Dataframe operations

Robert McDonald

2021-12-14

Contents

1 Basic operations on dataframes

Data in R commonly resides in one or more dataframes, which are rectangular data containers, like a spreadsheet worksheet. A given dataframe 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 analysis, you undertake basic data operations, which can include extracting subsets of the data frame, deleting rows and columns, adding rows and columns, sorting, renaming, summarizing, etc.

There are different methods you can use to accomplish any given manipulation of a dataframe. To complicate things further, there are three different versions of a dataframe:

- classic dataframes
- tibbles, which are the basic data structure of the `tidyverse`
- datatables, which are the basic data structure ot the `data.table` package.

The purpose this note is to discuss subsetting and data extraction for the three structures, and to illustrate several cases where the apparent same operation generates different results. The focus will be on dataframes and tibbles, but I will discuss datatables where it is relevant.

Much of this document discusses the closely related concepts of *subsetting* a data frame, and *extracting* from a data frame. Here is what we mean by these terms:

- **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. If you eliminate columns in a dataframe to create a new dataframe, you have created a subset, as the object remains a dataframe. You subset with single brackets (`[ ]`) or by using a function such as `subset`, which is a base function, or one of the `dplyr` functions such as `filter` or `select`.
- **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. You extract with double brackets (`[[ ]]`) or by using a function such as `unlist`.

It will be critical to distinguish the two concepts of subsetting and extraction. We will see that an important characteristic of tibbles is that *subsetting operators always subset*, they never extract. This is not true with dataframes. This consistency is a reason you might prefer tibbles.

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 to which you can extract them, other than the dataframe or tibble in which they reside, or a list.[1]

You can *subset* multiple columns at once, because you are retaining the data structure (dataframe or tibble) from which you create the subset.

You may find that you have a preferred way of performing the various operations. Even if you prefer one construct and stick to it, 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 and the various methods of working with them. The good news is that by working to understand subsetting and extraction operators and functions, you will significantly deepen your understanding of R in general and dataframes in particular.

2 Dataframes vs tibbles

2.1 Creating dataframes

We begin by defining two data frames, named `df0` and `df`, which contain recipe information, with ingredients listed by relative weight:[2]

```
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),
```

[1] Dataframes and tibbles are both lists, which is why they can contain columns of different vector classes.

[2] This example was inspired by Ruhlman (2009). You can read the recipe for pie crust, for example, as "3 parts flour to 2 parts butter to one part liquid", which could mean 150 grams of flour, 100 grams of butter, and 50 grams of liquid. Any errors are definitely mine!

```
                  stringsAsFactors=FALSE
                  )
all.equal(df0, df, check.attributes=FALSE)
[1] TRUE
```

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.

2.2 Creating tibbles

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 printed data frame looks like this:

```
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 printed tibble looks identical to the printed data frame:

```
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 the options that govern the way a tibble appears when you print it 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? Not specific to tibbles
        ,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
        )
```

2.3 Creating datatables

A datatable closely resembles a data frame:

```
dfdt <- data.table(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, dfdt, check.attributes=FALSE)
[1] TRUE
```

3 Subsetting and extraction 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. There is also)
- "`[[`" Double square brackets *extract* a single column at a time
- "`$`" The dollar sign also *extracts*, serving the same purpose as double square brackets

3.1 Extracting a column

A common operation with data frames is to create a subset by extracting columns. We will illustrate ways to do this with both dataframes and tibbles.

3.1.1 Extract one column from a dataframe

Suppose we want to extract the flour column from the dataframe `df`:

```
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 compute the mean of this vector.

It is worth discussing in more detail the operation that produced `c4`. The expression `df['flour']` subsets the dataframe to one column. The expression `[[1]]` then extracts that one column to a vector. In general, `[[1]]` appended to an expression extracts the first element. If there is more than one element, as with a dataframe, you can extract the fifth element by appending `[[5]]`.

Your subset can include multiple columns or rows, but you can only extract one element at a time. For example, `df[c('flour', 'butter', 'liquid')]` is legitimate, and `df[c('flour', 'butter', 'liquid')][[3]]` will extract the third column, `liquid`. If you write `df[c('flour', 'butter','liquid')][[c(1, 3)]]`, you will extract the third element of the first column, which is 1.

