

Intro To dplyr

Steve Pittard

2020-01-25

Contents

1	R Data Structures	5
1.1	Vectors	5
1.2	Data Frames	7
1.3	A Reference Data Frame	8
1.4	Relation to dplyr	9
2	Digging In	11
2.1	Structure	13
2.2	Meta Information	13
2.3	Printing	14
2.4	Accessing Rows And Columns	14
2.5	Interrogation	16
2.6	Missing values	18
2.7	Continuous vs Factors	21
2.8	Sorting data	22
2.9	Reading CSV Files	24
3	The tidyverse	25
3.1	Installing	26
3.2	dplyr Basics	27
3.3	First Steps	27
4	Split Apply Combine	39
4.1	What Are Pipes ?	40
4.2	Pipes and Split Apply Combine	42

Chapter 1

R Data Structures

There are a number of data structures in R such as **vectors**, **lists** and **matrices**. The **vector** structure winds up being helpful in understanding how to work with data frames.

1.1 Vectors

It is a container for a series of related data of the same type: height measurements of students, whether a group of people smoke or not, their blood pressure. The only rule here is that a vector can contain only one data type at a time.

```
names <- c("P1", "P2", "P3", "P4", "P5")
temp  <- c(98.2, 101.3, 97.2, 100.2, 98.5)
pulse <- c(66, 72, 83, 85, 90)
gender <- c("M", "F", "M", "M", "F")
```

To access elements, or ranges of elements, within a vector involves using the “bracket” notation:

```
# Get the first element of temp
temp[1]
```

```
## [1] 98.2
```

```
# Get elements 3, 4, and 5 from pulse
pulse[3:5]
```

```
## [1] 83 85 90
```

We can also use logical expressions to find elements that satisfy some condition. This is a very powerful capability in R.

```
temp < 98
```

```
## [1] FALSE FALSE TRUE FALSE FALSE
```

Whoa. What was that ? Well we get back a T/F logical vector that tells us what elements satisfy the specified condition. We can then use this info to get the elements of interest.

```
temp[temp < 98]
```

```
## [1] 97.2
```

Working with individual vectors is fine but a more general way of working with them is in data frames which provides more flexibility:

```
(my_df <- data.frame(names,temp,pulse,gender))
```

```
##   names  temp pulse gender
## 1   P1  98.2    66      M
## 2   P2 101.3    72      F
## 3   P3  97.2    83      M
## 4   P4 100.2    85      M
## 5   P5  98.5    90      F
```

Looking at each column, we see that they are the vectors we were just working with. If we need to access them from the data frame it's easy.

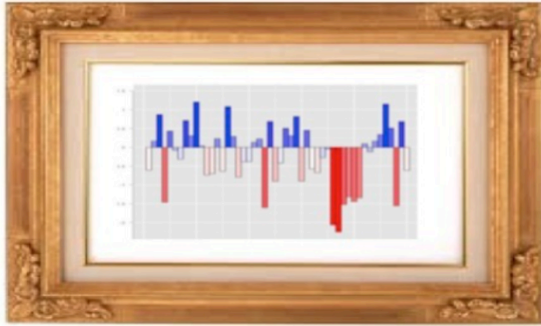
```
# Get the temp column
my_df$temp
```

```
## [1] 98.2 101.3 97.2 100.2 98.5
```

```
# What's the mean of the temp column
mean(my_df$temp)
```

```
## [1] 99.08
```

1.2 Data Frames



But we are getting ahead of ourselves. Just know that the **premier data structure** in R is the **data.frame**. This structure can be described as follows:

- A data frame is a special type of list that contains data in a format that allows for easier manipulation, reshaping, and open-ended analysis
- Data frames are tightly coupled collections of variables. It is one of the more important constructs you will encounter when using R so learn all you can about it
- A data frame is an analogue to the Excel spreadsheet but is much more flexible for storing, manipulating, and analyzing data
- Data frames can be constructed from existing vectors, lists, or matrices. Many times they are created by reading in comma delimited files, (CSV files), using the `read.table` command

Once you become accustomed to working with data frames, R becomes so much easier to use. In fact, it could be well argued tht UNTIL you wrap your head around the data frame concept then you cannot be productive in R. This is mostly true, in my experience.

R comes with with a variety of built-in data sets that are very useful for getting used to data sets and how to manipulate them.

<code>AirPassengers</code>	Monthly Airline Passenger Numbers 1949-1960
<code>BJsales</code>	Sales Data with Leading Indicator
<code>BOD</code>	Biochemical Oxygen Demand
<code>CO2</code>	Carbon Dioxide Uptake in Grass Plants
<code>ChickWeight</code>	Weight versus age of chicks on different diets
<code>DNase</code>	Elisa assay of DNase
<code>Formaldehyde</code>	Determination of Formaldehyde
<code>HairEyeColor</code>	Hair and Eye Color of Statistics Students
<code>Harman23.cor</code>	Harman Example 2.3

Harman74.cor	Harman Example 7.4
Indometh	Pharmacokinetics of Indomethacin
InsectSprays	Effectiveness of Insect Sprays
JohnsonJohnson	Quarterly Earnings per Johnson & Johnson Share
LakeHuron	Level of Lake Huron 1875-1972
LifeCycleSavings	Intercountry Life-Cycle Savings Data
Loblolly	Growth of Loblolly pine trees
Nile	Flow of the River Nile
Orange	Growth of Orange Trees
OrchardSprays	Potency of Orchard Sprays
PlantGrowth	Results from an Experiment on Plant Growth
Puromycin	Reaction Velocity of an Enzymatic Reaction
Theoph	Pharmacokinetics of Theophylline

1.3 A Reference Data Frame

We will use a well-known data frame, at least in R circles, called **mtcars** which is part of any default installation of R. It is a simple data set relating to, well, automobiles. This data frame has the distinction of being the most (ab)used data frame in R education.

The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973-74 models).

A data frame with 32 observations on 11 (numeric) variables.

```
[, 1]  mpg Miles/(US) gallon
[, 2]  cyl Number of cylinders
[, 3]  disp  Displacement (cu.in.)
[, 4]  hp   Gross horsepower
[, 5]  drat  Rear axle ratio
[, 6]  wt   Weight (1000 lbs)
[, 7]  qsec  1/4 mile time
[, 8]  vs   Engine (0 = V-shaped, 1 = straight)
[, 9]  am   Transmission (0 = automatic, 1 = manual)
[,10]  gear  Number of forward gears
[,11]  carb  Number of carburetors
```


1.4 Relation to dplyr

What you will discover is that the **dplyr** package, which itself is part of the much larger **tidyverse** package set, extends upon the idea of the basic R data frame in a way that some feel is superior. It depends on your point of view though the **tidyverse** has a lot of what I call a philosophic consistency in it which makes it **very** useful once you get some concepts in mind.

While you could start exclusively with **dplyr** and the **tidyverse** the world is still full of older code. Plus, many of the advantages of **dplyr** only become quite apparent when compared to the “older way” of doing things. So my recommendation is to know how to deal with data frames in base R while also spending time to learn the **dplyr** way of doing things.

Chapter 2

Digging In

Data frames look like an Excel Spreadsheet. The rows are observations and the columns are variables or “features” that represent some measurement or character-based description of a given observation. When viewed from the row point of view, the data can be heterogenous. When viewed as a column, the data is homogenous.

