Module02: Data with mostly continuous variables

Steve Simon

Created 2020-02-08

The tidyverse library

```
library(tidyverse)
## -- Attaching packages ------
------------------------------
-- tidyverse 1.3.1 --
## v ggplot2 3.3.5     v purrr
0.3.4
## v tibble  3.1.6     v dplyr
1.0.8
## v tidyr   1.2.0     v
stringr 1.4.0
```

The tidyverse package is a collection of several different packages which provide enhancements to the R programming language. These libraries share a common programming philosophy. There are several dozen libraries in total, but only a core set of libraries are loaded with the library(tidyverse) function. Other tidyverse packages must be loaded separately.

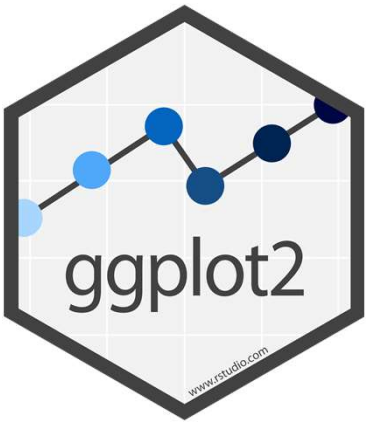
I recommend that you use the tidyverse library for all your programs in this class. Here are some of the libraries in core set of libraries.

dplyr

Hex sticker for dplyr

dplyr provides a set of functions for data manipulation.

ggplot2



Hex sticker for ggplot2

While R has some excellent graphics capabilities built in, they are somewhat difficult to use. The ggplot2 library simplifies the process of graphing by separating the parts of a graph into different layers. It is based on a conceptual framework developed by Leland Wilkinson in his book, The Grammar of Graphics.

magrittr

Hex sticker for magrittr

magrittr provides a pipe operator. The concept of the pipe was developed first in Unix systems almost 50 years ago. The pipe operator (percent-greater than-percent) takes input from the left side of the operator and feeds it to a function listed on the right side of the operator. Pipes can be chained together. They make your code simpler and more readable.

We may or may not cover pipes in this class.

readr

Hex sticker for readr

While R has many functions for reading text data, they are slow for very large files. The readr library reads text files much faster, offers some enhancements, and provides a simpler syntax.

stringr

Hex sticker for stingr

stringr simplifies the manipulation of string or text data.

tibble

Hex sticker for tibble

R has a variety of internal storage formats: arrays, lists, matrices, and data frames. We will focus mostly on data frames in this class. The tibble package offers an internal storage format, a tibble, that is very similar to a data frame, but it offers some extra features for convenience and simplicity.

tidyr

Hex sticker for tidyr

tidyr provides a series of functions that help with data manipulation, especially for longitudinal data.

Other packages in the tidyverse

- In the core package
  - forcats
  - purr
- Outside the core package
  - broom
  - lubridate
  - readxl
  - many others

Two other packages in the tidyverse core, forcats and purr, are for advanced applications.

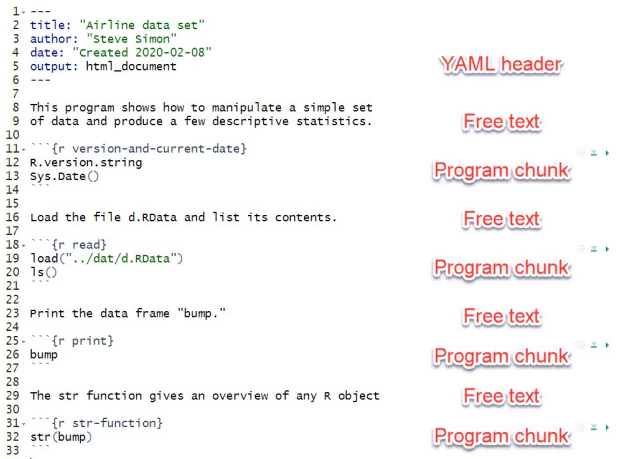
Outside of the core package, some of the packages that I like are broom (which simplifies and standardizes the output from different data analysis functions) lubridate (which simplifies the manipulaton of dates), and readxl (which reads Microsoft Excel files). There are quite a few others.

Load tidyverse quietly

```
suppressMessages(
  suppressWarnings(
    library(tidyverse)))
options(width=29)
```

Because tidyverse provides so many diagnostic messages and warnings, I have used the suppressMessages and suppressWarnings functions to avoid this in my programs. For this program, I need two more things. I need the yaml package to simplify the presentation of information from the data dictionary. I also cut the line width down to 45 characters. This is rather extreme, but it is necessary when you are producing PowerPoint documents.

General structure of Rmarkdown



Recall the general structure of an Rmarkdown file. It starts with a header, in a simple format known as YAML. Then you alternate between free text commentary and chunks of R programming statements. The chunks are surrounded by lines with three backticks and the opening set of three backticks is followed by the programming language (R) and an optional label for the chunk.

Documentation

```
R.version.string
## [1] "R version 4.1.2 (2021-
11-01)"
Sys.Date()
## [1] "2022-02-20"
```

You should always print out the version of R that you are using and the date that the program was run. This is one of the documentation practices that you should get in the habit of using.

