Chapter 1

The data cleaning process

Which of the following is NOT an essential part of the data cleaning process as outlined in the previous video?

Answer: Preparing data for analysis

Here's what messy data look like

In the final chapter of this course, you will be presented with a messy, real-world dataset containing an entire year's worth of weather data from Boston, USA. Among other things, you'll be presented with variables that contain column names, column names that should be values, numbers coded as character strings, and values that are missing, extreme, and downright erroneous!

We've placed some R code in the script to the right. Run the code as-is to see just how messy the weather data really are!

```
# View the first 6 rows of data
head(weather)

# View the last 6 rows of data
tail(weather)

# View a condensed summary of the data
str(weather)
```

_

Here's what clean data look like

In this course, you will acquire many new tools in your data cleaning toolbox for whipping the weather data into shape!

Run the code provided to see what the weather dataset will look like by the time you are done cleaning it. If it's not immediately clear what's changed, don't worry! You will have a much deeper understanding by the end of this course.

```
# View the first 6 rows of data
head(weather_clean)

# View the last 6 rows of data
tail(weather_clean)

# View a condensed summary of the data
str(weather_clean)
```

_

Getting a feel for your data

The first thing to do when you get your hands on a new dataset is to understand its structure. There are several ways to go about this in R, each of which may reveal different issues with your data that require attention.

In this course, we are only concerned with data that can be expressed in table format (i.e. two dimensions, rows and columns). As you may recall from earlier courses, tables in R often have the type data.frame. You can check the class of any object in R with the class() function.

Once you know that you are dealing with tabular data, you may also want to get a quick feel for the contents of your data. Before printing the entire dataset to the console, it's probably worth knowing how many rows and columns there are. The dim() command tells you this.

We've loaded a dataset called bmi into your workspace. The data, which give the (age standardized) mean body mass index (BMI) among males in each country for the years 1980-2008, come from the School of Public Health, Imperial College London.

- Check the class of bmi
- Find the dimensions of bmi
- Print the bmi column names

 # Check the class of bmi
class(bmi)

 # Check the dimensions of bmi
dim(bmi)

 # View the column names of bmi
names(bmi)

_

Viewing the structure of your data

Since bmi doesn't have a huge number of columns, you can view a quick snapshot of your data using the str() (for structure) command. In addition to the class and dimensions of your entire dataset, str() will tell you the class of each variable and give you a preview of its contents.

Although we won't go into detail on the dplyr package in this lesson (see the Data Manipulation in R with dplyr course), the glimpse() function from dplyr is a slightly cleaner alternative to str(). str() and glimpse() give you a preview of your data, which may reveal issues with the way columns are labelled, how variables are encoded, etc.

You can use the summary() command to get a better feel for how your data are distributed, which may reveal unusual or extreme values, unexpected missing data, etc. For numeric variables, this means looking at means, quartiles (including the median), and extreme values. For character or factor variables, you may be curious about the number of times each value appears in the data (i.e. counts), which summary() also reveals.

- View the structure of bmi using the traditional method

- Load the dplyr package
- View the structure of bmi using dplyr
- Look at a summary() of bmi

```
# Check the structure of bmi
str(bmi)

# Load dplyr
library(dplyr)

# Check the structure of bmi, the dplyr way
glimpse(bmi)

# View a summary of bmi
summary(bmi)
```

_

Looking at your data

You can look at all the summaries you want, but at the end of the day, there is no substitute for looking at your data -- either in raw table form or by plotting it.

The most basic way to look at your data in R is by printing it to the console. As you may know from experience, the print() command is not even necessary; you can just type the name of the object. The downside to this option is that R will attempt to print the entire dataset, which can be a nuisance if the dataset is too large.

One way around this is to use the head() and tail() commands, which only display the first and last 6 rows of data, respectively. You can view more (or fewer) rows by providing as a second argument to the function the number of rows you wish to view. These functions provide a useful method for quickly getting a sense of your data without overly cluttering the console.

