Data frame operations

Robert McDonald

Contents

1 Basic operations on dataframes

When you analyze data in R, you will almost certainly use dataframes, which are rectangular objects that can store multiple types of data at once, subject to the restriction that any given column only contains data of one type (numeric, character, logical, etc.). When performing your analysis, you will undertake basic operations, which can include extracting portions of the dataframe, subsetting the data frame, deleting rows and columns, adding rows and columns, sorting, renaming, summarizing, etc.

There are generally multiple ways to undertake a given dataframe manipulation in base R. The variety can be confusing, but to make things more complicated, Hadley Wickham has introduced tibbles as an an alternative to dataframes. The bad news is that tibbles behave differently than dataframes in certain respects. However, this is also the good news, because tibbles standardize and simplify behavior. In order to accomplish this, tibbles by definition have to behave differently than dataframes.

In this document I am going to recommend that you use tibbles. Although they are sometimes annoying, they behave more consistently than data frames and there is less to remember.

Two terms that we will use frequently require careful defintion. Here is what we mean by "subsetting" a data frame and and "extraction" from a data frame:

- **subsetting** means creating a new object that has fewer rows or columns, but where the new object has the same class as the old object. A dataframe remains a dataframe, for example.
- **extraction** means creating a *vector* that has a class appropriate to the vector contents (numeric, character, logical, etc.) By definition, if you extract a column from a dataframe or tibble, it is no longer a dataframe or a tibble; it is a vector.

It will be critical to distinguish these two concepts. We will see that an important contribution of tibbles is that subsetting operators only subset, they never extract. This is not true with dataframes and is an important reason to prefer tibbles.

The purpose of this document is to illustrate the different ways of performing various operations, and to illustrate differences between dataframes and tibbles. Even if you prefer one construct and stick to it, the odds are that you will encounter examples that work differently and co-workers who have different preferences. You need to understand both dataframes and tibbles. The good news is that in understanding tibbles, you will come to understand dataframes more deeply.

2 Dataframes vs tibbles

2.1 Creating dataframes and tibbles

We begin by defining two data frames, named `df0` and `df`:[1]

```
df0 <- data.frame(what=c('crust', 'cookie', 'cake', 'pasta'),
                  flour=c(3, 3, 1, 3),
                  butter=c(2, 2, 1, 0),
                  liquid=c(1, 0, 0, 0),
                  sugar=c(0, 1, 1, 0),
                  egg=c(0, 0, 1, 2)
                  )

df <- data.frame(what=c('crust', 'cookie', 'cake', 'pasta'),
                 flour=c(3, 3, 1, 3),
                 butter=c(2, 2, 1, 0),
                 liquid=c(1, 0, 0, 0),
                 sugar=c(0, 1, 1, 0),
                 egg=c(0, 0, 1, 2),
                 stringsAsFactors=FALSE
                 )
all.equal(df0, df, check.attributes=FALSE)
[1] "Component \"what\": 'current' is not a factor"
```

By default, the `data.frame` function converts strings to factors. To change this behavior, it is common to set `stringsAsFactors=FALSE`, as when creating `df`. In the rest of this document we will assume that we do *not* want strings automatically converted to factors.

We now create a tibble, `dft`, using the same inputs. The `tibble` function does not convert strings to factors:

```
dft <- tibble(what=c('crust', 'cookie', 'cake', 'pasta'),
              flour=c(3, 3, 1, 3),
              butter=c(2, 2, 1, 0),
              liquid=c(1, 0, 0, 0),
              sugar=c(0, 1, 1, 0),
              egg=c(0, 0, 1, 2)
              )
all.equal(df, dft, check.attributes=FALSE)
[1] TRUE
```

Although `df` and `dft` contain the same data with the same data types, the `all.equal` function will complain that `df` and `dft` have different classes unless we specify `check.attributes=FALSE`.

The data frame looks like this:

[1] This example was inspired by Ruhlman (2009). Ratios are by weight. Any errors related to recipes baking details are mine.

```
kable(df)
```

what	flour	butter	liquid	sugar	egg
crust	3	2	1	0	0
cookie	3	2	0	1	0
cake	1	1	0	1	1
pasta	3	0	0	0	2

The tibble looks like this:

```
kable(dft)
```

what	flour	butter	liquid	sugar	egg
crust	3	2	1	0	0
cookie	3	2	0	1	0
cake	1	1	0	1	1
pasta	3	0	0	0	2

When dealing with tibbles it can be helpful to customize printing options, i.e the way the data appears when you type the name of the tibble in the console. You can get a list of available options with `?tibble::tibble-options`. I like the following settings, but you can do something different and you can change them at any time by issuing another `options` command.

```
options(digits=2 ## how many digits to print?
        ,tibble.width=Inf ## how many tibble columns to print?
        ,tibble.print_min=10  ## 10 is the default
        ,pillar.subtle=FALSE  ## don't highlight significant digits
        )
```

