Mapping SAS Formats to R

A Comprehensive Guide for the SAS Programmer

Michael Walshe

# Introduction to SAS Formats

SAS formats and informats are a swiss-army knife of data manipulation, providing in one package methods to:

* Read formatted data as a different type, for example to read a value of £1,000.00 as a numeric value.
* Apply a ‘mask’ to data. This means that you can change the way that data is displayed, without changing the underlying data (e.g. a value of -0.6534 could print as (65.3%))
* Group data into bins, based on custom conditions and logic
* Perform a look-up from one value to another

Because of the sheer number of different use-cases for SAS formats, there is no single function or package in R which can fully replace it. However, this can be seen as a benefit, logically separating the different functionalities and making R easier to understand and use. In this article we’ll assess a few different methods for each of the above use-cases, starting with the simplest.

# Look-ups

A SAS format, typically a user-defined one, can be used as a look-up table - for example from a region code to the full name of that region. In R this could be achieved with:

* A join or merge
* A named vector
* Using a function such as dplyr::case\_when to encode the look-up

## Method 1: Using A Join

Joins are a widespread method for performing some form of look-up, we will demonstrate using the base R merge function.

my\_data <- data.frame(  
 region\_code = c("SA", "SA", "E", "AS", "AN", "NA"),  
 measure = runif(6)  
)  
  
lookup\_df <- data.frame(  
 region\_code = c("E", "NA", "SA", "AS", "AF", "AU", "AN"),  
 region\_full = c(  
 "Europe",  
 "North America",  
 "South America",  
 "Asia",  
 "Africa",  
 "Australia",  
 "Antartica"  
 )  
)  
  
merge(my\_data, lookup\_df)

#> region\_code measure region\_full  
#> 1 AN 0.5306974 Antartica  
#> 2 AS 0.9582330 Asia  
#> 3 E 0.6645159 Europe  
#> 4 NA 0.3405789 North America  
#> 5 SA 0.6323507 South America  
#> 6 SA 0.7420351 South America

## Method 2: Using A Named Vector

Where you have a simple mapping of keys to values, a named vector can be a good method in R to move from one to the other. Note here that the keys should be the *names* of the vector, and the values are the elements. Then indexing using a key produces the value.

lookup\_vector <- c(  
 "E" = "Europe",  
 "NA" = "North America",  
 "SA" = "South America",  
 "AS" = "Asia",  
 "AF" = "Africa",  
 "AU" = "Australia",  
 "AN" = "Antartica"  
)  
  
my\_data$region\_full <- lookup\_vector[my\_data$region\_code]

Note that if the keys were numbers, we would have to create the named vector in two steps, first creating a vector of values then assigning to the names function. However this is not recommended, as it can be unclear what e.g. lookup\_vector[3] should return.

## Method 3: Using [{dplyr}](https://dplyr.tidyverse.org/)

dplyr::case\_when() is a very useful way to create a new vector based on several conditions. It is similar to a case when expression in SQL. For a simple look-up, the conditions will be equality with our input vector. This also lets us map values not in our look-up table to some default.

my\_data$region\_full <- dplyr::case\_when(  
 my\_data$region\_code == "E" ~ "Europe",  
 my\_data$region\_code == "NA" ~ "North America",  
 my\_data$region\_code == "SA" ~ "South America",  
 my\_data$region\_code == "AS" ~ "Asia",  
 my\_data$region\_code == "AF" ~ "Africa",  
 my\_data$region\_code == "AU" ~ "Australia",  
 my\_data$region\_code == "AN" ~ "Antartica",  
 .default = "Unknown"  
)

This is a little clunky and includes some repetition, but we can use the more modern case\_match function to improve it, and at the same time move this into a function to let us easily re-use this mapping.

region\_lookup <- function(x) {  
 dplyr::case\_match(  
 x,  
 "E" ~ "Europe",  
 "NA" ~ "North America",  
 "SA" ~ "South America",  
 "AS" ~ "Asia",  
 "AF" ~ "Africa",  
 "AU" ~ "Australia",  
 "AN" ~ "Antartica",  
 .default = "Unknown"  
 )  
}  
  
region\_lookup(my\_data$region\_code)

#> [1] "South America" "South America" "Europe" "Asia"   
#> [5] "Antartica" "North America"

# Binning

For those unaware, “binning” is the process of grouping data in ranges of values. SAS user defined formats can be created to group numeric values into these bins. It’s used in a variety of situations, from analysis and visualisation to statistics. As such, there is a base R function cut that can provide most of the functionality we need.