- Print the full dataset to the console (you don't need print() to do this)
- View the first 6 rows of bmi
- View the first 15 rows of bmi
- View the last 6 rows of bmi
- View the last 10 rows of bmi

```
# Print bmi to the console
print(bmi)

# View the first 6 rows
head(bmi)

# View the first 15 rows
head(bmi, 15)

# View the last 6 rows
tail(bmi)

# View the last 10 rows
tail(bmi, 10)
```

_

Visualizing your data

There are many ways to visualize data. Since this is not a course about data visualization, we will only touch on two types of plots that may be useful for quickly identifying extreme or suspicious values in your data: histograms and scatter plots.

A histogram, created with the hist() function, takes a vector (i.e. column) of data, breaks it up into intervals, then plots as a vertical bar the number of instances within each interval. A scatter plot, created with the plot() function, takes two vectors (i.e. columns) of data and plots them as a series of (x, y) coordinates on a two-dimensional plane.

Let's look at a quick example of each.

For the bmi dataset:

- Use hist() to look at the distribution of average BMI across all countries in 2008
- Use plot() to see how each country's average BMI in 1980 (x-axis) compared with its BMI in 2008 (y-axis)

```
# Histogram of BMIs from 2008
hist(bmi$Y2008)

# Scatter plot comparing BMIs from 1980 to those from 2008
plot(bmi$Y1980,bmi$Y2008)
```

_

_

Chapter 2

Separating columns

The **separate() function** allows you to separate one column into multiple columns. Unless you tell it otherwise, it will attempt to separate on any character that is not a letter or number. You can also specify a specific separator using the sep argument.

We've loaded the small dataset from the video called treatments into your workspace. This dataset obeys the principles of tidy data, but we'd like to split the treatment dates into two separate columns: year and month. This can be accomplished with the following:

```
separate(treatments, year_mo, c("year", "month"))
```
Experiment with this in the console before attempting the exercise.

We've loaded a dataset called bmi_cc into your workspace that is a slight variation of bmi_long, which you've already seen. The Country_ISO column of bmi_cc has the name of each country as well its two-letter ISO country code, separated by a forward slash.

- Apply the separate() function to bmi_cc
- Separate Country_ISO into two columns: Country and ISO
- Be sure to specify the correct separator with the sep argument
- Save the result to a new object called bmi_cc_clean
- View the head of the result

```
# Apply separate() to bmi_cc
bmi_cc_clean <- separate(bmi_cc, col = Country_ISO, into = c("Country", "ISO"), sep = "/")

# Print the head of the result
head(bmi_cc_clean)
```

_

Uniting columns

The opposite of separate() is unite(), which takes multiple columns and pastes them together. By default, the contents of the columns will be separated by underscores in the new column, but this behavior can be altered via the sep argument.

We've loaded the treatments data into your workspace again, but this time the year_mo column has been separated into year and month. The original column can be recreated by putting year and month back together:

unite(treatments, year_mo, year, month)
Experiment with this in the console before attempting the exercise.

In the last exercise, you separated the Country_ISO column of the bmi_cc dataset into two columns (Country and ISO) and saved the result to bmi_cc_clean. Now you're going to put the columns back together!

- Apply the unite() function to bmi_cc_clean
- Reunite the Country and ISO columns into a single column called Country_ISO
- Separate each country name and code with a dash (-)
- Save the result as bmi_cc
- View the head of the result

```
# Apply unite() to bmi_cc_clean
bmi_cc <- unite(bmi_cc_clean, Country_ISO, Country, ISO, sep = "-")

# View the head of the result
head(bmi_cc)
```

_

Column headers are values, not variable names

You saw earlier in the chapter how we sometimes come across datasets where column names are actually values of a variable (e.g. months of the year). This is often the case when working with repeated measures data, where measurements are taken on subjects of interest on multiple occasions over time. The gather() function is helpful in these situations.