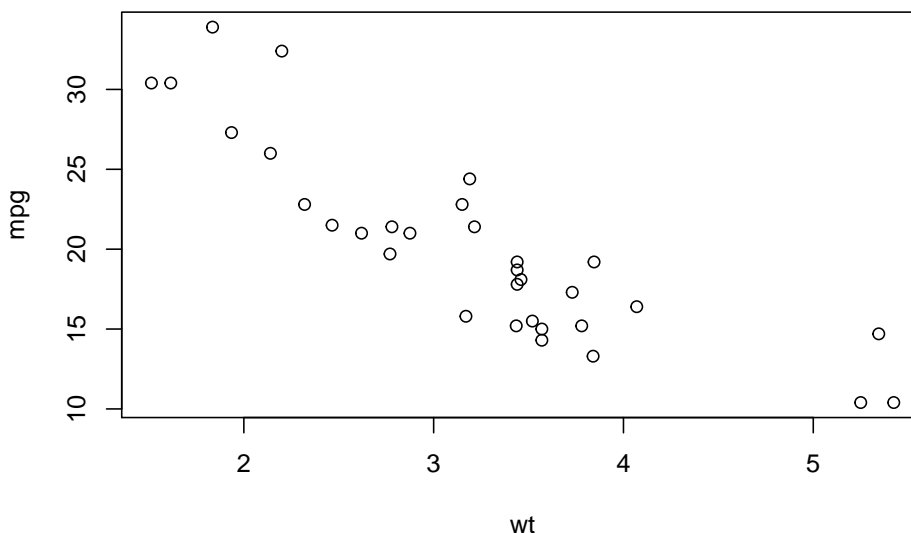
```
data(mtcars)
mtcars
```

##	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
## Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
## Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
## Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
## Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
## Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
## Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
## Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
## 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
## Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
## Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
## Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
## Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
## Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
## Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
## Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
## Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
## 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
## Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
## AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
## Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
## Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
## Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
## Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
## Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
## Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
## Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
## Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
## Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

We can do this with this data such as make plots or create models:

```
plot(mpg ~ wt, data=mtcars)
```



Let's create a regression model. It doesn't take long to realize that most functions in R will use a data frame as input. This means that you will spend a lot of time working with data frames to get them into shape for use with modeling and visualization tools. In fact you will spend most of your time **importing, transforming, and cleaning**.

```
(mylm <- lm(mpg ~ ., data = mtcars))
```

```
##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Coefficients:
```

```
## (Intercept)          cyl          disp          hp          drat
##    12.30337        -0.11144        0.01334       -0.02148        0.78711
##           wt          qsec          vs          am          gear
##   -3.71530         0.82104        0.31776         2.52023        0.65541
##           carb
##   -0.19942
```

There are some useful functions that help you understand the structure of a data frame. One of the most important ones is called the `str()` function which is short hand for **structure**.

2.1 Structure

```
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 ...
```

This gives you some idea about the number of rows and columns of the data frame along with a description of the variable types and their values. I use this function frequently. Other functions that will help you include the following.

2.2 Meta Information

```
# how many rows
nrow(mtcars)

## [1] 32

# how many columns
ncol(mtcars)

## [1] 11
```

```
# Column names
names(mtcars)

## [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
## [11] "carb"
```

2.3 Printing

Some data frames, such as `mtcars`, don't have many rows but others might have hundreds, thousands or even more than that ! Imagine trying to view one of those data frames. It is for this reason that the `head()` and `tail()` functions exist.

```
head(mtcars,5) # First 5 rows

##           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

tail(mtcars,3) # Last 3 rows

##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Ferrari Dino  19.7   6  145 175  3.62 2.77 15.5  0  1    5    6
## Maserati Bora 15.0   8  301 335  3.54 3.57 14.6  0  1    5    8
## Volvo 142E    21.4   4  121 109  4.11 2.78 18.6  1  1    4    2
```

2.4 Accessing Rows And Columns

There are various ways to select, remove, or exclude rows and columns of a data frame. We use the **bracket** notation to do this. This is very powerful. Keep in mind that data frames have rows and columns so it would make sense that you need a way to specify what rows and columns you want to access.

```
mtcars[1,] # First row, all columns

##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21    6  160 110  3.9 2.62 16.46 0  1    4    4

mtcars[1:3,] # First three rows, all columns

##           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
```

```
# All rows, and first 4 columns
```

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

```
##           mpg  cyl  disp  hp
## Mazda RX4      21.0   6 160.0 110
## Mazda RX4 Wag  21.0   6 160.0 110
## Datsun 710     22.8   4 108.0  93
## Hornet 4 Drive  21.4   6 258.0 110
## Hornet Sportabout 18.7   8 360.0 175
## Valiant        18.1   6 225.0 105
## Duster 360     14.3   8 360.0 245
## Merc 240D      24.4   4 146.7  62
## Merc 230       22.8   4 140.8  95
## Merc 280       19.2   6 167.6 123
## Merc 280C      17.8   6 167.6 123
## Merc 450SE     16.4   8 275.8 180
## Merc 450SL     17.3   8 275.8 180
## Merc 450SLC    15.2   8 275.8 180
## Cadillac Fleetwood 10.4   8 472.0 205
## Lincoln Continental 10.4   8 460.0 215
## Chrysler Imperial 14.7   8 440.0 230
## Fiat 128       32.4   4  78.7  66
## Honda Civic    30.4   4  75.7  52
## Toyota Corolla 33.9   4  71.1  65
## Toyota Corona  21.5   4 120.1  97
## Dodge Challenger 15.5   8 318.0 150
## AMC Javelin    15.2   8 304.0 150
## Camaro Z28     13.3   8 350.0 245
## Pontiac Firebird 19.2   8 400.0 175
## Fiat X1-9      27.3   4  79.0  66
## Porsche 914-2  26.0   4 120.3  91
## Lotus Europa   30.4   4  95.1 113
## Ford Pantera L  15.8   8 351.0 264
## Ferrari Dino   19.7   6 145.0 175
## Maserati Bora   15.0   8 301.0 335
## Volvo 142E     21.4   4 121.0 109
```

```
# Rows 1-5 and columns 1,2 and 8-10
```

```
mtcars[1:4,c(1:2,8:10)]
```

```
##           mpg  cyl vs am gear
## Mazda RX4      21.0   6  0  1   4
## Mazda RX4 Wag  21.0   6  0  1   4
## Datsun 710     22.8   4  1  1   4
## Hornet 4 Drive  21.4   6  1  0   3
```

```
# Rows 1-5 and columns 1,2 and 8-10
mtcars[1:4,c(1:2,8:10)]
```

```
##           mpg cyl vs am gear
## Mazda RX4      21.0   6  0  1    4
## Mazda RX4 Wag  21.0   6  0  1    4
## Datsun 710      22.8   4  1  1    4
## Hornet 4 Drive 21.4   6  1  0    3
```

```
# Rows 1-5 and columns by name
mtcars[1:4,c("mpg","wt","drat")]
```

```
##           mpg    wt drat
## Mazda RX4      21.0 2.620 3.90
## Mazda RX4 Wag  21.0 2.875 3.90
## Datsun 710      22.8 2.320 3.85
## Hornet 4 Drive 21.4 3.215 3.08
```

2.5 Interrogation

Many times you will wish to find rows that satisfy certain conditions. For example, what rows have an `mpg > 11` and at `wt < 2.0`? We use the bracket notation to help us. We can pass logical conditions into the brackets. Note the following:

```
mtcars$mpg > 11 & mtcars$wt < 2.0
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
## [23] FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE
```

There are 32 elements in this logical vector each with a value of either TRUE or FALSE. When passed into the row index of the bracket notation, it will print that row if the corresponding value is TRUE. If FALSE, the row will not be printed.

```
mtcars[mtcars$mpg > 11 & mtcars$wt < 2.0,]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Honda Civic  30.4   4 75.7  52 4.93 1.615 18.52  1  1   4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1   4    1
## 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
```

What if we just want to know how many cars satisfy this condition?


```
nrow(mtcars[mtcars$mpg > 11 & mtcars$wt < 2.0,])
```

```
## [1] 4
```

Find all rows that correspond to cars with 4 cylinders

```
mtcars[mtcars$cyl == 4,]
```

```
##           mpg  cyl  disp  hp drat   wt  qsec vs  am  gear  carb
## Datsun 710   22.8   4 108.0  93 3.85 2.320 18.61 1   1    4    1
## Merc 240D   24.4   4 146.7  62 3.69 3.190 20.00 1   0    4    2
## Merc 230    22.8   4 140.8  95 3.92 3.150 22.90 1   0    4    2
## Fiat 128    32.4   4  78.7  66 4.08 2.200 19.47 1   1    4    1
## Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52 1   1    4    2
## 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
## Fiat X1-9   27.3   4  79.0  66 4.08 1.935 18.90 1   1    4    1
## Porsche 914-2 26.0  4 120.3  91 4.43 2.140 16.70 0   1    5    2
## Lotus Europa 30.4  4  95.1 113 3.77 1.513 16.90 1   1    5    2
## Volvo 142E  21.4  4 121.0 109 4.11 2.780 18.60 1   1    4    2
```