2.2 Operators

When dealing with dataframes and tibbles there are three data subsetting and extraction constructs we need to understand:

- "`[`" Single square brackets are used to *subset*. (There is an important exception to this with dataframes, for which subsetting to a single column also extracts. This does not happen with tibbles.)
- "`[[`" Double square brackets *extract* a single column at a time
- "`$`" The dollar sign also *extracts*, serving the same purpose as double square brackets

2.3 Extract one column from a dataframe

Suppose we want to extract the flour column:

```
c1 <- df[, 'flour']
c2 <- df$flour
c3 <- df[['flour']]
c4 <- df['flour'][[1]]
class(c1)  ## this is a vector
[1] "numeric"
```

```
all.equal(c1, c2)
[1] TRUE
all.equal(c2, c3)
[1] TRUE
all.equal(c3, c4)
[1] TRUE
mean(c1)
[1] 2.5
mean(c2)
[1] 2.5
mean(c3)
[1] 2.5
mean(c4)
[1] 2.5
```

Note that in every case we have a numeric vector, and we can conpute the mean of this vector.

2.4 Extract one column from a tibble

```
c1t <- dft[, 'flour']  ## This subsets but does not extract!
c2t <- dft$flour
c3t <- dft[['flour']]
c4t <- dft['flour'][[1]]
class(c1)  ## this is a tibble, not a vector
[1] "numeric"
all.equal(c1t, c2t)
[1] "Cols in y but not x: `c(3, 3, 1, 3)`. "
[2] "Cols in x but not y: `flour`. "
all.equal(c2t, c3t)
[1] TRUE
all.equal(c3t, c4t)
[1] TRUE
mean(c1t)
Warning in mean.default(c1t): argument is not numeric or logical: returning
NA
[1] NA
mean(unlist(c1t))
[1] 2.5
mean(c2t)
[1] 2.5
mean(c3t)
[1] 2.5
mean(c4t)
[1] 2.5
```

Here we see an **important** difference between a dataframe and a tibble. If we extract the flour column from the tibble using the single bracket, the single column remains a tibble. If we extract the flour column from the data frame using the same syntax, the single column is converted to a vector.

Contrast this with extracting the flour column from either `df` or `dft` using double brackets or the `$` synonym, which creates a numeric vector. The tibble behavior is, in my opinion, appropriate, because the single bracket is a subsetting operator, not an extraction operator. You might ask why we care. The issue in this case is that the `mean` function cannot operate directly on either a dataframe or

tibble column. It can only operate on a vector. Using `unlist` permits us to calculate the mean of `c1t`.

By contrast with the tibble, you extract the column from the dataframe when using the subset operator, "`[`". This might seem innocuous, but it causes confusion, because if you specify *two* columns, you create a subset without extracting. So the behavior of "`[`" depends on how you use it. In my experience this causes confusion.

This behavior of tibbles may seem annoying if you are acccustomed to the behavior of data frames, but it creates consistency: single brackets are for *subsetting*, not for extraction. Tibbles maintain this consistency, dataframes do not.

Another source of confusion is that that single brackets behave differently with a dataframe depending upon whether or not there is a row indicator in the expression:

```
c1 <- df[, 'flour'] ## this is a vector of length 4
c5 <- df['flour']   ## this is a dataframe (list) of length 1
all.equal(c1, c5) ## c1 and c5 differ
[1] "Modes: numeric, list"
[2] "Lengths: 4, 1"
[3] "names for current but not for target"
[4] "Attributes: < target is NULL, current is list >"
[5] "target is numeric, current is data.frame"
c1t <- dft[, 'flour']
c5t <- dft['flour']
all.equal(c1t, c5t)  ## c1t and c5t are the same
[1] TRUE
c6 <- select(df, flour)
class(c5)  ## this is a dataframe
[1] "data.frame"
all.equal(c5, c6)
[1] TRUE
all.equal(c5, c5t)
[1] "Attributes: < Component \"class\": Lengths (1, 3) differ (string compare on first 1) >"
[2] "Attributes: < Component \"class\": 1 string mismatch >"
all.equal(c1, c5)  ## a vector and dataframe are not the same!
[1] "Modes: numeric, list"
[2] "Lengths: 4, 1"
[3] "names for current but not for target"
[4] "Attributes: < target is NULL, current is list >"
[5] "target is numeric, current is data.frame"
```

2.5 Subsetting vs extraction

The important thing to remember about extraction is that *you can't extract multiple columns at once*. The reason is that you can't know in advance the class of the extracted columns and they may be inconsistent. If column 1 is numeric and column 2 is character, there is no common data structure which you can extract them, other than a list, for example a dataframe or tibble.