View the head of census.
Gather the month columns, creating two new columns (month and amount), saving the result to census2.
Run the code given to arrange() the rows of census2 by the YEAR column.
View the first 20 rows of the result.

tidyr and dplyr are already loaded for you

 # View the head of census
head(census)

 # Gather the month columns
census2 <- gather(census, month, amount, -YEAR)

 # Arrange rows by YEAR using dplyr's arrange
census2 <- arrange(census2, YEAR)

 # View first 20 rows of census2
head(census2, 20)

_

Variables are stored in both rows and columns

Sometimes you'll run into situations where **variables are stored in both rows and columns**. To illustrate this, we've loaded the pets dataset from the video, which tells us in a convoluted way how many birds, cats, and dogs Jason, Lisa, and Terrence have. Print the pets dataset to see for yourself.

Although it may not be immediately obvious, if we treat the values in the type column as variables and create a separate column for each of them, we can set things straight. To do this, we use the spread() function. Run the following code to see for yourself:

spread(pets, type, num)
The result shows the exact same information in a much clearer way! Notice that the spread() function took in three arguments. The first argument takes the name of your messy dataset (pets), the second argument takes the name of the column to spread into new columns (type), and the third argument takes the column that contains the value with which to fill in the newly spread out columns (num).

Now let's try this on a new messy dataset census_long. What information does this tell us?

- View the first 50 rows of census_long
- Decide which column of census_long would be best to spread, and which column of census_long would be best to display in the newly spread out columns. Use the spread() function accordingly and save the result to census_long2
- View the first 20 rows of census_long2

tidyr is already loaded for you

 # View first 50 rows of census_long
head(census_long, 50)

 # Spread the type column
census_long2 <- spread(census_long, type, amount)

 # View first 20 rows of census_long2
head(census_long2, 20)

_

Multiple values are stored in one column
It's also fairly common that you will find two variables stored in a single column of data. These variables may be joined by a separator like a dash, underscore, space, or forward slash.

The separate() function comes in handy in these situations. To practice using it, we have created a slight modification of last exercise's result. Keep in mind that the into argument, which specifies the names of the 2 new columns being formed, must be given as a character vector (e.g. c("column1", "column2")).

- View the head of census_long3
- Use tidyr's separate() to split the yr_month column into two separate variables: year and month, saving the result to census_long4
- View the first 6 rows of the result

tidyr is already loaded for you

 # View the head of census_long3
head(census_long3)

 # Separate the yr_month column into two
census_long4 <- separate(census_long3,yr_month,c("year", "month"))

 # View the first 6 rows of the result
head(census_long4)

_

_

Chapter 3

Types of variables in R

As in other programming languages, R is capable of storing data in many different formats, most of which you've probably seen by now.

Loosely speaking, the class() function tells you what type of object you're working with. (There are subtle differences between the class, type, and mode of an object, but these distinctions are beyond the scope of this course.)

Change the argument of each call to the class() function so it evaluates to the following (in order):

|| "character"
|| "numeric"
|| "integer"
|| "factor"
V "logical"

Add or remove quotes, add an L to numerics to make them integers and use the factor() function when appropriate to accomplish this!

 # Make this evaluate to "character"
class(“TRUE”)

 # Make this evaluate to "numeric"
class(8484.00)

 # Make this evaluate to "integer"
class(99L)

 # Make this evaluate to "factor"
class(factor("factor"))

 # Make this evaluate to "logical"
class(FALSE)

_

Common type conversions

It is often necessary to change, or coerce, the way that variables in a dataset are stored. This could be because of the way they were read into R (with read.csv(), for example) or perhaps the function you are using to analyze the data requires variables to be coded a certain way.

Only certain coercions are allowed, but the rules for what works are generally pretty intuitive. For example, trying to convert a character string to a number gives an error: as.numeric("some text").