We can even use other R functions in the bracket notation. Extract all rows whose MPG value exceeds the mean MPG for the entire data frame.

```
mtcars[mtcars$mpg > mean(mtcars$mpg),]
```

```
##           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
## Datsun 710   22.8   4 108.0  93 3.85 2.320 18.61 1   1    4    1
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44 1   0    3    1
## Merc 240D   24.4   4 146.7  62 3.69 3.190 20.00 1   0    4    2
## Merc 230    22.8   4 140.8  95 3.92 3.150 22.90 1   0    4    2
## Fiat 128    32.4   4  78.7  66 4.08 2.200 19.47 1   1    4    1
## Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52 1   1    4    2
## 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
## Fiat X1-9   27.3   4  79.0  66 4.08 1.935 18.90 1   1    4    1
## Porsche 914-2 26.0  4 120.3  91 4.43 2.140 16.70 0   1    5    2
## Lotus Europa 30.4  4  95.1 113 3.77 1.513 16.90 1   1    5    2
## Volvo 142E  21.4  4 121.0 109 4.11 2.780 18.60 1   1    4    2
```

Now find the cars for which the MPG exceeds the 75% percentile value for MPG

```
mtcars[mtcars$mpg > quantile(mtcars$mpg)[4],]
```

```
##           mpg  cyl  disp  hp drat   wt  qsec vs  am  gear  carb
## Merc 240D   24.4   4 146.7  62 3.69 3.190 20.00 1   0    4    2
## Fiat 128    32.4   4  78.7  66 4.08 2.200 19.47 1   1    4    1
```

```
## Honda Civic      30.4   4  75.7  52 4.93 1.615 18.52  1  1   4   2
## Toyota Corolla  33.9   4  71.1  65 4.22 1.835 19.90  1  1   4   1
## Fiat X1-9        27.3   4  79.0  66 4.08 1.935 18.90  1  1   4   1
## Porsche 914-2    26.0   4 120.3  91 4.43 2.140 16.70  0  1   5   2
## Lotus Europa     30.4   4  95.1 113 3.77 1.513 16.90  1  1   5   2
```

2.6 Missing values

This is big deal. Most “real” data has rows that do not contain values for all columns. This is the so called “missing value” problem. Here is an example. The following code will read in a version of the mtcars data frame that has some missing values:

```
url <- "https://raw.githubusercontent.com/stevie42/utilities/master/data/mtcars_na.csv"
(mtcars_na <- read.csv(url, stringsAsFactors = FALSE))
```

```
##      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## 1  21.0   6  160.0  110 3.90 2.620 16.46  0  1   4    4
## 2  21.0   6  160.0  110 3.90   NA 17.02  0  1   4    4
## 3  22.8   4  108.0   93 3.85 2.320 18.61  1  1   4    1
## 4  21.4   6  258.0  110 3.08 3.215 19.44  1  0   3    1
## 5  18.7   8  360.0  175 3.15 3.440 17.02  0  0   3    2
## 6  18.1   6  225.0  105 2.76 3.460 20.22  1  0   3    1
## 7  14.3   8  360.0  245 3.21 3.570 15.84  0  0   3    4
## 8  24.4   4  146.7   62 3.69 3.190 20.00  1  0   4    2
## 9  22.8   4  140.8   95 3.92   NA 22.90  1  0   4    2
## 10 19.2   6  167.6  123 3.92 3.440 18.30  1  0   4   NA
## 11 17.8   6  167.6  123 3.92 3.440 18.90  1  0   4    4
## 12 16.4   8  275.8  180 3.07 4.070 17.40  0  0   3   NA
## 13 17.3   8  275.8  180 3.07 3.730 17.60  0  0   3    3
## 14 15.2   8  275.8  180 3.07 3.780 18.00  0  0   3    3
## 15 10.4   8  472.0  205 2.93 5.250 17.98  0  0   3    4
## 16 10.4   8  460.0  215 3.00 5.424 17.82  0  0   3    4
## 17 14.7   8  440.0  230 3.23 5.345 17.42  0  0   3    4
## 18 32.4   4   78.7   66 4.08 2.200 19.47  1  1   4    1
## 19 30.4   4   75.7   52 4.93 1.615 18.52  1  1   4   NA
## 20 33.9   4   71.1   65 4.22 1.835 19.90  1  1   4   NA
## 21 21.5   4  120.1   97 3.70 2.465 20.01  1  0   3    1
## 22 15.5   8  318.0  150 2.76 3.520 16.87  0  0   3    2
## 23 15.2   8  304.0  150 3.15   NA 17.30  0  0   3   NA
## 24 13.3   8  350.0  245 3.73 3.840 15.41  0  0   3    4
## 25 19.2   8  400.0  175 3.08 3.845 17.05  0  0   3    2
## 26 27.3   4   79.0   66 4.08 1.935 18.90  1  1   4    1
## 27 26.0   4  120.3   91 4.43 2.140 16.70  0  1   5    2
## 28 30.4   4   95.1  113 3.77 1.513 16.90  1  1   5   NA
```

```
## 29 15.8   8 351.0 264 4.22 3.170 14.50 0 1   5   4
## 30 19.7   6 145.0 175 3.62 2.770 15.50 0 1   5   6
## 31 15.0   8 301.0 335 3.54 3.570 14.60 0 1   5   8
## 32 21.4   4 121.0 109 4.11 2.780 18.60 1 1   4   2
```

If you look, you can see the missing values “NA” present in certain columns. This is R’s way of indicating what is missing. There are functions that can help you find these. This is important because, for example, if you wanted to find the average value of a column, say the `wt` column then there will be a problem as it contains a missing value:

```
mean(mtcars_na$wt)
```

```
## [1] NA
```

We have to tell the function to remove missing values from consideration.

```
mean(mtcars$wt, na.rm=TRUE)
```

```
## [1] 3.21725
```

A more general approach would involve the following functions.

```
complete.cases(mtcars_na)
```

```
## [1] TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE TRUE
## [12] FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE TRUE TRUE
## [23] FALSE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE
```

```
# How many rows in the df do not contain any NAs ?
```

```
sum(complete.cases(mtcars_na))
```

```
## [1] 24
```

```
# How many rows in the df do contain at least one NA ?
```

```
sum(!complete.cases(mtcars_na))
```

```
## [1] 8
```

How would we find those rows and print them ?

```
mtcars_na[complete.cases(mtcars_na),]
```

```
##      mpg  cyl  disp  hp drat   wt  qsec vs am gear carb
## 1  21.0    6 160.0 110 3.90 2.620 16.46 0 1   4   4
## 3  22.8    4 108.0  93 3.85 2.320 18.61 1 1   4   1
## 4  21.4    6 258.0 110 3.08 3.215 19.44 1 0   3   1
## 5  18.7    8 360.0 175 3.15 3.440 17.02 0 0   3   2
## 6  18.1    6 225.0 105 2.76 3.460 20.22 1 0   3   1
## 7  14.3    8 360.0 245 3.21 3.570 15.84 0 0   3   4
## 8  24.4    4 146.7  62 3.69 3.190 20.00 1 0   4   2
## 11 17.8    6 167.6 123 3.92 3.440 18.90 1 0   4   4
```

```
## 13 17.3 8 275.8 180 3.07 3.730 17.60 0 0 3 3
## 14 15.2 8 275.8 180 3.07 3.780 18.00 0 0 3 3
## 15 10.4 8 472.0 205 2.93 5.250 17.98 0 0 3 4
## 16 10.4 8 460.0 215 3.00 5.424 17.82 0 0 3 4
## 17 14.7 8 440.0 230 3.23 5.345 17.42 0 0 3 4
## 18 32.4 4 78.7 66 4.08 2.200 19.47 1 1 4 1
## 21 21.5 4 120.1 97 3.70 2.465 20.01 1 0 3 1
## 22 15.5 8 318.0 150 2.76 3.520 16.87 0 0 3 2
## 24 13.3 8 350.0 245 3.73 3.840 15.41 0 0 3 4
## 25 19.2 8 400.0 175 3.08 3.845 17.05 0 0 3 2
## 26 27.3 4 79.0 66 4.08 1.935 18.90 1 1 4 1
## 27 26.0 4 120.3 91 4.43 2.140 16.70 0 1 5 2
## 29 15.8 8 351.0 264 4.22 3.170 14.50 0 1 5 4
## 30 19.7 6 145.0 175 3.62 2.770 15.50 0 1 5 6
## 31 15.0 8 301.0 335 3.54 3.570 14.60 0 1 5 8
## 32 21.4 4 121.0 109 4.11 2.780 18.60 1 1 4 2
```