Break #1

- What have you learned
  - The tidyverse packages
  - Rmarkdown structure
- What is coming next
  - Printing a data frame
  - Printing pieces of a data frame

Reading existing data

```
load("../data/module02-
datasets.RData")
ls()
## [1] "bump"      "burgers"
## [3] "encoding"  "fat"
## [5] "inputFile" "sleep"
```

After loading data, use the ls function to review what information was loaded.

From burgers-data-dictionary.yaml

(From the original website) “Fast food is often considered unhealthy because muchof it is high in both fat and sodium. But are the two related? The data give the fat and sodium contents of several brands of burgers.”

The first dataset that we will use in this video is “burgers.”

Here is the first paragraph from the data dictionary.

It’s a very small dataset, which I chose deliberately so that it would fit well on the PowerPoint slides.

Displaying data

```
burgers
  Fat Sodium Calories
1  19    920      410
2  31   1500      580
3  34   1310      590
4  35    860      570
5  39   1180      640
6  39    940      680
7  43   1260      660
```

R has a print function, but if you just list the name of a data frame, it will, by default, display itself.

Showing the structure

```
glimpse(burgers)
Rows: 7
Columns: 3
$ Fat      <dbl> 19, 31, 34~
$ Sodium   <dbl> 920, 1500,~
$ Calories <dbl> 410, 580, ~
```

The “glimpse” function provides a description of “burgers.” It is a data frame. A data frame is a rectangular grid of data. The data values in a single column of data has to be consistent: either a set of numbers, a set of dates, a set of character strings, etc. But the type of data in one column can differ from the next column.

Data frames are the workhorse of R. There are other ways of storing data: lists, arrays, vectors. But most of the data you will analyze in R will be stored in a data frame.

Displaying a single value

```
burgers[1, 1]
[1] 19
burgers[7, 3]
[1] 660
```

You can display individual data points by specifying the row and column inside square brackets.

Displaying a single row

```
burgers[1, ]
  Fat Sodium Calories
1  19    920      410
burgers[7, ]
  Fat Sodium Calories
7  43   1260      660
```

You can display an entire row by specifying it and leaving the column entry blank.

Displaying a single column (1/2)

```
burgers[ , 1]
[1] 19 31 34 35 39 39 43
```

You can display an entire column similarly.

Displaying a single column (2/2)

```
burgers[ , 3]
[1] 410 580 590 570 640 680
[7] 660
```

There is an important difference in the display here. When you display a single column, it no longer keeps the structure of a data frame. It becomes a vector, which is a single dimensional quantity. You lose the name associated with the column and it prints the results horizontally, even though the column prints inside the data frame vertically.

This may seem to be a trivial matter and but once in a while it is not. The distinction may come up later in this class.

Using column names

```
burgers[ , "Fat"]
[1] 19 31 34 35 39 39 43
burgers$Fat
[1] 19 31 34 35 39 39 43
```

Each column has a name and you can use that name instead of the column number.

Using individual columns is so common that R developed a shorthand for it, using the dollar sign.

Displaying multiple rows or columns

```
burgers[1:3, ]
  Fat Sodium Calories
1  19    920      410
2  31   1500      580
3  34   1310      590
```

The code “1:3” means 1, 2, 3. Notice that when you select two columns rather than one that it maintains the dataframe structure.

This again represents a subtle distinction, but one that can sometimes be important.

Displaying multiple rows or columns

```
burgers[ , c(1, 3)]
  Fat Calories
1  19      410
2  31      580
3  34      590
4  35      570
5  39      640
6  39      680
7  43      660
```

The “c” function allows you to select values that are not in a perfect sequence.

Skipping a row or column

```
burgers[ , -1]
  Sodium Calories
1    920      410
2   1500      580
3   1310      590
4    860      570
5   1180      640
6    940      680
7   1260      660
```

A negative number means that you want to exclude that row or column. In this example, the -1 allows you to print everything except the first column.

The head function

```
head(burgers)
  Fat Sodium Calories
1  19    920      410
2  31   1500      580
3  34   1310      590
4  35    860      570
5  39   1180      640
6  39    940      680
```

The “head” function displays the first five rows of data.

The tail function

```
tail(burgers)
  Fat Sodium Calories
2  31   1500      580
3  34   1310      590
4  35    860      570
5  39   1180      640
6  39    940      680
7  43   1260      660
```

The “tail” function displays the last five rows of data. If there is a problem with a data set, it often shows up more obviously at the bottom of a data frame than the top.

Now using “head” or “tail” on a dataset as small as this one is silly, but these two functions are very useful for larger datasets.

Break #2

- What have you learned
  - Printing a data frame
  - Printing pieces of a data frame
- What is coming next
  - Definitions of categorical and continuous data
  - The “fat” data set
  - Outliers and missing values

Some definitions

- Categorical = small number of possible values
- Examples
  - Sex (Male or Female),
  - Race/ethnicity (Caucasian, African American, Hispanic, etc.),
  - Cancer stage (I, II, III, or IV),
  - Birth delivery type (Vaginal, C-section).

A **categorical variable** is a variable that can only take on a small number of values. Each value is usually associated with a particular category.