Of course you *can* subset multiple columns at once, because you are retaining the data frame or tibble from which subset.

2.6 Add a column

We will add "nuts" and "yeast" columns:

```
df1 <- df2 <- df3 <- df  ## create copies of the original data frame
df1$nuts <- c(0, 0.5, 0, 0)
df1$yeast <- 0  ## the recycling rule in action
df2[, "nuts"] <-  c(0, 0.5, 0, 0)
df2[, "yeast"] <- 0
df3 <- df %>% mutate(nuts=c(0, 0.5, 0, 0), yeast=0)
all.equal(df1, df2)
[1] TRUE
all.equal(df2, df3)
[1] TRUE
kable(df1)
```

what	flour	butter	liquid	sugar	egg	nuts	yeast
crust	3	2	1	0	0	0.0	0
cookie	3	2	0	1	0	0.5	0
cake	1	1	0	1	1	0.0	0
pasta	3	0	0	0	2	0.0	0

Very important: Notice that when using `dplyr` to construct `df3`, no quotes are needed. When using base R, quotes are needed for column names. The absence of quotation within the `dplyr` universe is great, but it can be very tricky (and frustrating) to mix `dplyr` and base R. My advice is to stick with one or the other as much as possible.

2.7 Delete columns

Now we decide we don't need the columns we just added:

```
df4 <- df1;
df4[c('nuts', 'yeast')] <- NULL  ## assigning to `NULL` deletes an object
df5 <- df1[-c(7, 8)]
df6 <- df1[-which(names(df1) %in% c('nuts', 'yeast'))]
df7 <- df1[, -which(names(df1) %in% c('nuts', 'yeast'))]
df8 <- df1 %>% select(-nuts, -yeast)
all.equal(df4, df5)
[1] TRUE
all.equal(df4, df6)
[1] TRUE
all.equal(df4, df7)
[1] TRUE
all.equal(df4, df8)
[1] TRUE
```

2.8 Add rows

We will use the dataframe including nut and yeast columns. Notice that we get the same result whether we define the new row as a `list` or as a `data.frame`. This occurs because a data.frame is a list!

```
newrow1 <- list('bread', 1, 0, .67, 0, 0, 0, .02)
newrow2 <- list(what='bread', flour=1, butter=0,
                liquid=.67, sugar=0, egg=0, nuts=0, yeast=.02)
```

```
newrow3 <- data.frame(what='bread', flour=1, butter=0,
                      liquid=.67, sugar=0, egg=0, nuts=0, yeast=.02)
df.ar1 <- rbind(df1, newrow1)
df.ar2 <- rbind(df1, newrow2)
df.ar3 <- bind_rows(df1, newrow2) ## dplyr
df.ar4 <- bind_rows(df1, newrow3)
Warning in bind_rows_(x, .id): binding character and factor vector,
coercing into character vector
all.equal(df.ar1, df.ar2)
[1] TRUE
all.equal(df.ar1, df.ar3)
[1] TRUE
all.equal(df.ar1, df.ar4)
[1] TRUE
```

2.9 Filter rows

We can choose to keep or delete rows that meet specific criteria. We will use `df.ar1` as the base dataframe.

Suppose we want only items that use butter

```
df.butter1 <- df.ar1[df.ar1$butter > 0, ]
df.butter2 <- df.ar1[df.ar1$"butter" > 0, ]
df.butter3 <- subset(df.ar1, df.ar1$butter > 0)
df.butter4 <- filter(df.ar1, butter > 0)
all.equal(df.butter1, df.butter2)
[1] TRUE
all.equal(df.butter1, df.butter3)
[1] TRUE
all.equal(df.butter1, df.butter4)
[1] TRUE
```

Suppose we want items that use butter and liquid. Any expression evaluating to a logical will work, so compound conditions will work. When using dplyr we can separate compound conditions either with a comma or by using an ampersand:

```
df.butterliquid1 <- df.ar1[df.ar1$butter & df.ar1$liquid > 0, ]
df.butterliquid2 <- filter(df.ar1, butter > 0, liquid > 0)
df.butterliquid3 <- filter(df.ar1, butter > 0 & liquid > 0)
all.equal(df.butterliquid1, df.butterliquid2)
[1] TRUE
all.equal(df.butterliquid1, df.butterliquid3)
[1] TRUE
```

References

Ruhlman, Michael. 2009. *Ratio: The Simple Codes Behind the Craft of Everyday Cooking.* Scribner.