And here are the ones that do contain missing values:

```
mtcars_na[!complete.cases(mtcars_na),]
```

```
##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 2  21.0   6 160.0 110 3.90    NA 17.02 0 1  4   4
## 9  22.8   4 140.8  95 3.92    NA 22.90 1 0  4   2
## 10 19.2   6 167.6 123 3.92 3.440 18.30 1 0  4  NA
## 12 16.4   8 275.8 180 3.07 4.070 17.40 0 0  3  NA
## 19 30.4   4  75.7  52 4.93 1.615 18.52 1 1  4  NA
## 20 33.9   4  71.1  65 4.22 1.835 19.90 1 1  4  NA
## 23 15.2   8 304.0 150 3.15    NA 17.30 0 0  3  NA
## 28 30.4   4  95.1 113 3.77 1.513 16.90 1 1  5  NA
```

One quick way to omit rows with missing values is:

```
na.omit(mtcars_na)
```

```
##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1  21.0   6 160.0 110 3.90 2.620 16.46 0 1  4   4
## 3  22.8   4 108.0  93 3.85 2.320 18.61 1 1  4   1
## 4  21.4   6 258.0 110 3.08 3.215 19.44 1 0  3   1
## 5  18.7   8 360.0 175 3.15 3.440 17.02 0 0  3   2
## 6  18.1   6 225.0 105 2.76 3.460 20.22 1 0  3   1
## 7  14.3   8 360.0 245 3.21 3.570 15.84 0 0  3   4
## 8  24.4   4 146.7  62 3.69 3.190 20.00 1 0  4   2
## 11 17.8   6 167.6 123 3.92 3.440 18.90 1 0  4   4
## 13 17.3   8 275.8 180 3.07 3.730 17.60 0 0  3   3
## 14 15.2   8 275.8 180 3.07 3.780 18.00 0 0  3   3
## 15 10.4   8 472.0 205 2.93 5.250 17.98 0 0  3   4
## 16 10.4   8 460.0 215 3.00 5.424 17.82 0 0  3   4
```

```
## 17 14.7    8 440.0 230 3.23 5.345 17.42 0 0    3    4
## 18 32.4    4  78.7  66 4.08 2.200 19.47 1 1    4    1
## 21 21.5    4 120.1  97 3.70 2.465 20.01 1 0    3    1
## 22 15.5    8 318.0 150 2.76 3.520 16.87 0 0    3    2
## 24 13.3    8 350.0 245 3.73 3.840 15.41 0 0    3    4
## 25 19.2    8 400.0 175 3.08 3.845 17.05 0 0    3    2
## 26 27.3    4  79.0  66 4.08 1.935 18.90 1 1    4    1
## 27 26.0    4 120.3  91 4.43 2.140 16.70 0 1    5    2
## 29 15.8    8 351.0 264 4.22 3.170 14.50 0 1    5    4
## 30 19.7    6 145.0 175 3.62 2.770 15.50 0 1    5    6
## 31 15.0    8 301.0 335 3.54 3.570 14.60 0 1    5    8
## 32 21.4    4 121.0 109 4.11 2.780 18.60 1 1    4    2
```

2.7 Continuous vs Factors

One **recipe** that I use frequently is given below. This tells me how many unique values are assumed by each column which then helps to identify continuous quantities and categories. If a column assumes only a small number of unique values then perhaps it should be classified as a factor. Don't let the code here scare you. If you are new to R and don't yet understand what is going on then just use this as a "recipe" for now.

```
sapply(mtcars, function(x) length(unique(x)))
```

```
## mpg  cyl disp  hp drat   wt  qsec    vs  am gear carb
##   25    3  27   22  22   29   30     2   2    3    6
```

So it looks to me, for example, that **cyl**, **vs**, **am**, **gear**, and **carb** are actually categories rather than measured quantities. If you look at the help page for `mtcars` you will see that **am** is a 0 or 1 which corresponds to, respectively, a car with an automatic transmission (0) or a manual transmission (1). If you use the **summarize** function it will treat this variable as a numeric, continuous quantity.

Is it actually possible to have a transmission value of 0.4062 ?

```
summary(mtcars$am)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.0000  0.0000  0.0000  0.4062  1.0000  1.0000
```

I might then use some code to transform this into factors so that when they are used with various modeling functions they will be recognized as such. For example, if we summarize the data frame right now, we will see the following

```
summary(mtcars$am)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
## 0.0000 0.0000 0.0000 0.4062 1.0000 1.0000
```

Let's turn **am** into a factor

```
mtcars$am <- factor(mtcars$am,
                    levels = c(0,1),
                    labels = c("Auto","Man") )
```

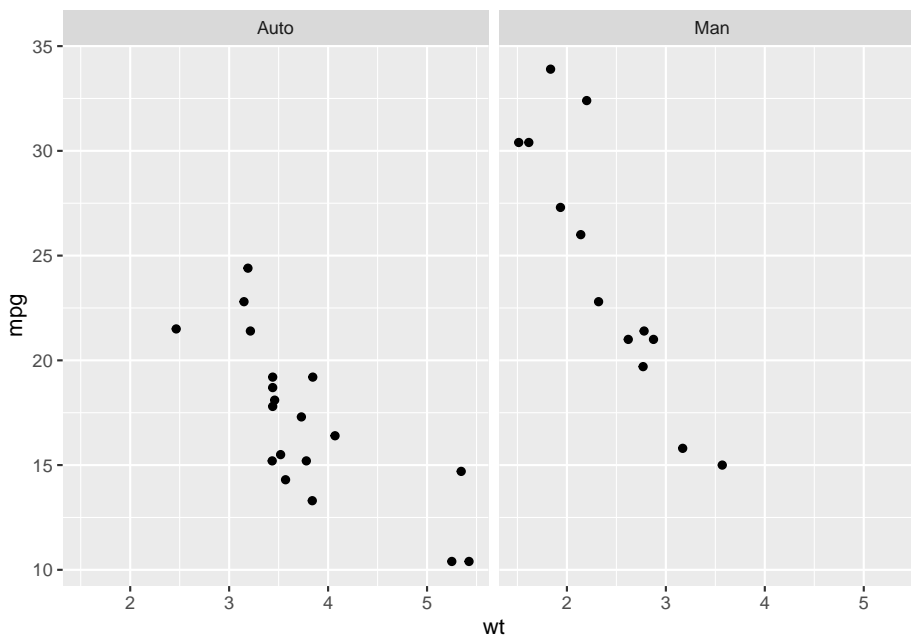
Now the summary will make more sense. This is also useful because graphics packages such as ggplot2 will know how to handle factors.

```
summary(mtcars$am)
```

```
## Auto  Man
```

```
##   19   13
```

```
ggplot(mtcars,aes(x=wt,y=mpg)) +
  geom_point() +
  facet_wrap(~am)
```



2.8 Sorting data

Sorting rows in a data frame is a common activity. However, in Base R this is called “ordering” because of the function used to “order” the data. Let's say we want to sort or “order” the mtcars data frame such that the row with the

lowest mpg value is listed first and the row with the highest mpg value is listed last. First, look at the **order** function's output. What are those numbers ?

```
order(mtcars$mpg)
```

```
## [1] 15 16 24 7 17 31 14 23 22 29 12 13 11 6 5 10 25 30 1 2 4 32 21
## [24] 3 9 8 27 26 19 28 18 20
```