Some definitions

- Continuous variable = large number of possible values
- Examples of continuous variables are
  - Birth weight in grams,
  - Gestational age,
  - Fasting LDL level.

A **continuous variable** is a variable that can take on a large number of possible values, potentially any value in some interval.

There are some variables that are on the boundary between categorical and continuous, but it is not worth quibbling about here.

The point to remember is that the types of graphs that you use and the types of statistics that you compute are dependent on many things, but first and foremost on whether the variables are categorical, continuous, or a mixture.

Today, you will see examples involving mostly continuous variables.

From fat-data-dictionary.yaml

This dataset includes two measures of body fat (a quantity that is normally quite difficult to measure) along with some simpler measures of body size that could be used to predict body fat.

Here is the first paragraph from the data dictionary.

First few lines of data

```
fat[1:5, 1:4]
##   case fat_b fat_s density
## 1    1  12.6  12.3  1.0708
## 2    2   6.9   6.1  1.0853
## 3    3  24.6  25.3  1.0414
## 4    4  10.9  10.4  1.0751
## 5    5  27.8  28.7  1.0340
```

Here are the first few lines of the data set.

Glimpse of the data

```
glimpse(fat)
## Rows: 252
## Columns: 19
## $ case    <dbl> 1, 2, 3, 4,~
## $ fat_b   <dbl> 12.6, 6.9, ~
## $ fat_s   <dbl> 12.3, 6.1, ~
## $ density <dbl> 1.0708, 1.0~
## $ age     <dbl> 23, 22, 22,~
## $ wt      <dbl> 154.25, 173~
```

Here is a glimpse at the structure of the data.

Some simple rules for data frames

- Rectangular grid
- Different types across columns
- Single type within a column
- Alternative ways to store data
  - Vector
  - Matrix
  - Array
  - List

R has many of the features of an object-oriented language, but it is not a true object-oriented programming language. There are a variety of objects in R like vectors, lists, matrices, and arrays, that are useful for storing, manipulating, and analyzing research data. We will spend most of this class using a particular object, the data frame.

The object, fat, that you just created with the read.table function is a data frame. Data frames are rectangular grids of data. Each column in the data frame has the same length. A data frame can store data of various types (numeric, character, and dates are the most common types of data). The data within a column has to have the same type, but the different columns can have different data types.

There are times when the rectangular grid of a data frame is too restrictive for your data, and R has other ways of storing this data (most notably, lists), but you will find that for most data analyses, a data frame will work just fine.

The head function shows the first few rows of the data set and the tail function shows the last few rows of the data set.

Always get in the habit of checking out the very bottom of your data frame. It's a

common location for glitches.

Variable names

- Short but descriptive
- Mix of letters and numbers
  - Must start with a letter
  - Avoid most symbols
- No blanks
  - CamelCase
  - dot.delimited.names
  - underscore_delimited_names

This data set did not have a header, a line at the very top of the file that lists variable names. R uses the default names V1, V2, etc. As a general rule, you should use brief (but descriptive) names for every variable in your data set. The names should be around 8 characters long. Longer variable names make your typing tedious and much shorter variable names makes your code terse and cryptic.

You should avoid special symbols in your variable names especially symbols that are likely to cause confusion (the dash symbol, for example, which is also the symbol for subtraction). You should also avoid blanks. These rules are pretty much universal across most statistical software packages. If you violate these rules, you will find out that, at a minimum, you will always have to surround your variable name by quotes to avoid problems.

There are times when you'd like to have a blank in your variable name and you can use two special symbols that you can use in R (and most other statistical pacakges), the underscore symbol (above the minus key on most keyboards) and the dot (period). These symbols create some artificial spacing that mimics the blanks. Another approach is "CamelCase" which is the mixture of upper and lower case within a variable name with each uppercase designating the beginning of a new

“word”.

Break #3

- What have you learned
  - Definitions of categorical and continuous data
  - The “fat” data set
- What is coming next
  - Handling outliers
  - Tracking missing values
  - Histograms

Unusual data value

```
summary(fat$ht)
##    Min. 1st Qu.  Median
##   29.50   68.25   70.00
##    Mean 3rd Qu.    Max.
##   70.15   72.25   77.75
```

There is an unusual data value, which you might not notice right away, but one of the heights is 29.5 inches. We'll talk in more detail about the summary function later, but right now I wanted to show you function because if you have an outlier in your data, you are most likely to discover it by using the summary function.

A height this small is not totally out of the realm of possibility. See, for example,

–> http://en.wikipedia.org/wiki/List_of_shortest_people

You can use the which function to identify the row with this unusual value for further investigation. Note the use of the double equals sign and how you display a single row of a data frame.

Which function

```
short_row <-
  which(fat$ht==29.5)
short_row
## [1] 42
```

Displaying the unusual row

```
fat[short_row, ]
##    case fat_b fat_s density
## 42   42  31.7  32.9   1.025
##    age  wt   ht  bmi   ffw
## 42  44 205 29.5 29.9 140.1
##    neck chest abdomen   hip
## 42 36.6   106   104.3 115.5
##    thigh knee ankle biceps
## 42  70.6 42.5  23.7   33.6
```

The other values look quite normal. You have to make a careful choice here. One possibility is to do nothing. If you leave the abnormal height in your data set, it may distort some of your graphs and skew some of your statistics. Still, it is often BETTER than some of the alternatives.

