Selecting Cases/Variables, Creating Variables in R

Data Modification with Data Frames. APSY510/511 Class Usage

Bruce Dudek

2020-07-23

Contents

Preface

This document is intended to be used by students in a graduate level introductory statistics sequence of courses. The primary intended audience has some basic familiarity with the R ecosystem and has used several data analytic functions. This audience also has had some exposure to SPSS methods for the analogous set of functions/procedures outlined here and the exposition makes reference, at times, to these SPSS approaches. Nonetheless, the document is written in such a way that a more general audience can find the document useful. It will be most helpful for researchers and students who are in early phases of a learning curve for the R language. It takes a functional approach rather than a programming approach and can be viewed as an applied document.

1 Introduction

Researchers who confront the prospect of learning to use R as a data analytic tool quickly realize the need to understand the three major domains of usage: 1. Data Import/Defnition and Modification (also called Data Wrangling in the Data Science world) 2. Data Visualization - both exploratory and presentation 3. Data Analysis - description, inference, and modeling

Of these, the first domain (especially data modification) may be the most uncomfortable, especially for those trained in commercial packages such as SPSS. This document focuses on some simple operations that are used with great frequency by virtually all researchers. The intent is NOT to give a comprehensive treatment of all Data Wrangling strategies, but rather a few basic ones that are likely to be confronted early in every research project's data analysis.

Data frames in R are structured in an arrangement where rows are cases and columns are variables. Often, we need to sort the rows according to values of one or more variables. Users of popular commercial software find this operation simple (e.g., with mouse/menuing operations in SPSS or writing code/syntax in SPSS or SAS) to implement with the sorting cases function. Similarly, we often need to select a subset of cases (analogous to the select cases procedure in SPSS) or a subset of variables. In R, several approaches to these methods are possible. Two general strategies are outlined here. One is the use of base system functions and the other is the tidyverse approach.

In order to make this document a bit more contextually full, I also develop strategies for creation of new variables (both numeric and logicals).

1.1 First, load the necessary packages

```
library(psych,quietly=TRUE, warn.conflicts=FALSE)
library(gt,quietly=TRUE, warn.conflicts=FALSE)
library(knitr,quietly=TRUE, warn.conflicts=FALSE)
library(car,quietly=TRUE, warn.conflicts=FALSE)
library(plyr,quietly=TRUE, warn.conflicts=FALSE)
library(dplyr,quietly=TRUE, warn.conflicts=FALSE)
library(tidyr,quietly=TRUE, warn.conflicts=FALSE)
```

When functions from an add-on package are used, I try to use the format: `pkg::functionname` so that the origin of the function is known. This also helps when some function names are found in more than one package.

1.2 The data set

Let's use the `mtcars` data set available in R. Looking at the structure and the first few rows gives us a sense of the data set. Notice that all variables are numeric. The "am" variable is transmission type, automatic or manual, coded 1 and zero respectively. This should actually be seen as a factor, and later we will change it to a factor and apply value labels. Exact specification of what the variables represent can be found by doing a help on the data frame: `?mtcars`.

```
data(mtcars)
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```r
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
#gt::gt(psych::headTail(mtcars))
```

Notice that the table above appears to have the car model as a row name, rather than as a variable. And if we use the `gt` function to display the data frame (or part of it using the `headTail` function) as a nicer table, then we don't see those row names. This may or may not be what we want. It is possible to include the row names (car makes) as a variable and we will see how to do that in a section below where we learn to create new variables.

In addition, the reader will notice that this document variously uses `gt` or `kable` to obtain nicer formatting of tables and data frame listing. The choice is usually based on whether I wished to show the row names (vehicle makes) as part of the table (with `kable`) or not (with `gt`)

```
data(mtcars)
gt::gt(psych::headTail(mtcars))
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.46	0	1	4	4
21	6	160	110	3.9	2.88	17.02	0	1	4	4
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
21.4	6	258	110	3.08	3.21	19.44	1	0	3	1
...	...	...	...	...	...	...	...	...	...	...
15.8	8	351	264	4.22	3.17	14.5	0	1	5	4
19.7	6	145	175	3.62	2.77	15.5	0	1	5	6
15	8	301	335	3.54	3.57	14.6	0	1	5	8
21.4	4	121	109	4.11	2.78	18.6	1	1	4	2

2 Data Frame Operations using base system funtions

All of the operations we consider in this document can be done with base system functions. In a later section we will also consider how to do the same things with **tidyverse** approaches

2.1 Selecting variables or cases - subsetting using the `[ ]` notation

Two primary approaches to subsetting are available with base system functions. A notation system using brackets, `[ ]` is capable of handling the rows or column selection often required when working with data frames. However, it is much more broadly capable of working with atomic vectors, matrices, lists and arrays. The second approach below, using the `subset` function, may be a simpler approach when working with data frames - as is the case with the most common analyses we encounter.

A brief introduction to the logical structure of this notation is helpful at this point. A pair of brackets (e.g., `[ , ]`) is most often associated with a data frame name. So, when we see something like `mtcars[x,y]`, the x refers to something related to rows of the data frame and the y refers to something about columns. An series of illustrations can help. First, see that `mtcars[2:4,]` returns the 2nd through fourth rows of the data frame. The fact that nothing is specified after the comma implies that all columns will be returned.