Oh, so they are row numbers corresponding to rows in `mtcars`. Row 15 has the car with the lowest mpg. Row 16 corresponds to the car with the next lowest mpg and so on. So we can use this information to order our dataframe accordingly:

```
mtcars[order(mtcars$mpg),]
```

```
##          mpg cyl  disp  hp drat   wt  qsec vs  am gear carb
## Cadillac Fleetwood 10.4  8 472.0 205 2.93 5.250 17.98 0 Auto    3    4
## Lincoln Continental 10.4  8 460.0 215 3.00 5.424 17.82 0 Auto    3    4
## Camaro Z28         13.3  8 350.0 245 3.73 3.840 15.41 0 Auto    3    4
## Duster 360         14.3  8 360.0 245 3.21 3.570 15.84 0 Auto    3    4
## Chrysler Imperial  14.7  8 440.0 230 3.23 5.345 17.42 0 Auto    3    4
## Maserati Bora       15.0  8 301.0 335 3.54 3.570 14.60 0 Man     5    8
## Merc 450SLC         15.2  8 275.8 180 3.07 3.780 18.00 0 Auto    3    3
## AMC Javelin        15.2  8 304.0 150 3.15 3.435 17.30 0 Auto    3    2
## Dodge Challenger   15.5  8 318.0 150 2.76 3.520 16.87 0 Auto    3    2
## Ford Pantera L     15.8  8 351.0 264 4.22 3.170 14.50 0 Man     5    4
## Merc 450SE         16.4  8 275.8 180 3.07 4.070 17.40 0 Auto    3    3
## Merc 450SL         17.3  8 275.8 180 3.07 3.730 17.60 0 Auto    3    3
## Merc 280C          17.8  6 167.6 123 3.92 3.440 18.90 1 Auto    4    4
## Valiant            18.1  6 225.0 105 2.76 3.460 20.22 1 Auto    3    1
## Hornet Sportabout  18.7  8 360.0 175 3.15 3.440 17.02 0 Auto    3    2
## Merc 280           19.2  6 167.6 123 3.92 3.440 18.30 1 Auto    4    4
## Pontiac Firebird   19.2  8 400.0 175 3.08 3.845 17.05 0 Auto    3    2
## Ferrari Dino       19.7  6 145.0 175 3.62 2.770 15.50 0 Man     5    6
## Mazda RX4          21.0  6 160.0 110 3.90 2.620 16.46 0 Man     4    4
## Mazda RX4 Wag      21.0  6 160.0 110 3.90 2.875 17.02 0 Man     4    4
## Hornet 4 Drive     21.4  6 258.0 110 3.08 3.215 19.44 1 Auto    3    1
## Volvo 142E         21.4  4 121.0 109 4.11 2.780 18.60 1 Man     4    2
## Toyota Corona      21.5  4 120.1  97 3.70 2.465 20.01 1 Auto    3    1
## Datsun 710         22.8  4 108.0  93 3.85 2.320 18.61 1 Man     4    1
## Merc 230           22.8  4 140.8  95 3.92 3.150 22.90 1 Auto    4    2
## Merc 240D          24.4  4 146.7  62 3.69 3.190 20.00 1 Auto    4    2
## Porsche 914-2      26.0  4 120.3  91 4.43 2.140 16.70 0 Man     5    2
## Fiat X1-9          27.3  4  79.0  66 4.08 1.935 18.90 1 Man     4    1
## Honda Civic        30.4  4  75.7  52 4.93 1.615 18.52 1 Man     4    2
## Lotus Europa       30.4  4  95.1 113 3.77 1.513 16.90 1 Man     5    2
## Fiat 128           32.4  4  78.7  66 4.08 2.200 19.47 1 Man     4    1
```

```
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  Man    4    1
```

To invert the sense of the order use the **rev** function. We'll also use the **head** function to list only the first 5 rows of the result. Note that in base R, using composite functions is welcomed although you will find out that this is not a value in the tidyverse. For math people, using a composite function is natural which, in large part, is why R embraced that approach early on.

```
head(mtcars[rev(order(mtcars$mpg)),])
```

```
##           mpg cyl  disp  hp drat   wt  qsec vs  am  gear carb
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  Man    4    1
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  Man    4    1
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  Man    5    2
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  Man    4    2
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  Man    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  Man    5    2
```

2.9 Reading CSV Files

Many times data will be read in from a comma delimited file exported from Excel. These are known as Comma Separated Value files - generally abbreviated as **CSV**. The file can be read from a local drive or even from the Web as long as you know the URL associated with the file. In this example, there is a file on the Internet relating to some testing data involving students and various subjects.

```
url <- "https://raw.githubusercontent.com/pittardsp/bios545r_spring_2018/master/SUPPORTING_FILES/01%20Data%20Sets/01%20Data%20Sets.csv"
```

```
data1 <- read.csv(url,header=T,sep=",")
```

```
head(data1)
```

```
##    id female race ses schtyp prog read write math science socst
## 1  70      0    4   1     1    1   57    52   41      47     57
## 2 121      1    4   2     1    3   68    59   53      63     61
## 3  86      0    4   3     1    1   44    33   54      58     31
## 4 141      0    4   3     1    3   63    44   47      53     56
## 5 172      0    4   2     1    2   47    52   57      53     61
## 6 113      0    4   2     1    2   44    52   51      63     61
```


Chapter 3

The tidyverse

The **dplyr** package is part of the larger **tidyverse** package set which has expanded considerably in recent years and continues to grow in size and utility such that many people never learn the “older way” of doing things in R. But we’ve already been through that in the previous section. The tidyverse has the following packages. The descriptions have been lifted from the tidyverse home page.

ggplot2 - ggplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

dplyr - dplyr provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges.

tidyr - tidyr provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable.

readr - readr provides a fast and friendly way to read rectangular data (like csv, tsv, and fwf). It is designed to flexibly parse many types of data found in the wild, while still cleanly failing when data unexpectedly changes.

tibble - tibble is a modern re-imagining of the data frame, keeping what time has proven to be effective, and throwing out what it has not. Tibbles are data.frames that are lazy and surly: they do less and complain more forcing you to confront problems earlier, typically leading to cleaner, more expressive code.

stringr - stringr provides a cohesive set of functions designed to make working with strings as easy as possible. It is built on top of stringi, which uses the ICU C library to provide fast, correct implementations of common string manipulations.

lubdriate - Date-time data can be frustrating to work with in R. R commands for date-times are generally unintuitive and change depending on the type of date-time object being used. Moreover, the methods we use with date-times must be robust to time zones, leap days, daylight savings times, and other time related quirks. Lubridate makes it easier to do the things R does with date-times and possible to do the things R does not.

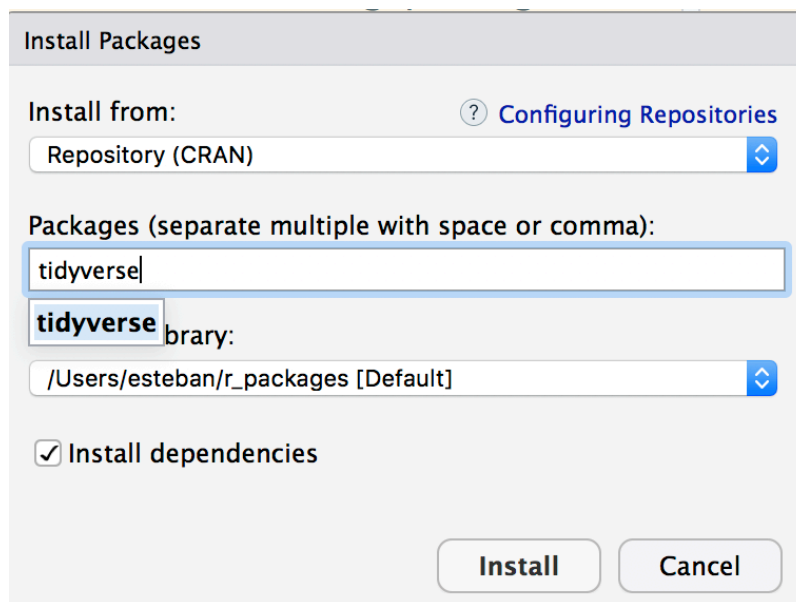
3.1 Installing

You will probably use a number of functions from several of these packages so it's best to go ahead and install the entire **tidyverse** in one go. To install it, do one of the following:

