification Unit,” *Addiction*, 98, 509–516. [102]
- Smith, T. W., Mardsen, P., Hout, M., and Kim, J. (2015), “General Social Surveys,” 1972–2014 [machine-readable data file]. Available at <https://gssdataexplorer.norc.org/> [98]
- Wickham, H. (2014), “Tidy Data,” *Journal of Statistical Software*, 59, 1–23. [97]
- (2016), “The Tidy Tools Manifesto,” available at <https://cran.r-project.org/web/packages/tidyverse/vignettes/manifesto.html>. [97]
- (2017), *forcats: Tools for Working with Categorical Variables (Factors)*, r package version 0.2.0. [101]
- Wickham, H., Hester, J., and Francois, R. (2017), *readr: Read Rectangular Text Data*, r package version 1.1.0. [98]
- Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., and Teal, T. K. (2016), “Good Enough Practices for Scientific Computing,” available at <https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/>. [97]