Remove the outlier

```
fat1 <- fat[-short_row, ]
summary(fat1$ht)
##    Min. 1st Qu.  Median
##   64.00   68.25   70.00
##    Mean 3rd Qu.    Max.
##   70.31   72.25   77.75
```

A second choice is to remove the entire row from the data frame. The - means everything EXCEPT that row.

Set to missing

```
fat2 <- fat
fat2[short_row,"ht"] <- NA
summary(fat2$ht)
##    Min. 1st Qu.  Median
##   64.00   68.25   70.00
##    Mean 3rd Qu.    Max.
##   70.31   72.25   77.75
##    NA's
##       1
```

A third possibility is to designate the abnormal value as missing. In R, a missing value is represented by NA.

Notice that the summary function for the ht variable notes that one of the values is missing. You should watch these missing values obsessively. This can get a bit tricky.

There is no one method that is preferred in this setting. If you encounter an unusual value, you should discuss it with your research team, investigate the original data sources, if possible, and review any procedures for handling unusual data values that might be specified in your research protocol.

Your data set may arrive with missing values in it already. Data might be designated as missing for a variety of reasons (lab result lost, value below the limit of detection, patient refused to answer this question) and how you handle missing values is way beyond the scope of this class. Just remember to tread cautiously around missing values as they are a minefield.

Notice that I store the revised data sets with the row removed and with the 29.5 replaced by a missing value in different data frames. This is good programming

practice. If you ever have to make a destructive change to your data set (a change that wipes out one or more values or a change that is difficult to undo), it is good form to store the new results in a fresh spot. That way, if you get cold feet, you can easily backtrack.

We'll use the data set with the 29.5 changed to a missing value for most of the remaining analyses of this data set.

Break #4

- What have you learned
  - Handling outliers
- What is coming next
  - Tracking missing values

You cannot test missingness directly

```
which(fat2$ht==NA)
## integer(0)
```

Logic involving missing values is tricky. If you checking for equality and one of the things in the comparison is missing, then the result is neither TRUE, not FALSE, but rather missing.

Fair enough, but R takes it a bit further, and if both sides when you are checking for equality are missing, then they might both be 5 is they weren't missing or maybe one might be 5 and the other one 10. So it might be TRUE or it might be FALSE, so we're better off calling the logical result as missing.

This is called a three valued logic system and it has advantages and disadvantages. I won't get into any technical details, except to say that you should never make assumptions. Check what you do when you are working with missing values to make sure that the three valued logic system doesn't produce results that you didn't expect.

Use is.na to test missingness

```
which(is.na(fat2$ht))
## [1] 42
mean(fat2$ht)
## [1] NA
sd(fat2$ht)
## [1] NA
```

The short term solution to the above problem is to use a special function, is.na.

The summary function will trap and remove missing values, but most other functions in R will, by default, report a result as missing if any values going into that function are missing. The philosophy in R, I suppose, is that you need to actively select an approach for handling missing values rather than relying on a lazy default.

R is also erring on the side of caution most of the time. You may not be aware that there are missing values lurking in your data, and R is going to go out of its way to remind you, unless you tell it otherwise.

This is different from SAS and SPSS, where the default options choose perfectly reasonable approaches, but approaches that don't raise concern about missingness to the degree that R does.

Read the help file for these functions (enter ?mean or ?sd at the command prompt).

Look carefully and note that the na.rm option allows you to compute the statistic after missing values are removed.

Using the na.rm argument

```
mean(fat2$ht,na.rm=TRUE)
## [1] 70.31076
sd(fat2$ht,na.rm=TRUE)
## [1] 2.614296
```

For univariate functions, there are only two realistic ways to handle missing values, but for bivariate and multivariate function, there are a multitude of approaches, such as pairwise deletion, listwise deletion, last observation carried forward, single imputation, and multiple imputation. There is a lot of controversy over various methods for handling missing values.

Remember to round your results

```
round(
  mean(
    fat2$ht,
    na.rm=TRUE), 1)
## [1] 70.3
round(
  sd(
    fat2$ht,
    na.rm=TRUE), 1)
## [1] 2.6
```

For univariate functions, there are only two realistic ways to handle missing values, but for bivariate and multivariate function, there are a multitude of approaches, such as pairwise deletion, listwise deletion, last observation carried forward, single imputation, and multiple imputation. There is a lot of controversy over various methods for handling missing values.

Break #5

- What have you learned
  - Tracking missing values
- What is coming next
  - Histograms

Histogram with default options - code

```
histogram_white <-
  ggplot(fat, aes(x=ht)) +
    geom_histogram(
      binwidth=1,
      fill="white",
      color="black")
```

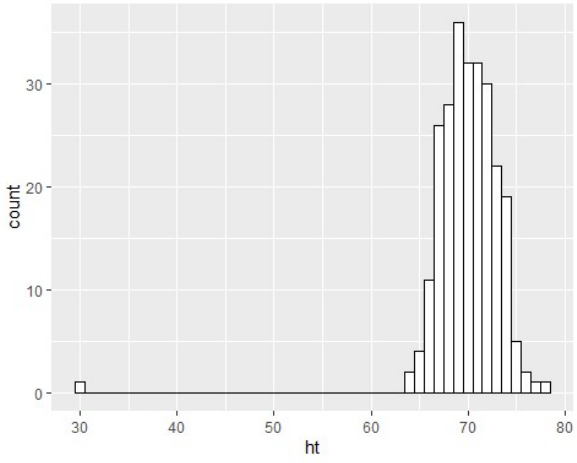
A histogram is useful for displaying a continuous variable graphically.