```
mtcars[2:4,]
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

Similarly, `mtcars[,1:4]` would return all rows but only the first through fourth columns. Since this would be N rows long, I suppressed the output.

```
mtcars[,1:4]
```

These operations, using the bracketing notation are called subsetting. We can simultaneously subset the data frame for both rows and columns. Note that here, I selected non-adjacent rows and non-adjacent columns.

```
mtcars[c(1,4,20:24),c(1:3,5,9)]
```

```
##                   mpg cyl  disp drat am
## Mazda RX4        21.0   6 160.0 3.90  1
## Hornet 4 Drive   21.4   6 258.0 3.08  0
## Toyota Corolla   33.9   4  71.1 4.22  1
## Toyota Corona    21.5   4 120.1 3.70  0
## Dodge Challenger 15.5   8 318.0 2.76  0
## AMC Javelin      15.2   8 304.0 3.15  0
## Camaro Z28       13.3   8 350.0 3.73  0
```

2.1.1 Select only a few of the Variables

In order to simplify our considerations, let's subset this full data frame, keeping only a few of the variables so that our orderings are more visible. Note that "am" is a code for automatic vs manual transmission. Notice that standard R practice would be to create a new object for this subsetted data frame. The word subset is used in R in lieu of "select", at least in base systems operations.

```
data1 <- mtcars[,c(1:4,6:7,9)]
gt::gt(psych::headTail(data1))
```

mpg	cyl	disp	hp	wt	qsec	am
21	6	160	110	2.62	16.46	1
21	6	160	110	2.88	17.02	1
22.8	4	108	93	2.32	18.61	1
21.4	6	258	110	3.21	19.44	0
...	...	...	...	...	...	...
15.8	8	351	264	3.17	14.5	1
19.7	6	145	175	2.77	15.5	1
15	8	301	335	3.57	14.6	1
21.4	4	121	109	2.78	18.6	1

2.1.2 Select a subset of cases

And to further simplify, for some initial illustrations, lets also subset only those cars that have 6 cylinder engines. This is effectively a "select cases" operation.

But first lets count the number of cases for each value of cylinder.

```
knitr::kable(table(data1$cyl))
```

Var1	Freq
4	11
6	7
8	14

Now we know to expect seven cases from our row subsetting operation. The several sorting operations carried out below all begin with this data frame (data2). Here, we implement the selection by using the `which` function, nested inside the brackets. This approach provides an efficient way to conditionalize the output. The user needs to have some familiarity with the concept of using two equal signs here. The double equal sign `==` is a relational operator (used analogously to `>` or `<`, or `<=`, for example). The double equal sign is

needed to differentiate this conditional/relational operation from the situation where a single equal sign (=) means assignment (equivalent to <-) and is also used to specify the value of an argument passed to a function (e.g., `dbinom(x=4, size=25, prob=.5)`). It helps to understand the double equal sign as meaning "exactly equal to."

```
data2 <- data1[which(data1$cyl==6),]
gt::gt(data2)
```

mpg	cyl	disp	hp	wt	qsec	am
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1
21.4	6	258.0	110	3.215	19.44	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0
19.7	6	145.0	175	2.770	15.50	1

2.1.3 Select both subsets of cases and variables

We saw, above, how to select a subset of both cases and variables by using their numeric position. (code duplicated here without printing the result)

```
mtcars[c(1,4,20:24),c(1:3,5,9)]
```

What if we want to conditionally subset the cases on a value of a variable (the 6 cylinder cars, for example) and also select a subset of variables? With the following code, an additional way of subsetting the variables is exemplified with a method that permits using the variable names rather than numeric positions.

```
databoth <- mtcars[which(mtcars$cyl==6),names(mtcars) %in% c("mpg","cyl","disp", "hp", "wt", "qsec", "a
gt::gt(databoth)
```

mpg	cyl	disp	hp	wt	qsec	am
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1
21.4	6	258.0	110	3.215	19.44	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0
19.7	6	145.0	175	2.770	15.50	1

This approach, using the '%in%' operator effectively requests a subset of variables by choosing them from the list of "names" of variables associated with the data frame.

2.2 Selecting variables or cases using `subset`

Although the bracketing notation system, once learned, is rapid and effective, the `subset` function from **base** may be easier for the novice to master - in the limited situaion of working with data frames.

This section duplicates the selections outlined above where new data frames "data1", and "data2" were produced. To parallel that notation, I will call them "data1b" and "data2b" here.

2.2.1 Select only a few of the Variables using `subset`

In order to simplify our considerations, let's subset this full data frame, keeping only a few of the variables so that our orderings are more visible. The same set of variables are selected as above. Note the use of the colon to indicate a range of variables from their original order in the initial data frame.

```
data1b <- subset(mtcars, select=c(mpg:hp,wt:qsec,am)) #(1:4,6:7,9)
gt::gt(headTail(data1b))
```

mpg	cyl	disp	hp	wt	qsec	am
21	6	160	110	2.62	16.46	1
21	6	160	110	2.88	17.02	1
22.8	4	108	93	2.32	18.61	1
21.4	6	258	110	3.21	19.44	0
...	...	...	...	...	...	...
15.8	8	351	264	3.17	14.5	1
19.7	6	145	175	2.77	15.5	1
15	8	301	335	3.57	14.6	1
21.4	4	121	109	2.78	18.6	1

2.2.2 Select a subset of cases using `subset`

Once again, lets also subset only those cars that have 6 cylinder engines. This is effectively a "select cases" operation.