There are a few less intuitive results. For example, under the hood, the logical values TRUE and FALSE are coded as 1 and 0, respectively. Therefore, as.logical(1) returns TRUE and as.numeric(TRUE) returns 1.

We've loaded a dataset called students into your workspace. These data provide information on 395 students including their grades in three classes (in the Grades column, separated by /).

- Use str() to preview students and see the class of each variable
- Coerce the following columns:
- Grades to character
- Medu to factor (categorical variable representing mother's education level)
- Fedu to factor (categorical variable representing father's education level)
- Use str() again to see the changes to students

 # Preview students with str()
str(students)

 # Coerce Grades to character
students$Grades <- as.character(students$Grades)

 # Coerce Medu to factor
students$Medu <- as.factor(students$Medu)

 # Coerce Fedu to factor
students$Fedu <- as.factor(students$Fedu)

 # Look at students once more with str()
str(students)

_

Working with dates

Dates can be a challenge to work with in any programming language, but thanks to the lubridate package, working with dates in R isn't so bad. Since this course is about cleaning data, we only cover the most basic functions from lubridate to help us standardize the format of dates and times in our data.

As you saw in the video, these functions combine the letters y, m, d, h, m, s, which stand for year, month, day, hour, minute, and second, respectively. The order of the letters in the function should match the order of the date/time you are attempting to read in, although not all combinations are valid. Notice that the functions are "smart" in that they are capable of parsing multiple formats.

We have loaded a dataset called students2 into your workspace. students2 is similar to students, except now instead of an age for each student, we have a (hypothetical) date of birth in the dob column. There's another new column called nurse_visit, which gives a timestamp for each student's most recent visit to the school nurse.

- Preview students2 with str(). Notice that dob and nurse_visit are both stored as character
- Load the lubridate package
- Print "17 Sep 2015" as a date
- Print "July 15, 2012 12:56" as a date and time (note there are hours and minutes, but no seconds!)
- Coerce dob to a date (with no time)
- Coerce nurse_visit to a date and time
- Use str() to see the changes to students2

 # Preview students2 with str()
str(students2)

 # Load the lubridate package
library(lubridate)

 # Parse as date
dmy("17 Sep 2015")

 # Parse as date and time (with no seconds!)
mdy_hm("July 15, 2012 12:56")

 # Coerce dob to a date (with no time)
students2$dob <- ymd(students2$dob)

 # Coerce nurse_visit to a date and time
students2$nurse_visit <- ymd_hms(students2$nurse_visit)

 # Look at students2 once more with str()
str(students2)

Reading materials: Parsing Dates and Times

_

Trimming and padding strings

One common issue that comes up when cleaning data is the need to remove leading and/or trailing white space. The str_trim() function from stringr makes it easy to do this while leaving intact the part of the string that you actually want.

```
> str_trim("  this is a test     ")
[1] "this is a test"
```

A similar issue is when you need to pad strings to make them a certain number of characters wide. One example is if you had a bunch of employee ID numbers, some of which begin with one or more zeros. When reading these data in, you find that the leading zeros have been dropped somewhere along the way (probably because the variable was thought to be numeric and in that case, leading zeros would be unnecessary.)

```
> str_pad("24493", width = 7, side = "left", pad = "0")
[1] "0024493"
```

Load the stringr package
Trim all leading and trailing whitespace from the first set of strings
Pad the second set of strings with leading zeros such that all are 9 characters in length

 # Load the stringr package

```
 library("stringr")
```

 # Trim all leading and trailing whitespace

```
c("   Filip ", "Nick  ", " Jonathan")
str_trim(c("   Filip ", "Nick  ", " Jonathan"))
```

 # Pad these strings with leading zeros

```
c("23485W", "8823453Q", "994Z")
str_pad(c("23485W", "8823453Q", "994Z"), width=9, side="left", pad="0" )
```

_

Upper and lower case

In addition to trimming and padding strings, you may need to adjust their case from time to time. Making strings uppercase or lowercase is very straightforward in (base) R thanks to toupper() and tolower(). Each function takes exactly one argument: the character string (or vector/column of strings) to be converted to the desired case.