The simplest behaviour is to provide the number of bins, R will create a factor with labels indicating the intervals.

ages <- c(59, 22, 20, 62, 30)  
  
cut(ages, breaks=3)

#> [1] (48,62] (20,34] (20,34] (48,62] (20,34]  
#> Levels: (20,34] (34,48] (48,62]

We can also define custom intervals and labels, note that labels needs to be 1 element shorter than breaks.

cut(  
 ages,  
 breaks=c(0, 40, 60, 100),  
 labels=c("Young", "Middle-Aged", "Old")  
)

#> [1] Middle-Aged Young Young Old Young   
#> Levels: Young Middle-Aged Old

There are many packages in R that provide a function to bin data for different use-cases, as one example we can again use dplyr::case\_when(), this time to group data. This produces a character vector rather than factor.

dplyr::case\_when(  
 ages <= 40 ~ "Young",  
 ages <= 60 ~ "Middle-Aged",  
 .default = "Old"  
)

#> [1] "Middle-Aged" "Young" "Young" "Old" "Young"

# Masks

Using a format as a “mask” for data is the most common use-case in SAS, keeping the underlying data the same but applying formatting when it is displayed. There are several different methods we can use to replicate this in R.

## Method 1: Base R

Using Base R, there are several different functions that can format data.

The first we’ll use is the format() function, which can format any R object for printing, but is typically used for numeric vectors and has a large set of options:

numbers <- runif(10, -1000000, 1000000)  
  
format(numbers, justify="right", width="10", big.mark=",") |> print()

#> [1] " 113,417.6" " -923,279.3" " 676,393.9" " 377,751.7" " 853,984.9"  
#> [6] " 910,208.7" " 687,678.4" " -194,466.7" " 709,365.0" " -220,054.1"

format(numbers, digits=3, scientific=TRUE) |> print()

#> [1] " 1.13e+05" "-9.23e+05" " 6.76e+05" " 3.78e+05" " 8.54e+05" " 9.10e+05"  
#> [7] " 6.88e+05" "-1.94e+05" " 7.09e+05" "-2.20e+05"

Alternatively, the sprintf function provides a familiar interface to those used to C style formatting, for more detail run ?sprintf.

sprintf("$%.2f", numbers)

#> [1] "$113417.57" "$-923279.33" "$676393.90" "$377751.70" "$853984.88"   
#> [6] "$910208.72" "$687678.41" "$-194466.74" "$709364.99" "$-220054.11"

## Method 2: [{scales}](https://scales.r-lib.org/)

The {scales} package is used by {ggplot2} to scale data for graphs, but it also includes some useful formatting functions that can help with common formats. For example:

scales::label\_dollar(prefix="£", style\_negative="parens")(numbers)

#> [1] "£113,418" "(£923,279)" "£676,394" "£377,752" "£853,985"   
#> [6] "£910,209" "£687,678" "(£194,467)" "£709,365" "(£220,054)"

Note that all the scale::label\_... functions return another function, and we’ve passed the numbers vector into this. There are label functions for currency, percents, dates, time, mathematics, SI units, p-values, and more! If a particular label is used frequently in your code, it may be helpful to assign it to a helper function

percent <- scales::label\_percent(  
 accuracy=0.1,  
 suffix=" %",  
 decimal.mark=",",  
 style\_negative="parens"  
)  
  
percent(runif(5, -1, 1))

#> [1] "98,2 %" "(51,4 %)" "46,9 %" "(47,3 %)" "44,8 %"

## Method 3: [{formatttable}](https://renkun-ken.github.io/formattable/)

All the examples covered so far have 1 key difference to SAS formats, they return the data converted to a new format, rather than the original data with a mask that applies when printing. For this, we can use {formattable}. This lets us apply formats to a vector or data.frame, and then continue to work with them as if they were standard R vectors. Only when it comes time to print does the format get applied.