But first lets count the number of cases for each value of cylinder. We know to expect seven cases from this selection based on the parallel steps we took above.

```
data2b <- subset( data1b, cyl==6)
gt::gt(data2b)
```

mpg	cyl	disp	hp	wt	qsec	am
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1
21.4	6	258.0	110	3.215	19.44	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0
19.7	6	145.0	175	2.770	15.50	1

It does seem to be a more straightforward approach using `subset`, but it can be even more efficient. We can combine the selection of cases and variables in one application of the subset function.

```
data2c <- subset(mtcars, cyl==6, select=c(mpg:hp,wt:qsec,am))
gt::gt(data2c)
```

mpg	cyl	disp	hp	wt	qsec	am
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1
21.4	6	258.0	110	3.215	19.44	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0

19.7	6	145.0	175	2.770	15.50	1

2.3 Random Samples of cases

At times, the need may arise for randomly sampling a subset of cases. The `sample` function provides this capability.

```
set.seed(12345)
sample1 <- mtcars[sample(1:nrow(mtcars), 8, replace=FALSE),]
knitr::kable(sample1)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4

It is possible to use the `subset` function to perform this case sampling action, but it is more complicated.

```
rn <- row.names(mtcars)
set.seed(12346)
sampRows <- sample(rn,8)
sampRows
```

```
## [1] "Toyota Corolla"     "Porsche 914-2"      "Hornet 4 Drive"
## [4] "Volvo 142E"         "Toyota Corona"      "Camaro Z28"
## [7] "Cadillac Fleetwood" "Merc 230"
```

```
sample2 <- subset(mtcars,rn%in%sampRows)
knitr::kable(sample2)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Note that the **dplyr** approach outlined below is simpler than both of these methods illustrated here.

2.4 Sorting the data frame using base system methods

With this simplified data frame (data2) we will more readily be able to see the results of our sorting operations. For reference, here is that simplified data frame created above:

```
gt::gt(data2)
```

mpg	cyl	disp	hp	wt	qsec	am
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1
21.4	6	258.0	110	3.215	19.44	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0
19.7	6	145.0	175	2.770	15.50	1

The most basic sorting operation is to sort a single numeric variable. Here, we sort the whole data frame as a function of car weight. Note that the default condition for the `order` function is to sort in ascending order. This generic usage of "bracketing" code is central for a variety of r operations using data frames.

```
data3 <- data2[order(data2$wt),]
gt::gt(data3)
```

mpg	cyl	disp	hp	wt	qsec	am
21.0	6	160.0	110	2.620	16.46	1
19.7	6	145.0	175	2.770	15.50	1
21.0	6	160.0	110	2.875	17.02	1
21.4	6	258.0	110	3.215	19.44	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0
18.1	6	225.0	105	3.460	20.22	0

2.4.1 Sorting in descending order by adding an argument.

```
data4 <- data2[order(data2$wt, decreasing=TRUE),]
gt::gt(data4)
```

mpg	cyl	disp	hp	wt	qsec	am
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
17.8	6	167.6	123	3.440	18.90	0
21.4	6	258.0	110	3.215	19.44	0
21.0	6	160.0	110	2.875	17.02	1
19.7	6	145.0	175	2.770	15.50	1
21.0	6	160.0	110	2.620	16.46	1

2.4.2 Sorting with more than one variable

We can sort simultaneously on more than one variable. This might be easiest to see using the "am" variable as one of the sorted ones. In this illustration with "am" and "mpg", it makes more sense to sort "am" first.

```
data5 <- data2[order(data2$am,data2$mpg),]
gt::gt(data5)
```

mpg	cyl	disp	hp	wt	qsec	am
17.8	6	167.6	123	3.440	18.90	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
21.4	6	258.0	110	3.215	19.44	0
19.7	6	145.0	175	2.770	15.50	1
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1

If we want to sort "am" as descending and mpg as ascending, we can still use just one implementation of the `order` function. However, to denote a descending/decreasing sort order, we put a minus sign in front of the variable name.

```
data6 <- data2[order(-data2$am,data2$mpg),]
gt::gt(data6)
```

mpg	cyl	disp	hp	wt	qsec	am
19.7	6	145.0	175	2.770	15.50	1
21.0	6	160.0	110	2.620	16.46	1
21.0	6	160.0	110	2.875	17.02	1
17.8	6	167.6	123	3.440	18.90	0
18.1	6	225.0	105	3.460	20.22	0
19.2	6	167.6	123	3.440	18.30	0
21.4	6	258.0	110	3.215	19.44	0

2.4.3 Create New Variables. E.g., Transformations.

The need often arises for scale transformations. This class of operations is what some of you will be familiar with as the COMPUTE procedure in SPSS.

If we examined the distribution of the "horsepower" variable in the `mtcars` data frame (<hist(mtcars$hp)>) we would find it to be positively skewed. We might want to do a scale transform to normalize it and might find that a square root transformation works well. How can we create this square root transformation and add the variable to the data frame? Let's do it with the minimized data frame we called "data2" above (the 6 cyl cars and only the subset of seven variables).