It is very important to explicitly choose the widths of your bars. It's a bit too busy, perhaps, but I chose bar widths of 1.

The default colors in ggplot2, black bars with a black border, are terrible. At a minimum, you should use a different color for the inside (fill) and border (color). Here is how you draw white bars with a black border.

Please note that I cannot easily add speaker notes to a slide with a graph on it. So I will typically comment about the graph before I display it.

Histogram with default options - graph



Histogram with nice labels - code

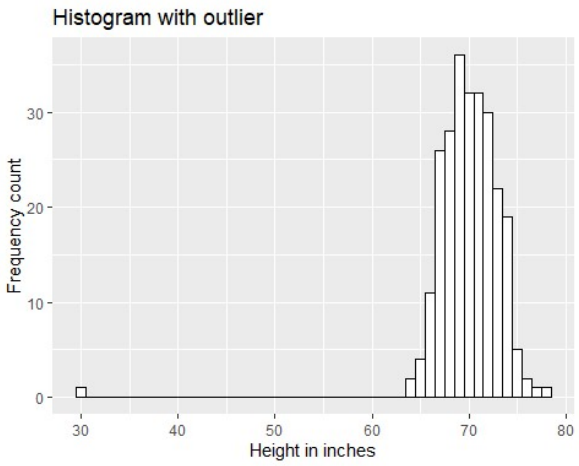
```
histogram_labeled <-
  histogram_white +
    xlab("Height in inches") +
    ylab("Frequency count") +
    ggtitle("Histogram with
outlier")
```

This code shows how to put nice labels on the X and Y axes along with a title.

There is an important pattern to note here. Graphs created with ggplot2 can be built with layers. This makes programming a bit easier and your code is also more readable.

The next slide shows what the labeled histogram looks like.

Histogram with nice labels - graph



Break #6

- What have you learned
  - Histograms
- What is coming next
  - Correlations

Correlation coefficients

– Correlation
  - Always between -1 and 1
  - Strong positive if > 0.7
  - Strong negative if < -0.7
  - Weak positive if between 0.3 and 0.7
  - Weak negative if between -0.3 and -0.7
  - No relationship if between -0.3 and 0.3

The correlation coefficient is a single number between -1 and +1 that quantifies the strength and direction of a relationship between two continuous variables. As a rough rule of thumb, a correlation larger than +0.7 indicates a strong positive association and a correlation smaller than -0.7 indicates a strong negative association. A correlation between +0.3 and +0.7 (-0.3 and -0.7) indicates a weak positive (negative) association. A correlation between -0.3 and +0.3 indicates little or no association.

Don't take these rules too literally. You're not trying to make definitive statements about your data set. You are just trying to get comfortable with some general patterns that occur in your data set. A complex and definitive statistical analysis will almost certainly not agree with at least some of the preliminary correlations noted here.

Correlation coefficient

```
cor(fat2$fat_b,fat2$age)
## [1] 0.2891735
```

You can get a matrix of correlations for every possible pair of variables. This command becomes a bit more complicated if there are some categorical variables in your data set, as you need to exclude these prior to calculating the correlation matrix. Since you are just trying to get a general feel for your data, a bit of rounding will help you. Also, you should remove the case number before calculating the correlation matrix.

Always round your results

```
round(cor(fat2$fat_b,fat2$age),
2)
## [1] 0.29
```

You can get a matrix of correlations for every possible pair of variables. This command becomes a bit more complicated if there are some categorical variables in your data set, as you need to exclude these prior to calculating the correlation matrix. Since you are just trying to get a general feel for your data, a bit of rounding will help you. Also, you should remove the case number before calculating the correlation matrix.

Correlation matrix - code

```
correlation_matrix <-
  round(cor(fat2[ ,-1]), 2)
dim(correlation_matrix)
## [1] 18 18
```

Notice that the -1 in fat2 omits the first column, which is the case number, a variable that should not be incorporated in any statistical analyses.

I don't want to display the full correlation matrix because it would go off the margins of the slide. But looking at just the first eight rows and two columns, you can see something interesting. R does not know how to compute a correlation coefficient when there are missing values.

Correlation matrix - results

```
##          fat_b fat_s
## fat_b     1.00  1.00
## fat_s     1.00  1.00
## density -0.99 -0.99
## age      0.29  0.29
## wt       0.61  0.61
## ht         NA    NA
## bmi      0.73  0.73
## ffw      0.02  0.02
```

I don’t want to display the full correlation matrix because it would go off the margins of the slide. But looking at just the first eight rows and two columns, you can see something interesting. R does not know how to compute a correlation coefficient when there are missing values.