3.1.2 Extract one column from a tibble

We repeat the previous syntax, only operating on a tibble rather than a dataframe.

```
c1t <- dft[, 'flour']  ## This subsets but does not extract!
c2t <- dft$flour
c3t <- dft[['flour']]
c4t <- dft['flour'][[1]]
class(c1t)  ## this is a tibble, not a vector
[1] "tbl_df"     "tbl"        "data.frame"
all.equal(c1t, c2t)
[1] "Modes: list, numeric"
[2] "names for target but not for current"
[3] "Attributes: < Modes: list, NULL >"
[4] "Attributes: < names for target but not for current >"
[5] "Attributes: < current is not list-like >"
[6] "Length mismatch: comparison on first 1 components"
[7] "Component 1: Numeric: lengths (4, 1) differ"
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(c2t)
[1] 2.5
mean(c3t)
[1] 2.5
mean(c4t)
[1] 2.5
```

It turns out the `mean` function expects a numeric vector as input. Because `c1t` is a tibble, we receive an error message. If you wish to compute the mean of a tibble column, you have several alternatives:

- convert it to a numeric vector using `unlist`, `[[1]]`, or the `dplyr pull` function
- use the colMeans function, which works on dataframes
- use dplyr's summarize function

```
ct1 <- dft[, 'flour']
mean(ct1[[1]])
[1] 2.5
unlist(ct1)
flour1 flour2 flour3 flour4
     3      3      1      3
mean(unlist(ct1))
[1] 2.5
mean(pull(ct1))
[1] 2.5
colMeans(ct1)
flour
  2.5
ct1a <- dft %>% select(flour) %>% summarize(mean=mean(flour))
ct1a
# A tibble: 1 x 1
   mean
  <dbl>
1   2.5
ct1a[[1]]
[1] 2.5
```

Notice that `ct1a` is a tibble! The dplyr functions return a tibble, even when you have just computed a single number. We can extract the result using `[[1]]` or one of the other methods above.

Notice also that the `unlist` function creates a named vector. You can prevent the creation of names by specifying an option: `unlist(test1, use.names=FALSE)`. I find `[[1]]` or the `pull` function to be cleaner solutions.

3.1.3 Extract one column from a datatable‘

Repeat as before:

```
c1dt <- dfdt[, 'flour']  ## This subsets but does not extract!
c2dt <- dfdt$flour
c3dt <- dfdt[['flour']]
## c4dt <- dfdt['flour'][[1]] ## this fails
class(c1dt)  ## this is a tibble, not a vector
[1] "data.table" "data.frame"
all.equal(c1dt, c2dt)
[1] "target is data.table, current is numeric"
all.equal(c2dt, c3dt)
[1] TRUE
mean(c1dt)
Warning in mean.default(c1dt): argument is not numeric or logical: returning NA
[1] NA
mean(c2dt)
[1] 2.5
mean(c3dt)
[1] 2.5
```

It is important to understand why the fourth line fails. Consider these expressions. I will not execute them, but you should try:

```
df[2]  ## returns the `flour` column
dfdt[2] ## returns the second row, not the second column!
```

```
dfdt['flour']  ## fails -- there is no row named 'flour'
```

A datatable column cannot be subset using single brackets. Essentially a datatable is more like a matrix. This is an important difference that can be hard to debug if you're not expecting this behavior.

3.1.4 Subsetting tibbles produces tibbles

The tibble examples illustrate that subsetting and summarizing operations on tibbles produce tibbles. 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 dataframe 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, either of 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. The tibble philosophy is that if you use a subset operator on a tibble, you get a tibble.

3.1.5 Subsetting dataframes may or may not produce dataframes

Consider these examples:

```
onecol <- df[, c('flour')]  ## this is a vector
twocol <- df[, c('flour', 'liquid')]  ## this is a dataframe
```

You can see that you get a different structure when you extract one or two columns from a dataframe. Extracting a single column gives you a numeric vector, while extracting two columns gives you a dataframnote.

Further, single brackets behave differently with dataframes 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"
all.equal(c1, c5[[1]]) ## the same
[1] TRUE
```

The object `c1` is a vector. The object `c5` is a subset, a new dataframe consisting of the `flour` column from the original dataframe. When you perform operations on a `tibble`, you get a new `tibble`. If you want a vector, you have to ask for a vector by using the extraction function `[[1]]` (which I recommend) or by using the dplyr `pull` function‘.

3.1.6 Subsetting differs with a datatable

The datatable behavior is superficially similar to the behavior of dataframes and tibbles, but it is different. In fact, the datatable is a structure with a rich syntax that permits you to sort, extract, join, create computed columns, etc, all using a dataframe-like syntax. The `data.table` package exists because it is dramatically faster than traditional and `dplyr` operations. There are two things to know about this:

1. You can use the datatable syntax in lieu of traditional dataframe and `dplyr` operations
2. You can use the `dtplyr` package and get most of the benefits of datatable's speed using the `dplyr` syntax. Be aware that this is a work in progress.

3.2 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 a helpful simplification, but it is easy to become confused if you mix `dplyr` and base R. My advice is to stick with one or the other as much as possible.

3.3 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
```

3.4 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)
all.equal(df.ar1, df.ar2)
[1] TRUE
all.equal(df.ar1, df.ar3)
[1] TRUE
all.equal(df.ar1, df.ar4)
[1] TRUE
```

3.5 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.