Even though the code for this operation is not challenging, readers may find that web searches for this method sometimes turn up a recommendation for a preceding step. The idea is to "initialize" the new variable by creating it with missing data for all cases. I find this unnecessaary, so I've commented it out as the first of the two lines of code that follow. The key idea here is that the arithmetic change to "hp" uses standard R code for the mathematical operator for square root. Essentially, we have "newvar" equals "old var" raised to the one-half power.

```
#data2$sqrthp <- NA
data2$sqrthp <- data2$hp**.5
gt::gt(data2)
```

mpg	cyl	disp	hp	wt	qsec	am	sqrthp
21.0	6	160.0	110	2.620	16.46	1	10.48809
21.0	6	160.0	110	2.875	17.02	1	10.48809
21.4	6	258.0	110	3.215	19.44	0	10.48809
18.1	6	225.0	105	3.460	20.22	0	10.24695
19.2	6	167.6	123	3.440	18.30	0	11.09054
17.8	6	167.6	123	3.440	18.90	0	11.09054
19.7	6	145.0	175	2.770	15.50	1	13.22876

2.4.4 Create New Variables. Conditionalizing new variable values (Recoding)

Often we wish to create new variables contingent on values of other variables. In the MTCARS data set, the "carb" variable is number of carburetors. It is coded as a numeric varialbe but could just as easily be a factor. Here, I create a new variable that is a factor with levels "high", "medium", and "small" to indicate groups of cars with carbureators in those subjectively created ranges. For this illustration the whole MTCARS data frame is used rather than the subsetted one from the illustrations above. First, a duplicate of the data frame is created just to separate this illustration from others in the document.

```
mtcarsnew <- mtcars
mtcarsnew$carbfactor[mtcarsnew$carb < 4] <- "low"
mtcarsnew$carbfactor[mtcarsnew$carb == 4] <- "middle"
mtcarsnew$carbfactor[mtcarsnew$carb > 4] <- "high"
#library(car)
#mtcarsnew$carbfactor <-  as.factor(recode(mtcarsnew$carb,
#                                  " c('1','2','3','4') = 'low'; else='high' "))
# look at first few rows to see the new variable
gt::gt(head(mtcarsnew))
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	carbfactor
21.0	6	160	110	3.90	2.620	16.46	0	1	4	4	middle
21.0	6	160	110	3.90	2.875	17.02	0	1	4	4	middle
22.8	4	108	93	3.85	2.320	18.61	1	1	4	1	low
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1	low

18.7	8	360	175	3.15	3.440	17.02	0	0	3	2	low
18.1	6	225	105	2.76	3.460	20.22	1	0	3	1	low

At first glance, this looks ok, However, the new variable, "carbfactor" is actually only a character vector at this point.

```
str(mtcarsnew)
```

```
## 'data.frame':    32 obs. of  12 variables:
##  $ mpg       : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl       : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp      : num  160 160 108 258 360 ...
##  $ hp        : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat      : num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt        : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec      : num  16.5 17 18.6 19.4 17 ...
##  $ vs        : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am        : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear      : num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb      : num  4 4 1 1 2 1 4 2 2 4 ...
##  $ carbfactor: chr  "middle" "middle" "low" "low" ...
```

In order to convert it to a factor we can use the `as.factor` function.

```
mtcarsnew$carbfactor <- as.factor(mtcarsnew$carbfactor)
```

This all works just fine, but is a bit laborious. An alternative that is more efficient is to use the `recode` function from the **car** package written by John Fox. It is used by a great many R analysts.

2.4.5 Using `recode` to recode/create new variables.

The example above with carburetors is one where we wanted to recode a numeric into a factor. This a can be accomplished with `recode`. Lets do the "carb" to "carbfactor" recode with the `recode` function in the **car** package.

Note that both **dplyr** and **car** have a `recode` function. Since both packages are loaded in this doc, I have explicitly called the **car** package `recode` here with the `car::` syntax.

The use of `car::recode` is probably easier than the approach taken in the previous section, but the tidyverse approach (below) may be equally simple to learn.

```
# start with a fresh copy of the data frame
mtcarsnew2 <- mtcars
mtcarsnew2$carbfactor <-
  car::recode(mtcarsnew2$carb,"1:3='low';
                               4:4='middle';
                               5:8='high'")
# and again make sure it is a factor
mtcarsnew2$carbfactor <- as.factor(mtcarsnew2$carbfactor)
gt::gt(head(mtcarsnew2))
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	carbfactor
21.0	6	160	110	3.90	2.620	16.46	0	1	4	4	middle
21.0	6	160	110	3.90	2.875	17.02	0	1	4	4	middle
22.8	4	108	93	3.85	2.320	18.61	1	1	4	1	low
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1	low
18.7	8	360	175	3.15	3.440	17.02	0	0	3	2	low
18.1	6	225	105	2.76	3.460	20.22	1	0	3	1	low

2.4.6 Recoding is a large topic

In addition, we need to broaden perspectives a bit. We have several potential related needs:

- Recoding into the same variable
- Recoding into a different variable
- Recoding a numeric into a factor
- Recoding a factor into a factor
- Recoding a numeric into different numeric values or ranges.