There's a vector of state abbreviations called states in your workspace, but there's a problem...it's all lowercase. It's more common for state abbreviations to be all uppercase.

Print states to the console
Make states all uppercase and save the result to states_upper
Make states_upper all lowercase again, but don't save the result

 # Print state abbreviations
states

 # Make states all uppercase and save result to states_upper
states_upper <- toupper(states)

 # Make states_upper all lowercase again
tolower(states_upper)

_

Finding and replacing strings

The stringr package provides two functions that are very useful for finding and/or replacing patterns in strings: str_detect() and str_replace().

Like all functions in stringr, the first argument of each is the string of interest. The second argument of each is the pattern of interest. In the case of str_detect(), this is the pattern we are searching for. In the case of str_replace(), this is the pattern we want to replace. Finally, str_replace() has a third argument, which is the string to replace with.

```
> str_detect(c("banana", "kiwi"), "a")
[1]  TRUE FALSE
> str_replace(c("banana", "kiwi"), "a", "o")
[1] "bonana" "kiwi"
```

The data.frame students2 is already available for you in the workspace. stringr is already loaded. students3 is a copy of it for you to work on so you can always start from scratch if you happen to make a mistake.

The students2 dataset from earlier in the chapter has been loaded for you again.

- Look at the head() of students3 to remind yourself of how it looks.
- Detect all dates of birth (dob) in 1997 using str_detect(). This should return a vector of TRUE and FALSE values.
- Replace all instances of "F" with "Female" in students3$sex
- Replace all instances of "M" with "Male" in students3$sex
- View the head() of students2 to see the result of these replacements

 # Copy of students2: students3
students3 <- students2

 # Look at the head of students3
#head(students3)

 # Detect all dates of birth (dob) in 1997
str_detect(students3$dob, "1997")

 # In the sex column, replace "F" with "Female" ...
students3$sex <- str_replace(students3$sex, "F", "Female")

 # ... and "M" with "Male"
students3$sex <- str_replace(students3$sex, "M", "Male")

 # View the head of students3
head(students3)

_

Dealing with missing values

Missing values can be a rather complex subject, but here we'll only look at the simple case where you are simply interested in normalizing and/or removing all missing values from your data. For more information on why this is not always the best strategy, search online for "missing not at random."
Looking at the social_df dataset again, we asked around a bit and figured out what's causing the missing values that you saw in the last exercise. Tom doesn't have a social media account on this particular platform, which explains why his number of friends and current status are missing (although coded in two different ways). Alice is on the platform, but is a passive user and never sets her status, hence the reason it's missing for her.

- Replace all empty strings (i.e. "") with NA in the status column of social_df.
- Print the updated version of social_df to confirm your changes.
- Use complete.cases() to return a vector containing TRUE and FALSE to see which rows have NO missing values.
- Use na.omit() to remove all rows with one or more missing values (without saving the result).

The stringr package is preloaded

 # Replace all empty strings in status with NA

social_df$status[social_df$status == ""] <- NA

 # Print social_df to the console

print(social_df)

 # Use complete.cases() to see which rows have no missing values

complete.cases(social_df)

 # Use na.omit() to remove all rows with any missing values

na.omit(social_df)

_

Identifying outliers and obvious errors

Which two of the following are most useful for identifying outliers?
a. summary()
b. str()
c. hist()
d. outlier()

Possible Answers

- a and b
- **a and c <This one>**
- b and c

- b and d
- a and d

_

Dealing with outliers and obvious errors

When dealing with strange values in your data, you often must decide whether they are just extreme or actually erroneous. Extreme values show up all over the place, but you, the data analyst, must figure out when they are plausible and when they are not.
We have loaded a dataset called students3, which is another slight variation of the original students dataset. Two variables appear to have suspicious values: age and absences. Let's explore these values further.