Excerpt from the help file

use an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings `"everything"`, `"all.obs"`, `"complete.obs"`, `"na.or.complete"`, or `"pairwise.complete.obs"`.

If `use` is `"everything"`, `NA`s will propagate conceptually, i.e., a resulting value will be `NA` whenever one of its contributing observations is `NA`. If `use` is `"all.obs"`, then the presence of missing observations will produce an error. If `use` is `"complete.obs"` then missing values are handled by casewise deletion (and if there are no complete cases, that gives an error).

`"na.or.complete"` is the same unless there are no complete cases, that gives `NA`. Finally, if `use` has the value `"pairwise.complete.obs"` then the correlation or covariance between each pair of variables is computed using all complete pairs of observations on those variables. This can result in covariance or correlation matrices which are not positive semi-definite, as well as `NA` entries if there are no complete pairs for that pair of variables. For `cov` and `var`, `"pairwise.complete.obs"` only works with the `"pearson"` method. Note that (the equivalent of) `var(double(0), use = *)` gives `NA` for `use = "everything"` and `"na.or.complete"`, and gives an error in the other cases.

This is two excerpts from the help file. The documentation is quite confusing, but that is because the options here are quite confusing. The two interesting options are casewise deletion (complete.obs) and pairwise deletion (pairwise.complete.obs). For the former, the row with missing values is tossed out for every correlation calculation, so that each and every correlation uses 251 rows of data rather than 252. For the latter, most correlations use all 252 rows of data, but any correlation involving height uses only 251 rows of data.

Which choice is best is beyond the scope of this class.

Correlation with pairwise deletion - code

```
correlation_matrix <-
  round(
    cor(fat2[ ,-1],
      use="pairwise"), 2)
```

Here is what the correlation matrix looks like with pairwise deletion. Notice that you can abbreviate “pairwise-complete-obs” to just “pairwise”. Actually, any abbreviation that is not ambiguous would work, so you could say “pair” or even just “p” here since none of the other choices for the use option begins with the letter “p.”

Correlation with pairwise deletion - results

```
##          fat_b fat_s
## fat_b     1.00  1.00
## fat_s     1.00  1.00
## density -0.99 -0.99
## age      0.29  0.29
## wt       0.61  0.61
## ht      -0.02 -0.02
## bmi      0.73  0.73
## ffw      0.02  0.02
```

Here is what the correlation matrix looks like with pairwise deletion. Notice that you can abbreviate “pairwise-complete-obs” to just “pairwise”. Actually, any abbreviation that is not ambiguous would work, so you could say “pair” or even just “p” here since none of the other choices for the use option begins with the letter “p.”

Different parts of the correlation matrix (1/3)

```
correlation_matrix[10:12, 1:2]
##          fat_b fat_s
## chest     0.70  0.70
## abdomen   0.81  0.81
## hip       0.63  0.63
```

Rounding the correlations makes them easier to read. Here are three different parts of the correlation matrix, showing the circumference measurements versus the fat measurements.

It's something you might not notice easily if you didn't round the data.

Here is the correlation of fat measures with measures of the central part of the body.

Different parts of the correlation matrix (2/3)

```
correlation_matrix[13:15, 1:2]
##        fat_b fat_s
## thigh   0.56  0.56
## knee    0.51  0.51
## ankle   0.27  0.27
```

Here is the correlation with measures at various parts of the leg.

Different parts of the correlation matrix (3/3)

```
correlation_matrix[16:18, 1:2]
##          fat_b fat_s
## biceps    0.49  0.49
## forearm   0.36  0.36
## wrist     0.35  0.35
```

Here is the correlations with measures at various parts of the parts of the arm.

The interesting pattern is that the strongest correlations are measurements near the middle (abdomen, chest, hip) and the correlation goes down as you move to the extremities. The lowest correlations are at the forearm, wrist, and ankle.

Break #7

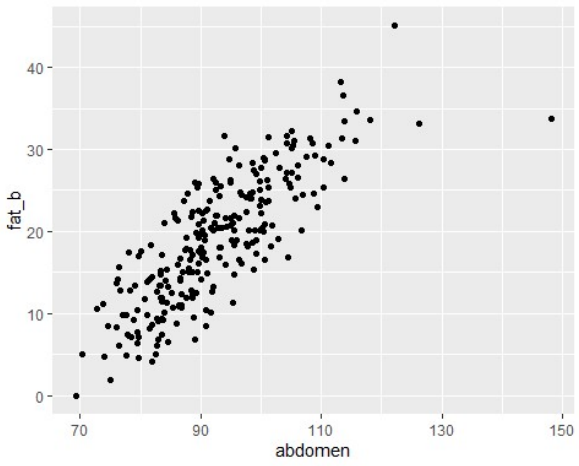
- What have you learned
  - Histograms
- What is coming next
  - Correlations
  - Scatterplots

Simple scatterplot - code

```
scatterplot_default <-
  ggplot(fat2,
    aes(x=abdomen, y=fat_b)) +
      geom_point()
```

A simple and commonly used graphical approach to showing a relationship between two continuous variables is a scatterplot.