There are many R functions in many packages that address this suite of needs. It is helpful to be aware of them, but this document has limited the illustrations to the ones above, and the **dplyr** approach below. The following functions are worthwhile exploring (in addition to the base system bracketing notation approach):

- **base**
  - `cut` function
  - `ifelse` function
- **car**
  - `recode` function
- **dplyr**
  - `recode` function
  - `if_else` function
  - `mutate` function
- **sjmisc**
  - `rec` function
  - `rec_to` function
  - `rec_to_if`)

The following URLs provides a starting point to describe the possibilities with some of these functions:

http://dwoll.de/rexrepos/posts/recode.html

http://www.cookbook-r.com/Manipulating_data/Recoding_data/

2.5 A post script on base system subsetting and data wrangling

It is important to realize that both the bracketing notation system and the `subset` function are designed for a far broader range of applications than just the data frame tasks we have outlined here (such as operations on vectors and matrices). Eventually, the user may find some pitfalls in suing the bracketing notation for recoding. For example, examine the results from this sequence of code and compare what the bracketing and `subset` approaches produce.

```
mtcars2 <- mtcars
#set the cylinder value for the first case to missing
mtcars2[1,]$cyl <- NA
#examine the outcome by looking at the first 3 rows
head(mtcars2, 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0  NA  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
# now select only the six cylinder models using bracketing
mtcars[mtcars2$cyl == 6,]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## NA               NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
```

```
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

```
# select the six cylinder cars by using `subset`
subset(mtcars, cyl == 6)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

The product of the `subset` function (2nd illustration) would probably be what one expected and would have wanted. The product of the bracketd notation does something completely unexpected (the "NA" row). Rather than explain why (long story), it is useful to see this illustration as a result of the fact that the bracketing notation has nuanced meaning depending on its application. The issue with missing data is a reflection of that.

Perhaps the take-home message is that `subset` is less prone to problems in the types of applications that researchers most commonly encounter with data frames.

However, there is an easy way to fix the problem demonstrated above:

```
na.omit(mtcars[mtcars2$cyl == 6,])
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

This issue is partly why I subset the data frame to six cylinder cars using the `which` function in conjunction with the bracketing notation when I first showed the select cases approach above (code repeated here).

```
data2 <- data1[which(data1$cyl==6),]
```

3 Data Frame Operations with tidyverse tools

The base system methods above, using subsetting practices, the bracketing notation and the `subset` function may be non-intuitive to the R novice, but they work well once learned. Alternative approaches are available with tools from the **dplyr** package. These functions and notational structure may be simpler for the novice.

Here is a listing of the primary **dplyr** tools and their descriptions:

Functions	Description
select()	select columns
filter()	filter rows
arrange()	re-order or arrange rows
mutate()	create new columns/variables
summarise()	summarize values
group_by()	allows for group operations in the split-apply-combine family
recode	recode values of many kinds of variables and vectors
if_else	use as a replacement for the base system `ifelse` function

3.1 Selecting a subset of variables using the `select` function from dplyr

We will begin with the same **mtcars** data frame and subset it so that we use the identical set of variables as above.

This is a bit simpler than the bracketing notation. Make sure to notice that the first argument passed to `select` is the name of the data frame to be subject to selection - then we just name the variables to be selected.

```
data7 <- dplyr::select(mtcars, mpg, cyl,hp, disp, wt, qsec,am)
gt::gt(headTail(data7))
```

mpg	cyl	hp	disp	wt	qsec	am
21	6	110	160	2.62	16.46	1
21	6	110	160	2.88	17.02	1
22.8	4	93	108	2.32	18.61	1
21.4	6	110	258	3.21	19.44	0
...	...	...	...	...	...	...
15.8	8	264	351	3.17	14.5	1
19.7	6	175	145	2.77	15.5	1
15	8	335	301	3.57	14.6	1
21.4	4	109	121	2.78	18.6	1

Sometimes it is useful to select a subset of variables by indicating which variables are to be excluded, rather than included as we just did above. This can be done by placing a minus sign in front of the names of the variables to be excluded.

Same result, different way of getting there:

```
data7b <- dplyr::select(mtcars, -drat, -vs, -gear, -carb)
gt::gt(headTail(data7b))
```

mpg	cyl	disp	hp	wt	qsec	am
21	6	160	110	2.62	16.46	1
21	6	160	110	2.88	17.02	1

22.8	4	108	93	2.32	18.61	1
21.4	6	258	110	3.21	19.44	0
...	...	...	...	...	...	...
15.8	8	351	264	3.17	14.5	1
19.7	6	145	175	2.77	15.5	1
15	8	301	335	3.57	14.6	1
21.4	4	121	109	2.78	18.6	1

3.2 Selecting Cases using the `filter` function from dplyr

Selecting cases (rows) is accomplished using the `filter` function from **dplyr**. It has the same syntax as `select` where the first argument to be passed is the name of the data frame, and then the specification of variable values on which to base the selection. This data8 data frame is identical to the data2 data frame that we used above in the base systems section We will do the same sorting on this dataframe that we did above (after detour in the section below this data8 creation)

```
data8 <- dplyr::filter(data7, cyl==6)
gt::gt(data8)
```

mpg	cyl	hp	disp	wt	qsec	am
21.0	6	110	160.0	2.620	16.46	1
21.0	6	110	160.0	2.875	17.02	1
21.4	6	110	258.0	3.215	19.44	0
18.1	6	105	225.0	3.460	20.22	0
19.2	6	123	167.6	3.440	18.30	0
17.8	6	123	167.6	3.440	18.90	0
19.7	6	175	145.0	2.770	15.50	1

Also notice that we might select on the basis of more than one variable's values. Here, as an illustration, we select cases that have manual transmissions, and have mpg less than 20.