- 1) At the R Console from within RStudio, type:

```
install.packages("tidyverse")
```

- 2) Use the Tools -> Install Packages menu item in RStudio:



After you have installed the package you may load it by doing:

```
suppressMessages(library(tidyverse))
```

Note that the **cheatsheet** for **dplyr** can be found [here](#)

Data Transformation with dplyr : : CHEAT SHEET

3.2 dplyr Basics

dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges. In fact if you were paying attention during the opening section on data frames you will have noticed that most of the activities we performed related to the following activities. In dplyr-speak there are the **verbs** that help us get work done.

```
mutate() - adds new variables that are functions of existing variables
select() - picks variables based on their names.
filter() - picks cases based on their values.
summarise() - reduces multiple values down to a single summary.
arrange() - changes the ordering of the rows.
```

3.3 First Steps

Note that this material references “Becoming a data ninja with dplyr” as well as this dplyr tutorial

We’ll go back to the basics here by using a very small data frame which will make it clear how the various **dplyr** verbs actually work:

```
df <- data.frame(id = 1:5,
                 gender = c("MALE", "MALE", "FEMALE", "MALE", "FEMALE"),
                 age = c(70, 76, 60, 64, 68))
```

ID	GENDER	AGE
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

3.3.1 filter()

The **filter()** function allows us to sift through the data frame to find rows that satisfy some logical condition. (With the older approach we would be using the bracket notation). The following example allows us to find only the observations relating to a declared gender of **female**.

```
filter(df, gender == "FEMALE")
```

```
##   id gender age
## 1  3 FEMALE 60
## 2  5 FEMALE 68
```

Given this data frame, the following is equivalent

```
filter(df, gender != "MALE")
```

```
##   id gender age
## 1  3 FEMALE 60
## 2  5 FEMALE 68
```

ID	GENDER	AGE
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	AGE
3	FEMALE	60
5	FEMALE	68

So, now find only the **ids** that relate to rows 1,3, or 5. This is a highly specialized search but it is helpful to show that you can use a wide variety of logical constructs.

```
filter(df, id %in% c(1,3,5))
```

```
##   id gender age
## 1  1  MALE  70
## 2  3 FEMALE  60
## 3  5 FEMALE  68
```

ID	GENDER	AGE
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

ID	GENDER	AGE
1	MALE	70
3	FEMALE	60
5	FEMALE	68

3.3.2 mutate()

Mutate is used to add or remove columns in a data frame. Let's create a new column in the data frame that contains the mean value of the age column.

```
mutate(df, meanage = mean(age))
```

```
##   id gender age meanage
## 1  1  MALE  70    67.6
## 2  2  MALE  76    67.6
## 3  3 FEMALE  60    67.6
## 4  4  MALE  64    67.6
## 5  5 FEMALE  68    67.6
```

ID	GENDER	AGE	MEANWT
1	MALE	70	67.6
2	MALE	76	67.6
3	FEMALE	60	67.6
4	MALE	64	67.6
5	FEMALE	68	67.6

Next we will create a new column designed to tell us if a given observation has an age that is greater than or equal to the average age. Specifically, create a variable called **old_young** and assign a value of “Y” if the observed age for that row is above the mean age and a value of “N” if it is not.

```
mutate(df,old_young=ifelse(df$age>=mean(df$age),"Y","N"))
```

```
##   id gender age old_young
## 1  1  MALE  70         Y
## 2  2  MALE  76         Y
## 3  3 FEMALE  60         N
## 4  4  MALE  64         N
## 5  5 FEMALE  68         Y
```

One way we could use something like this is in making a plot where the observations exhibiting an age value above the mean are plotted in a certain color and those below the mean are in another color.

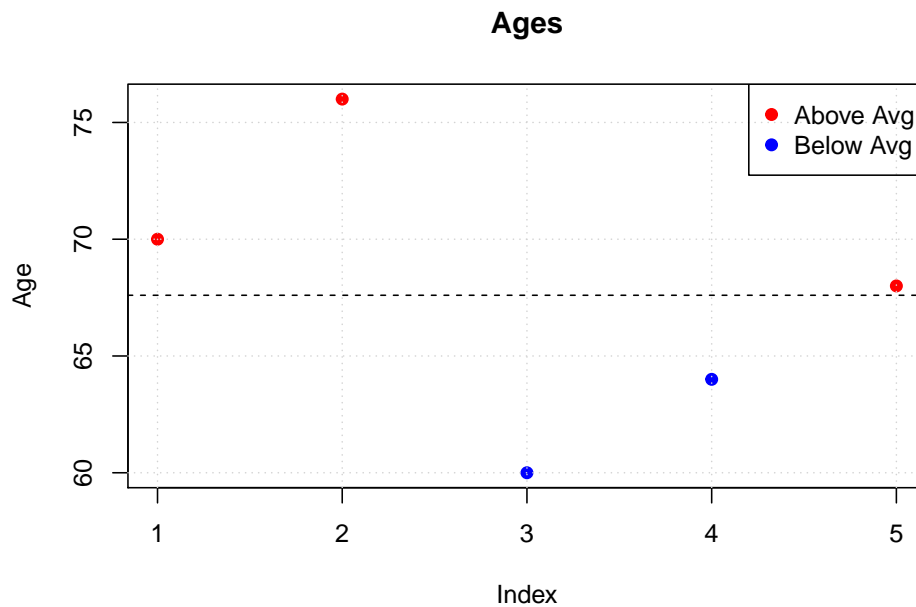
```
tmp <- mutate(df, color = ifelse(age > mean(age),"red","blue"))

plot(tmp$age,col=tmp$color, type="p",
     pch=19,main="Ages",ylab="Age")

grid()

abline(h=mean(tmp$age),lty=2)

legend("topright",
     c("Above Avg","Below Avg"),col=c("red","blue"),pch=19)
```



3.3.3 arrange()

Use `arrange` for sorting the data frame by one or more columns. When using the basic data frame structure from R we had to use the `order()` function to help us generate a vector that has the row numbers of the data frame that correspond to the desired order of display (lowest to highest, etc).

Let's sort the data frame `dff` by age from oldest to youngest. First we'll use the older approach. While this will work, it is not exactly very intuitive.

```
df[rev(order(df$age)),]
```

```
##   id gender age
## 2  2  MALE  76
## 1  1  MALE  70
## 5  5 FEMALE  68
## 4  4  MALE  64
## 3  3 FEMALE  60
```

`dplyr` makes this process more simple - at least in my opinion

```
arrange(df, desc(age))
```

```
##   id gender age
## 1  2  MALE  76
## 2  1  MALE  70
## 3  5 FEMALE  68
```

```
## 4 4 MALE 64
## 5 3 FEMALE 60
```

Next, let's sort **df** by gender (alphabetically) and then by age from oldest to youngest. The rows relating to a gender of **female** are going to be listed first because, alphabetically speaking, the letter “F” comes before the letter “M”. Then within those categories we have the ages sorted from oldest to youngest.

```
arrange(df, gender, desc(age))
```

```
##   id gender age
## 1  5 FEMALE 68
## 2  3 FEMALE 60
## 3  2 MALE 76
## 4  1 MALE 70
## 5  4 MALE 64
```

If we used the older approach it would look like the following. Ugh !

```
df[order(df$gender, -df$age),]
```

```
##   id gender age
## 5  5 FEMALE 68
## 3  3 FEMALE 60
## 2  2 MALE 76
## 1  1 MALE 70
## 4  4 MALE 64
```

3.3.4 select()