Simple scatterplot - graph



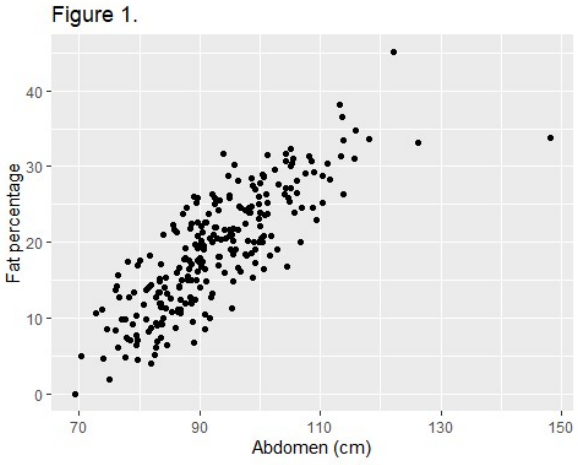
Scatterplot with labels - code

```
scatterplot_enhanced <-
scatterplot_default +
  xlab("Abdomen (cm)") +
  ylab("Fat percentage") +
  ggtitle("Figure 1.")
```

There are lots of options available to customize your graph. Here are just a few. The xlab and ylab arguments in the plot function control what is displayed on the horizontal (x) and vertical (y) axes. The pch argument control what is used as the plotting character.

Here are some adaptations, including better labels, on the following slide.

Scatterplot with labels - graph



Scatterplot with a trend line - code

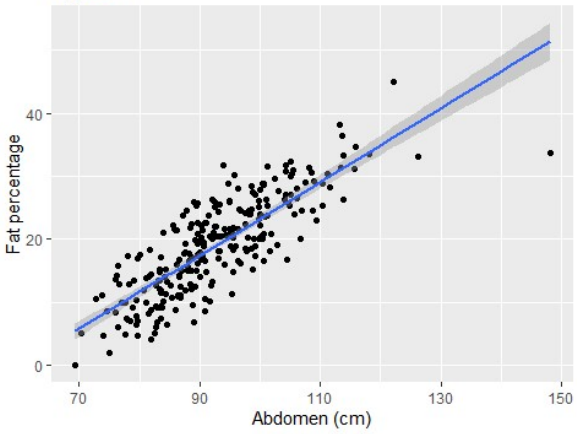
```
scatterplot_lm <-
scatterplot_enhanced +
  geom_smooth(method=lm)
```

Here's the code to produce a smooth curve on top of the data points. The graph is on the following slide.

Scatterplot with a linear trend line - graph

```
## `geom_smooth()` using
formula 'y ~ x'
```

Figure 1.



Scatterplot with smoothing - code

```
scatterplot_loess <-
scatterplot_enhanced +
  geom_smooth(method=loess)
```

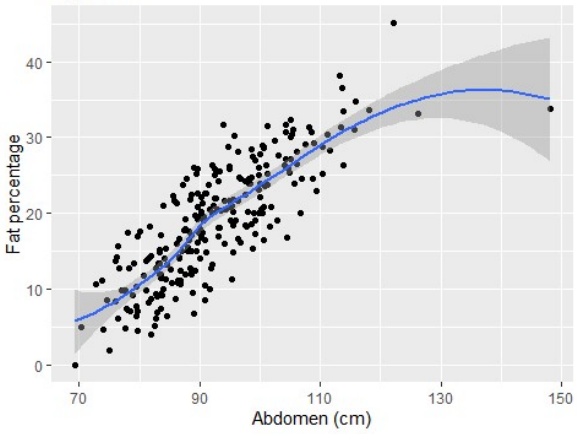
Here's the code to produce a smooth curve on top of the data points.

Always be cautious about smooth curves and splines at the left and right edges of the graph. These are often highly variable, because there is less data to use at the extremes. So the downward dip you see on the right hand side of the graph on the following page is driven by just a few data points.

Scatterplot with smoothing - graph

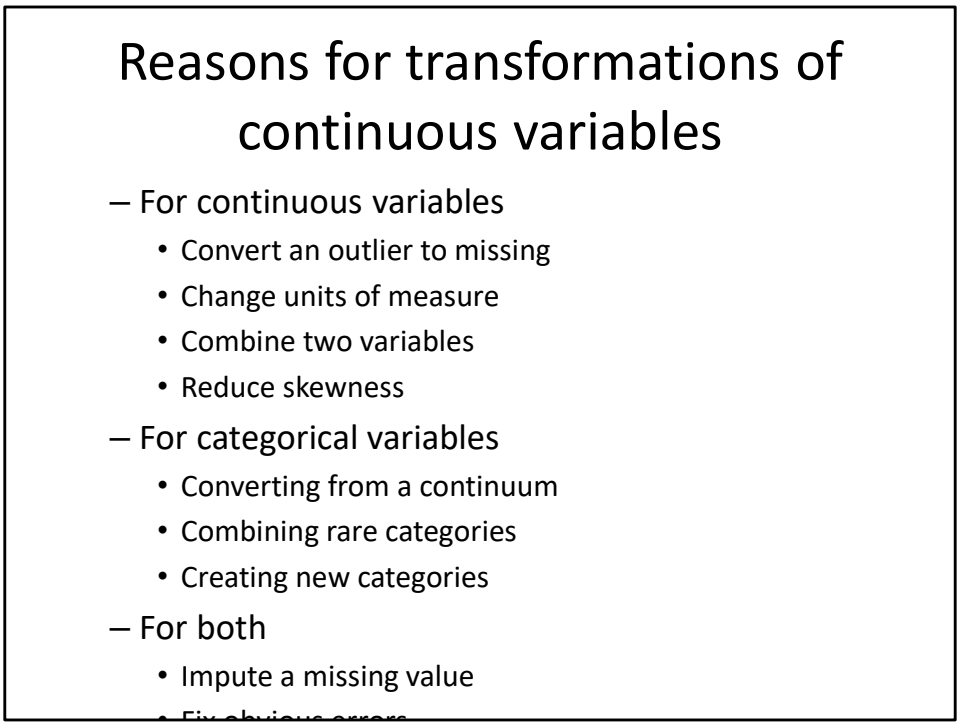
```
## `geom_smooth()` using
formula 'y ~ x'
```