```
data9 <- dplyr::filter(mtcars, mpg < 20, am==0)
gt::gt(data9)
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2

3.3 Using the arrange function for sorting data frames.

If we need to sort whole data frames according to one or more varialbes' values, the `arrange()` function in **dplyr** is our tidyverse tool of choice. These illustrations mirror what we did to produce data3 through data6 above.

First, lets sort the data8 data frame on the "wt" variable.

```
data10 <- dplyr::arrange(data8, wt )
gt::gt(data10)
```

mpg	cyl	hp	disp	wt	qsec	am
21.0	6	110	160.0	2.620	16.46	1
19.7	6	175	145.0	2.770	15.50	1
21.0	6	110	160.0	2.875	17.02	1
21.4	6	110	258.0	3.215	19.44	0
19.2	6	123	167.6	3.440	18.30	0
17.8	6	123	167.6	3.440	18.90	0
18.1	6	105	225.0	3.460	20.22	0

Next, sort on "wt" once again, but in descending order this time. Note the descending order specification with the minus sigh. We could also have specified `desc(wt)`.

```
data11 <- dplyr::arrange(data8, -wt)
gt::gt(data11)
```

mpg	cyl	hp	disp	wt	qsec	am
18.1	6	105	225.0	3.460	20.22	0
19.2	6	123	167.6	3.440	18.30	0
17.8	6	123	167.6	3.440	18.90	0
21.4	6	110	258.0	3.215	19.44	0
21.0	6	110	160.0	2.875	17.02	1
19.7	6	175	145.0	2.770	15.50	1
21.0	6	110	160.0	2.620	16.46	1

Next, we repeat the data5 sort that sorted both on "am" and "mpg", both ascending (ascending is the default).

```
data12 <- dplyr::arrange(data8, am, mpg)
gt::gt(data12)
```

mpg	cyl	hp	disp	wt	qsec	am
17.8	6	123	167.6	3.440	18.90	0
18.1	6	105	225.0	3.460	20.22	0
19.2	6	123	167.6	3.440	18.30	0
21.4	6	110	258.0	3.215	19.44	0
19.7	6	175	145.0	2.770	15.50	1
21.0	6	110	160.0	2.620	16.46	1
21.0	6	110	160.0	2.875	17.02	1

Finally, we will repeat the data6 sort by sorting on "am", descending, and "mpg" ascending:

```
data13 <- dplyr::arrange(data8, -am, mpg)
gt::gt(data13)
```

mpg	cyl	hp	disp	wt	qsec	am
19.7	6	175	145.0	2.770	15.50	1
21.0	6	110	160.0	2.620	16.46	1
21.0	6	110	160.0	2.875	17.02	1
17.8	6	123	167.6	3.440	18.90	0
18.1	6	105	225.0	3.460	20.22	0
19.2	6	123	167.6	3.440	18.30	0
21.4	6	110	258.0	3.215	19.44	0

3.4 Creating new variables with the `mutate` function from dplyr

New variables can be created and added to data frames simply with the `mutate` function. Two examples are provided here. First, lets do a numeric transformation of a variable. If we examined the distribution of the "horsepower" variable in the `mtcars` data frame (<hist(mtcars$hp)>) we would find it to be positively skewed. We might want to do a scale transform to normalize it and might find that a square root transformation works well. How can we create this square root transformation and add the variable to the data frame? Let's do it with the minimized data frame we called "data8" above (the 6 cyl cars and only the subset of seven variables).

```
data14 <- dplyr::mutate(data8, sqrt.hp = hp^.5)
gt::gt(data14)
```

mpg	cyl	hp	disp	wt	qsec	am	sqrt.hp
21.0	6	110	160.0	2.620	16.46	1	10.48809
21.0	6	110	160.0	2.875	17.02	1	10.48809
21.4	6	110	258.0	3.215	19.44	0	10.48809
18.1	6	105	225.0	3.460	20.22	0	10.24695
19.2	6	123	167.6	3.440	18.30	0	11.09054
17.8	6	123	167.6	3.440	18.90	0	11.09054
19.7	6	175	145.0	2.770	15.50	1	13.22876

A second example with mutate does something a little bit different. Please recall that the original `mtcars` data frame has rownames that are the make/model of the car. There may be instances where we would rather have those text descriptors as a variable (a string variable or factor). We can achieve that with `mutate`.

```
data15 <- dplyr::mutate(mtcars, make=rownames(mtcars))
gt::gt(headTail(data15))
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	make
21	6	160	110	3.9	2.62	16.46	0	1	4	4	Mazda RX4
21	6	160	110	3.9	2.88	17.02	0	1	4	4	Mazda RX4 Wag
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1	Datsun 710
21.4	6	258	110	3.08	3.21	19.44	1	0	3	1	Hornet 4 Drive
...	...	...	...	...	...	...	...	...	...	...	NA
15.8	8	351	264	4.22	3.17	14.5	0	1	5	4	Ford Pantera L
19.7	6	145	175	3.62	2.77	15.5	0	1	5	6	Ferrari Dino
15	8	301	335	3.54	3.57	14.6	0	1	5	8	Maserati Bora
21.4	4	121	109	4.11	2.78	18.6	1	1	4	2	Volvo 142E

3.5 Recoding variables with the `recode` function in dplyr

This section duplicates the recode application that we did above with the **car** function also called `recode`. The **dplyr** `recode` function is used with similar logic. To be explicit, I accomplished recodes for individual values of carb by using separate arguments rather than ranges of values. Once again, an additional step is required to specify the recoded variable as a factor, and we can doublecheck with 'str'.