For example:

values <- runif(5, -1, 1)  
  
percs <- formattable::percent(values)  
  
print(percs)

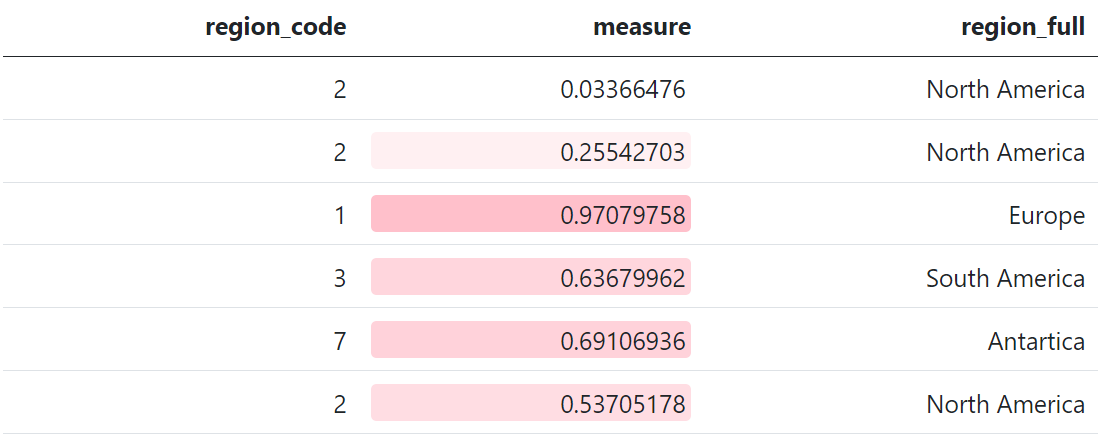
#> [1] 66.69% 29.35% 49.45% -26.66% -91.56%

print(percs \*\* 2)

#> [1] 44.47% 8.61% 24.46% 7.11% 83.83%

This is not limited to standard numeric formats, for which {formattable} includes methods for: digits, commas, percents, accounting, currency, and scientific notation. There are also advanced methods for styling the output table, for example conditional formatting similar to Excel output:

formattable::formattable(  
 my\_data,  
 list(measure=formattable::color\_tile("white", "pink"))  
)



Note however that these formats will not be persisted unless the data is saved in an Rdata or RDS format.

# Reading in Data

The final use of a SAS format that we’d like to replicate is reading in character data as numeric. The simplest way to do this is using parse\_number from [{readr}](https://readr.tidyverse.org/), which provides a method to ignore non-numeric data in a column and attempt to parse it as a number:

readr::parse\_number(c("£1,000.00"))

#> [1] 1000

And for date formats, {readr} also provides readr::parse\_date or readr::parse\_datetime.

However, when it comes to more complex formats (e.g. “(82.5%)”) this will not take into account that it is a percentage and negative, and so more custom methods must be used:

values <- c("(82.5%)", "2%", "28.55%")  
  
my\_format <- function(x){  
 x |>   
 stringr::str\_replace\_all(c("\\(" = "-", "\\)" = "")) |>   
 readr::parse\_number() / 100  
}  
  
my\_format(values)

#> [1] -0.8250 0.0200 0.2855

# Honourable Mentions

* An incredibly useful package for formatting tables for output is [{gt}](https://gt.rstudio.com/index.html). This also includes a large number of formatting functions; however these can only be used at the point of printing the data and you cannot extract the formatted data, therefore this was not featured.
* An excellent package that provides a very large number of built-in formats is [{numform}](https://cran.r-project.org/web/packages/numform/index.html). This had the most included formats of all packages considered, however due to its relatively low usage (~1.2K downloads per month) the {scales} package was featured instead.
* Another package to mention is [{fmtr}](https://fmtr.r-sassy.org/articles/fmtr.html), which is designed to try and replicate the experience of SAS user-defined formats in R, including conditions and format catalogues. However, it also has limited popularity, and further it’s best practice to try to write idiomatic R rather than replicate SAS exactly.
* The {formattable} package has unreleased functions for parsing different formats from text, for example parse\_accounting. On release, these may be a good method to read in formatted data.