Figure 1.



Break #8

- What have you learned
  - Scatterplots
- What is coming next
  - Transformations



Transformations are an important part of data management. The reasons are different for continuous and categorical variables, For continuous variables, you've already seen how to convert an outlier to a missing value. You might also transform a variable to change its units of measurement. An example would be moving measurements from the English system to the metric system. Sometimes you might combine two variables to create a third. An example might be calculation of the body mass index which is the weight divided by the height squared. Another common transformation is using logarithms to reduce skewness in a variable.

For categorical data, you might convert a continuous variable into categories. As an example, you might convert an age variable so that any vaule under 18 is categorized as a child and 18 and above is categorized as an adult. Often some categories occur so infrequently that you need to combine them. The standard categories for race, as an example, might have the vast majority being Black, and you would combine White, Asian, and Native American into an "Other" category. You could create new categories, such as using separate race and ethnicity variables into a combined race/ethnicity variable.

For both you might impute a missing value. In studies of smoking cessation, for

example, it is common to treat dropouts as treatment failures. So a missing value for smoking yes/no, could be converted to yes for anyone who drops out midway through the study.

Finally, some errors are so obvious that it is easy to fix them. You have a LIkert scale 1 through 5 and you notice a value of 11. Clearly this is a typo and should be 1.

Avoid transformations not easily undone.

- Bad
  - fat[short_row, “ht”] <- NA
- Good

```
fat2 <- fat
fat2[short_row, "ht"] <- NA
```

- Also good

```
fat$ht_trimmed <- fat$ht
fat$ht_trimmed[short_row] <- NA
```

It is good programming practice to avoid destructive transformations. A destructive transformation is one that is not easily undone. You might decide at a later date that you don’t want a particular transformation. Or perhaps you want a different transformation.

You can avoid destrictuve transformations by creating a new dataframe or creating a new variable within an existing dataframe.

Units conversion

```
metric <- fat
metric$ht <- fat$ht/39.37
metric$wt <- fat$wt/2.205
round(mean(metric$ht), 1)
## [1] 1.8
round(mean(fat$ht), 1)
## [1] 70.1
```

If you look at the data dictionary

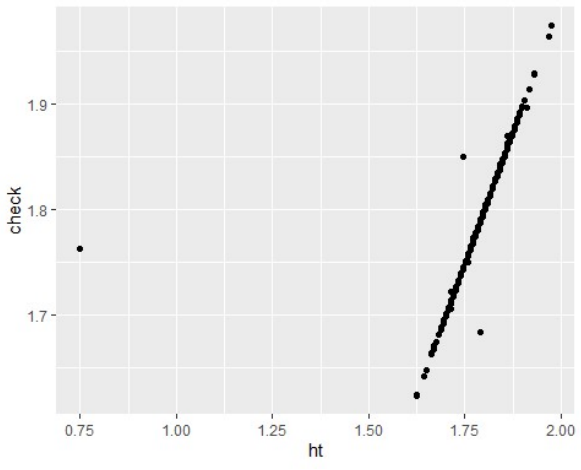
Combining two variables - code

```
metric$check <-
  sqrt(metric$wt / metric$bmi)
height_comparisons <-
  ggplot(metric,
    aes(x=ht, y=check)) +
      geom_point()
```

Usually, you use weight and height to calculate body mass index, but you can invert that calculation to compute height from weight and bmi. This allows us to see what the value for the unusual height of 29.5 inches (70 centimeters) might be if the weight and bmi values were correct.

The graph on the following slide shows the height values compared to the back calculated height values.

Combining two variables - graph



Two more errors?

```
metric$delta <-
  abs(metric$ht-metric$check)
odd_rows <-
  which(metric$delta > 0.1)
```

Two more errors?

```
round(metric[odd_rows , c(
  "ht",
  "check",
  "wt",
  "bmi")], 2)
##       ht check    wt  bmi
## 42  0.75  1.76 92.97 29.9
## 163 1.75  1.85 83.56 24.4
## 221 1.79  1.68 69.50 24.5
```

Conclusion

– What have you learned?
  - The tidyverse packages
  - Printing pieces of a data frame
  - Tracking missing values
  - Histograms
  - Correlations
  - Scatterplots
  - Transformations