```
mtcarsnew3 <- mtcars
mtcarsnew3$carbfactor <- dplyr::recode(mtcarsnew3$carb,
                                `1`="low",
                                `2`="low",
                                `3`="low",
                                `4`="middle",
                                `5`="high",
                                `6`="high",
                                `7`="high",
                                `8`="high",
                                )
mtcarsnew3$carbfactor <- as.factor(mtcarsnew3$carbfactor)
str(mtcarsnew3)
```

```
## 'data.frame':    32 obs. of  12 variables:
##  $ mpg       : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl       : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp      : num  160 160 108 258 360 ...
##  $ hp        : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat      : num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt        : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec      : num  16.5 17 18.6 19.4 17 ...
##  $ vs        : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am        : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear      : num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb      : num  4 4 1 1 2 1 4 2 2 4 ...
##  $ carbfactor: Factor w/ 3 levels "high","low","middle": 3 3 2 2 2 2 3 2 2 3 ...
```

```
gt::gt(head(mtcarsnew3))
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	carbfactor
21.0	6	160	110	3.90	2.620	16.46	0	1	4	4	middle
21.0	6	160	110	3.90	2.875	17.02	0	1	4	4	middle
22.8	4	108	93	3.85	2.320	18.61	1	1	4	1	low
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1	low
18.7	8	360	175	3.15	3.440	17.02	0	0	3	2	low
18.1	6	225	105	2.76	3.460	20.22	1	0	3	1	low

As was indicated in the earlier recode section in this document, there are many different types of applications of recoding that may be required. This section was meant only as a brief introduction to the `dplyr::recode` approach. The reader might consult the following URL for a broader set of illustrations using `dplyr::recode`:

https://dplyr.tidyverse.org/reference/recode.html

In addition to the `dplyr::recode` approach, the tidyverse provides several other possibilities for various recoding capabilities.

- **dplyr**
  - `recode` function

  - `if_else` function
  - `mutate` function
- **forcats**
  - `fct_recode` function plus many others for working with factors

See the following URL for information on the **forcats** package which is a tidyverse package that provides many helper functions for working with factors.

https://cran.r-project.org/web/packages/forcats/forcats.pdf

3.6 Pipes

In the world of tidyverse programming, there is a growing approach to writing code that is rather different than what you have seen from many of our examples over the course of the apsy510/511 courses. This involves the use of what are called pipes. To illustrate, lets assume we wanted to accomplish five things with our original "mtcars" data frame: 1. select only the 6 cylinder models 2. include only the "mpg", "wt", "hp", and "am" variables 3. Calculate the square root of "hp" 4. Add value labels for the "am" variable (transmission type) 5. Summarize the "mpg" and "hp" variables as a function of the "am" variable (transmission type)

In base system code, we would create several intermediate data frames to acheive the final one on which to use `tapply` to obtain the group means.

```
# first create a working copy of mtcars and then change "am" to a factor, adding value labels
mtcars2 <- mtcars
mtcars2$am <- factor(mtcars2$am,labels=c("automatic","manual"))

data16 <- mtcars2[which(mtcars$cyl==6),]
data17 <- data15[,c(1:4,6:7,9)]
data17$sqrt.hp <- data17$hp^.5
#gt::gt(data17)
tapply(data17$sqrt.hp, data17$am, mean)
```

```
##        0        1
## 12.47573 10.81007
```

This approach (above) is comfortable, in some ways. It permits creation of the intermediate objects that can then be used for other purposes. But it is a bit inefficient and in some ways is the reverse of the way of thinking about extracting the information in a more natural sequence. The use of "pipes" is a way to be efficient and more direct in the process. We recognize that the goal was to obtain two group means on a created variable in a subset of the original data set. The "pipes" operator is **%>%**. One can read the following code in a manner that says, "with the mtcars3 data frame, first use 'filter' to choose six cylinder cars, then use 'mutate', then use 'select', then use 'group_by', and then summarize". This singular flow accomplishes five steps in one process.

```
# first, rework "am" as a factor and add value labels
mtcars3 <- mtcars
mtcars3$am <- factor(mtcars3$am,labels=c("automatic","manual"))
# use pipes to integrate all the steps into one flow
mtcars3 %>%
  dplyr::filter(cyl==6)%>%
  dplyr::mutate(sqrt.hp = hp^.5)%>%
  dplyr::select(mpg,cyl,wt,hp,disp,am,sqrt.hp)%>%
  dplyr::group_by(am)%>%
  dplyr::summarize(mean_sqrt.hp=mean(sqrt.hp))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 2
```

```
##   am        mean_sqrt.hp
##   <fct>            <dbl>
## 1 automatic         10.7
## 2 manual            11.4
```