The `select()` functions allows us to select one or more columns from a data frame.

```
# Reorder the columns
select(df, gender, id, age)
```

```
##   gender id age
## 1 MALE 1 70
## 2 MALE 2 76
## 3 FEMALE 3 60
## 4 MALE 4 64
## 5 FEMALE 5 68
```

```
# Select all but the age column
select(df, -age)
```

```
##   id gender
## 1 1 MALE
## 2 2 MALE
## 3 3 FEMALE
```



```
## 4 4 MALE
## 5 5 FEMALE

# Can use the ":" operator to select a range
select(df,id:age)
```

```
##   id gender age
## 1  1  MALE  70
## 2  2  MALE  76
## 3  3 FEMALE  60
## 4  4  MALE  64
## 5  5 FEMALE  68
```

The `select()` function provides the ability to select by “regular expressions” or numeric patterns:

```
# Select all columns that start with an "a"
select(df,starts_with("a"))
```

```
##   age
## 1  70
## 2  76
## 3  60
## 4  64
## 5  68
```

```
names(mtcars)
```

```
## [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
## [11] "carb"
```

```
# Get only columns that start with "c"
select(mtcars,starts_with("c"))
```

```
##           cyl carb
## Mazda RX4         6    4
## Mazda RX4 Wag     6    4
## Datsun 710         4    1
## Hornet 4 Drive     6    1
## Hornet Sportabout  8    2
## Valiant           6    1
## Duster 360        8    4
## Merc 240D         4    2
## Merc 230          4    2
## Merc 280          6    4
## Merc 280C         6    4
## Merc 450SE        8    3
## Merc 450SL        8    3
```

```
## Merc 450SLC      8    3
## Cadillac Fleetwood 8    4
## Lincoln Continental 8    4
## Chrysler Imperial 8    4
## Fiat 128         4    1
## Honda Civic      4    2
## Toyota Corolla   4    1
## Toyota Corona    4    1
## Dodge Challenger 8    2
## AMC Javelin      8    2
## Camaro Z28       8    4
## Pontiac Firebird  8    2
## Fiat X1-9        4    1
## Porsche 914-2    4    2
## Lotus Europa     4    2
## Ford Pantera L   8    4
## Ferrari Dino     6    6
## Maserati Bora    8    8
## Volvo 142E       4    2
```

This example is more realistic in that data frames can have a large number of columns named according to some convention. For example, the measurements on a patient might not be labelled specifically - they might have a common prefix such as “m_” followed by some sequential number (or not).

```
testdf <- expand_grid(m_1=seq(60,70,10),
                      age=c(25,32),
                      m_2=seq(50,60,10),
                      m_3=seq(60,70,10))
```

```
testdf
```

```
##   m_1 age m_2 m_3
## 1   60  25  50  60
## 2   70  25  50  60
## 3   60  32  50  60
## 4   70  32  50  60
## 5   60  25  60  60
## 6   70  25  60  60
## 7   60  32  60  60
## 8   70  32  60  60
## 9   60  25  50  70
## 10  70  25  50  70
## 11  60  32  50  70
## 12  70  32  50  70
## 13  60  25  60  70
## 14  70  25  60  70
```

```
## 15 60 32 60 70
## 16 70 32 60 70
```

Find all the columns that include a “_” character

```
select(testdf,matches("_"))
```

```
##      m_1 m_2 m_3
## 1    60  50  60
## 2    70  50  60
## 3    60  50  60
## 4    70  50  60
## 5    60  60  60
## 6    70  60  60
## 7    60  60  60
## 8    70  60  60
## 9    60  50  70
## 10   70  50  70
## 11   60  50  70
## 12   70  50  70
## 13   60  60  70
## 14   70  60  70
## 15   60  60  70
## 16   70  60  70
```

This will select columns beginning with “m_” but only those with a suffix of 1 or 2.

```
select(testdf,num_range("m_",1:2))
```

```
##      m_1 m_2
## 1    60  50
## 2    70  50
## 3    60  50
## 4    70  50
## 5    60  60
## 6    70  60
## 7    60  60
## 8    70  60
## 9    60  50
## 10   70  50
## 11   60  50
## 12   70  50
## 13   60  60
## 14   70  60
## 15   60  60
## 16   70  60
```

3.3.5 group_by()

The **group_by()** function lets you organize a data frame by some factor or grouping variable. This is a very powerful function that is typically used in conjunction with a function called **summarize**. Here is what it looks like by itself. It's somewhat underwhelming. It does seem to create a table of some kind but it doesn't do much else.

```
df
```

```
##   id gender age
## 1  1  MALE  70
## 2  2  MALE  76
## 3  3 FEMALE  60
## 4  4  MALE  64
## 5  5 FEMALE  68
```

Hmm. the following doesn't do anything - or so it seems

```
group_by(df)
```

```
## # A tibble: 5 x 3
##       id gender   age
##   <int> <fct> <dbl>
## 1     1  MALE    70
## 2     2  MALE    76
## 3     3 FEMALE   60
## 4     4  MALE    64
## 5     5 FEMALE   68
```

So as mentioned, the **group_by** function is usually paired with the **summarize** function. Ah, so what this does is to first group the data frame by the **gender** column and then it **counts** the number of occurrences therein. So this is a form of aggregation.

```
summarize(group_by(df,gender),total=n())
```

```
## # A tibble: 2 x 2
##   gender total
##   <fct> <int>
## 1 FEMALE     2
## 2 MALE      3
```

ID	GENDER	AGE
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

GENDER	TOTAL
FEMALE	2
MALE	3

Let's group the data frame by gender and then compute the average age for each group.

```
summarize(group_by(df,gender),av_age=mean(age))
```

```
## # A tibble: 2 x 2
##   gender av_age
##   <fct>   <dbl>
## 1 FEMALE    64
## 2 MALE     70
```

ID	GENDER	AGE
1	MALE	70
2	MALE	76
3	FEMALE	60
4	MALE	64
5	FEMALE	68

GENDER	AV_AGE
FEMALE	64
MALE	70

Let's group by gender and then compute the total number of observations in each gender group and then compute the mean age.

```
summarize(group_by(df,gender),av_age=mean(age),total=n())
```

```
## # A tibble: 2 x 3
##   gender av_age total
##   <fct>   <dbl> <int>
## 1 FEMALE    64     2
## 2 MALE     70     3
```