3.7 Frequency counts of cases for categorical variables

This section is included as another example of **tidyverse** tools, including pipes, but also a useful data summary method. The illustration is taken from a blog post at "Statistical Odds & Ends":

https://statisticaloddsandends.wordpress.com/2020/07/23/tidyrcomplete-to-show-all-possible-combinations-of-variables/

Often, it is useful to find frqeuency counts of numbers of cases in categories defined by multiple categorical variables. We can use the `mtcars` data set for this and pose the question of how many makes of cars are found in combinations of cylinder type ("cyl") and transmission gears ("gear"). There are three cylinder types and three gear types as determined from:

```
plyr::count(mtcars, 'cyl')
```

```
##   cyl freq
## 1   4   11
## 2   6    7
## 3   8   14
```

```
plyr::count(mtcars, 'gear')
```

```
##   gear freq
## 1    3   15
## 2    4   12
## 3    5    5
```

Now let's look at combinations of "cyl" and "gear". There are nine possible combinations. The `group_by` function doesn't change `mtcars` by sorting it, it justs set in place a directive to other `tidyverse` functions to perform their operation by the categorization set by `group_by`. Also, the '.groups='drop'' argument is included to avoid a warning message that is irrelevant for the purpose of this example (see ?dplyr::summarize for details).

```
mtcars %>%
  dplyr::group_by(cyl, gear) %>%
  dplyr::summarize(count = n(), .groups = 'drop')
```

```
## # A tibble: 8 x 3
##     cyl  gear count
##   <dbl> <dbl> <int>
## 1     4     3     1
## 2     4     4     8
## 3     4     5     2
## 4     6     3     2
## 5     6     4     4
## 6     6     5     1
## 7     8     3    12
## 8     8     5     2
```

Notice that only eight of the nine combinations of the two categorical variables are listed. This implies that one of them must have a zero count.

In order to have a table that includes zero-count categories, we can modify the code this using the `ungroup` and `complete` functions.

```
mtcars %>%
    group_by(cyl, gear) %>%
    dplyr::summarize(count = n(), .groups = 'drop') %>%
    dplyr::ungroup() %>%
    tidyr::complete(cyl, gear)
```

```
## # A tibble: 9 x 3
##     cyl  gear count
##   <dbl> <dbl> <int>
## 1     4     3     1
## 2     4     4     8
## 3     4     5     2
## 4     6     3     2
## 5     6     4     4
## 6     6     5     1
## 7     8     3    12
## 8     8     4    NA
## 9     8     5     2
```

Now we have all nine combinations but it doesn't show the freuency count of eight cylinder four gear cars as zero. Instead it shows it as missing (NA). In order to rectify this, we modify the code again, this time producing the finished/useful approach:

```
mtcars %>%
    dplyr::group_by(cyl, gear) %>%
    dplyr::summarize(count = n(), .groups = 'drop') %>%
    dplyr::ungroup() %>%
    tidyr::complete(cyl, gear, fill = list(count = 0))
```

```
## # A tibble: 9 x 3
##     cyl  gear count
##   <dbl> <dbl> <dbl>
## 1     4     3     1
## 2     4     4     8
## 3     4     5     2
## 4     6     3     2
## 5     6     4     4
## 6     6     5     1
## 7     8     3    12
## 8     8     4     0
## 9     8     5     2
```

Now lets try a combination of three categorical variables ("am" is automatic vs manual transmission), thus producing 18 categories:

```
mtcars %>%
    dplyr::group_by(cyl, gear, am) %>%
    dplyr::summarize(count = n(), .groups = 'drop') %>%
    dplyr::ungroup() %>%
    tidyr::complete(cyl, gear, am, fill = list(count = 0))
```

```
## # A tibble: 18 x 4
##      cyl  gear    am count
##    <dbl> <dbl> <dbl> <dbl>
##  1     4     3     0     1
##  2     4     3     1     0
```

```
##  3     4     4     0     2
##  4     4     4     1     6
##  5     4     5     0     0
##  6     4     5     1     2
##  7     6     3     0     2
##  8     6     3     1     0
##  9     6     4     0     2
## 10     6     4     1     2
## 11     6     5     0     0
## 12     6     5     1     1
## 13     8     3     0    12
## 14     8     3     1     0
## 15     8     4     0     0
## 16     8     4     1     0
## 17     8     5     0     0
## 18     8     5     1     2
```

3.8 Postscript on Tidyverse tools

The data wrangling tools and other capabilities in the Tidyverse are extensive. Web resources are numerous. Here are a few links that lead to useful sites for basics of **dplyr** usage.

https://dplyr.tidyverse.org/
https://www.listendata.com/2016/08/dplyr-tutorial.html
https://genomicsclass.github.io/book/pages/dplyr_tutorial.html
https://datacarpentry.org/R-ecology-lesson/03-dplyr.html
https://mockquant.blogspot.com/2015/07/the-complete-catalog-of-argument.html
https://www.dataschool.io/dplyr-tutorial-for-faster-data-manipulation-in-r/
https://rpubs.com/justmarkham/dplyr-tutorial