Chapter 4

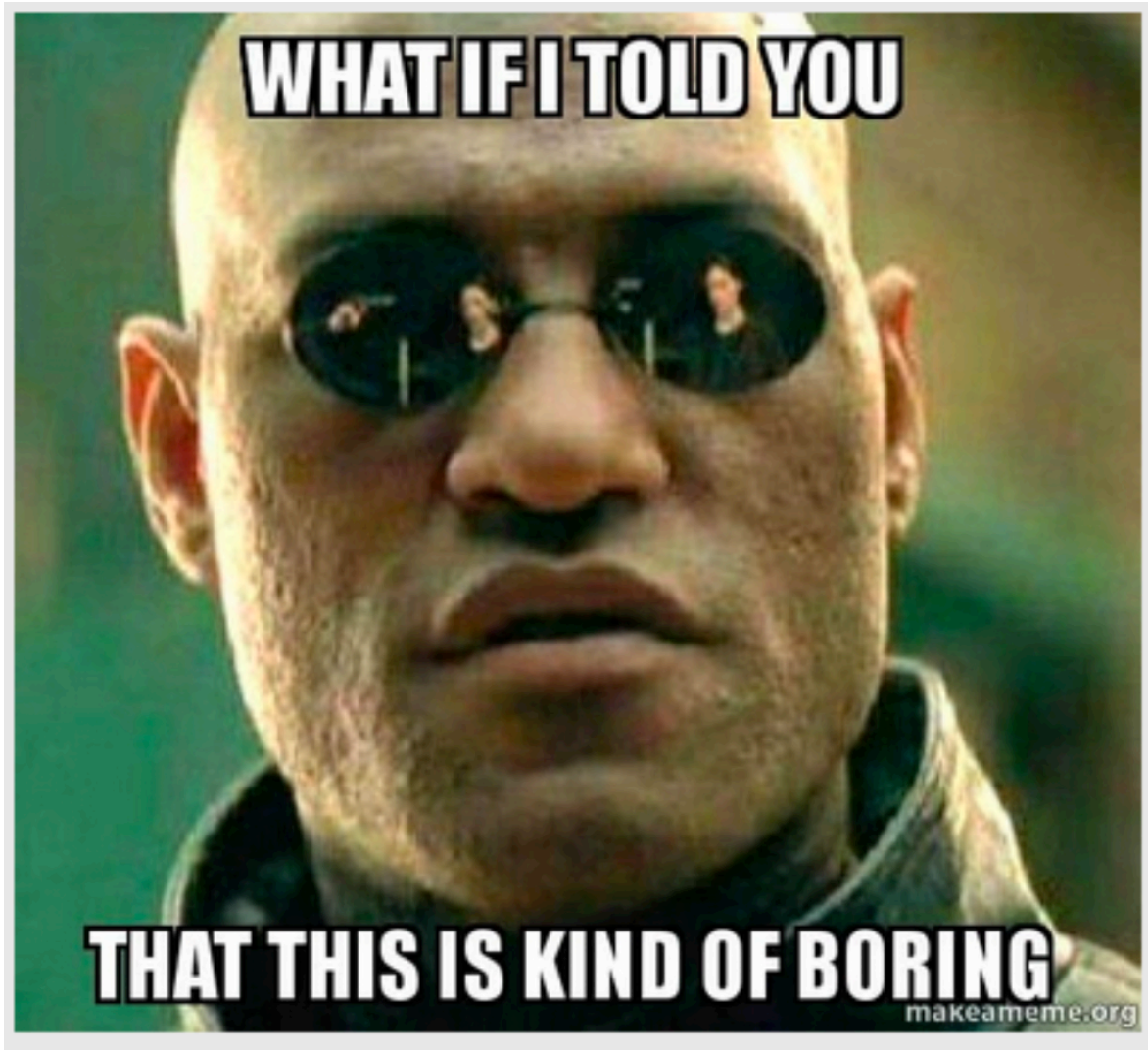
Split Apply Combine

This pattern of using `group_by()` followed by `summarize()` is called **Split Apply Combine**. The idea is that we

- 1) Split up the data frame by gender group
- 2) Then for each group, apply the average function
- 3) Then combine the average results with each group

```
group_by(df,gender)
```

```
## # A tibble: 5 x 3
## # Groups:   gender [2]
##       id gender  age
##   <int> <fct> <dbl>
## 1     1  MALE    70
## 2     2  MALE    76
## 3     3 FEMALE   60
## 4     4  MALE    64
## 5     5 FEMALE   68
```



4.1 What Are Pipes ?

Before moving forward let us consider the “pipe” operator that is included with the dplyr - well actually **magrittr** package. This is used to make it possible to “pipe” the results of one command into another command and so on.

The inspiration for this comes from the UNIX/LINUX operating system where pipes are used all the time. So in effect using “pipes” is nothing new in the world of research computation. **Warning:** Once you get used to pipes it is hard to go back to not using them.

Let's use the `mtcars` data frame to illustrate the basics of the piping mechanism as used by `dplyr`. Here we will select the **mpg** and **am** column from `mtcars` and view the top 5 rows.

```
head(select(mtcars, mpg, am))
```

```
##           mpg    am
## Mazda RX4      21.0  Man
## Mazda RX4 Wag  21.0  Man
## Datsun 710     22.8  Man
## Hornet 4 Drive  21.4 Auto
## Hornet Sportabout 18.7 Auto
## Valiant        18.1 Auto
```

Okay, how would we do this using pipes ? Whoa !

```
mtcars %>% select(mpg, am) %>% head
```

```
##           mpg    am
## Mazda RX4      21.0  Man
## Mazda RX4 Wag  21.0  Man
## Datsun 710     22.8  Man
## Hornet 4 Drive  21.4 Auto
## Hornet Sportabout 18.7 Auto
## Valiant        18.1 Auto
```

Break this down:

```
mtcars %>% select(mpg, am)
```

```
##           mpg    am
## Mazda RX4      21.0  Man
## Mazda RX4 Wag  21.0  Man
## Datsun 710     22.8  Man
## Hornet 4 Drive  21.4 Auto
## Hornet Sportabout 18.7 Auto
## Valiant        18.1 Auto
## Duster 360     14.3 Auto
## Merc 240D      24.4 Auto
## Merc 230       22.8 Auto
## Merc 280       19.2 Auto
## Merc 280C      17.8 Auto
## Merc 450SE     16.4 Auto
## Merc 450SL     17.3 Auto
## Merc 450SLC    15.2 Auto
## Cadillac Fleetwood 10.4 Auto
## Lincoln Continental 10.4 Auto
## Chrysler Imperial 14.7 Auto
## Fiat 128       32.4  Man
```

```
## Honda Civic          30.4  Man
## Toyota Corolla       33.9  Man
## Toyota Corona        21.5  Auto
## Dodge Challenger     15.5  Auto
## AMC Javelin          15.2  Auto
## Camaro Z28           13.3  Auto
## Pontiac Firebird     19.2  Auto
## Fiat X1-9            27.3  Man
## Porsche 914-2        26.0  Man
## Lotus Europa         30.4  Man
## Ford Pantera L       15.8  Man
## Ferrari Dino         19.7  Man
## Maserati Bora        15.0  Man
## Volvo 142E           21.4  Man
```

The key to understanding how this works is to **read** this from left to right. Let’s use our new found knowledge to reimagine our use of the **group_by** and **summarize** functions.

4.2 Pipes and Split Apply Combine

```
df %>% group_by(gender) %>% summarize(avg=mean(age))
```

```
## # A tibble: 2 x 2
##   gender  avg
##   <fct> <dbl>
## 1 FEMALE    64
## 2 MALE      70
```

Same as this but the pipes are far more elegant

```
summarize(group_by(df,gender), avg=mean(age))
```

```
## # A tibble: 2 x 2
##   gender  avg
##   <fct> <dbl>
## 1 FEMALE    64
## 2 MALE      70
```

This approach allows us to build a “pipeline” containing commands. We don’t have to commit to a specific sequence. This enables a free-form type of exploration.

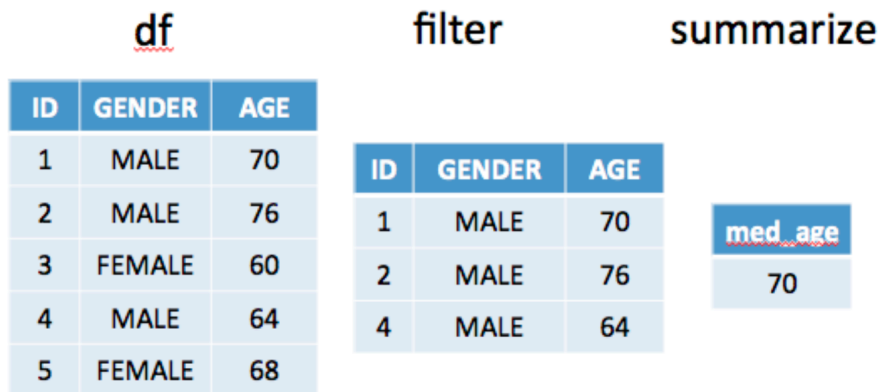
```
df %>%
  group_by(gender) %>%
  summarize(avg=mean(age),total=n())
```

```
## # A tibble: 2 x 3
##   gender  avg total
##   <fct>  <dbl> <int>
## 1 FEMALE    64     2
## 2 MALE      70     3
```

What is the median age of all males ?

```
df %>%
  filter(gender == "MALE") %>%
  summarize(med_age=median(age))
```

```
##   med_age
## 1      70
```



Using the built in mtcars dataframe, **filter** for records where the wt is greater than 3.3 tons.

Then, using the **mutate** function, create a column called ab_be (Y or N) that indicates whether that observation's mpg is greater (or not) than the average mpg for the filtered set.

Then present the average mpg for each group. This is easy using pipes and dplyr verbs

```
mtcars %>% filter(wt > 3.3) %>%
  mutate(ab_be=ifelse(mpg > mean(mpg), "Y", "N")) %>%
  group_by(ab_be) %>%
  summarize(mean_mpg=mean(mpg))
```

```
## # A tibble: 2 x 2
##   ab_be mean_mpg
##   <chr>    <dbl>
## 1 N      13.8
## 2 Y      18.1
```