- Call summary() on the full students3 dataset to expose the concerning values of age and absences.
- View a histogram (using hist()) of the age variable.
- View a histogram of the absences variable.
- View another histogram of absences, but force values of zero to be bucketed to the right of zero on the x-axis with right = FALSE (see ?hist for more info

 # Look at a summary() of students3
summary(students3)

 # View a histogram of the age variable
hist(students3$age)

 # View a histogram of the absences variable
Hist (students3$absences)

 # View a histogram of absences, but force zeros to be bucketed to the right of zero
Hist (students3$absences, **<u>right = FALSE</u>**)

_

Another look at strange values

Another useful way of looking at strange values is with boxplots. Simply put, boxplots draw a box around the middle 50% of values for a given variable, with a bolded horizontal line drawn at the median. Values that fall far from the bulk of the data points (i.e. outliers) are denoted by open circles. (If you're curious about the exact formula for determining what is "far", check out ?hist.)
In this situation, we are concerned about three things:

1. Since this dataset is about students and the only student above the age of 22 is 38 years old, we must wonder whether this is an error in the data or just an older student (perhaps returning to school after working for several years)
2. There are four values of -1 for the absencesvariable, which is either a mistake or an intentional coding meant to say, for example, "this value is missing"
3. There are several extreme values of absencesin the positive direction, with a maximum value of 75 (which is over 18 times the median value of 4)

- View a boxplot() of the age variable from students3
- View a boxplot() of the absences variable from students3

 # View a boxplot of age

boxplot(students3$age)

 # View a boxplot of absences
boxplot(students3$absences)

_

_

Chapter 4

Get a feel for the data

Before diving into our data cleaning routine, we must first understand the basic structure of the data. This involves looking at things like the `class()` of the data object to make sure it's what we expect (generally a `data.frame`) in addition to checking its dimensions with `dim()` and the column names with `names()`.

For the `weather` dataset, which is loaded in your workspace:

- Check that it's a `data.frame` using the function `class()`
- Look at the dimensions
- View the column names

 # Verify that weather is a data.frame
class(weather)

 # Check the dimensions
dim(weather)

 # View the column names
names(weather)

_

Summarize the data

Next up is to look at some summaries of the data. This is where functions like `str()`, `glimpse()` from dplyr, and `summary()` come in handy.

- View the structure of `weather` using base R
- Load the dplyr package
- View the structure of `weather`, the dplyr way
- View a `summary()` of `weather`

 # View the structure of the data
str(weather)

 # Load dplyr package
library(dplyr)

 # Look at the structure using dplyr's glimpse()

glimpse(weather)

 # View a summary of the data
summary(weather)

Take a closer look

After understanding the structure of the data and looking at some brief summaries, it often helps to preview the actual data. The functions head() and tail() allow you to view the top and bottom rows of the data, respectively. Recall you'll be shown 6 rows by default, but you can alter this behavior with a second argument to the function.

For the weather data:

- View the first 6 rows
- View the first 15 rows
- View the last 6 rows
- View the last 10 rows

 # View first 6 rows
head(weather)

 # View first 15 rows
head(weather, 15)

 # View the last 6 rows
tail(weather)

 # View the last 10 rows
tail(weather, 10)

_

Column names are values

The weather dataset suffers from one of the five most common symptoms of messy data: column names are values. In particular, the column names X1-X31 represent days of the month, which should really be values of a new variable called day.
The tidyr package provides the gather()function for exactly this scenario. To remind you of how it works, we've loaded a small dataset called df in your workspace. Give the following a try in the console before attempting the instructions below.
gather(df, time, val, t1:t3)

Notice that gather() allows you to select multiple columns to be gathered by using the :operator.

- Load the tidyr package
- Call gather() on the weather data to gather columns X1-X31. The two columns created as a result should be called day and value. Save the result as weather2
- View the result with head()

 # Load the tidyr package
library(tidyr)

 # Gather the columns
weather2 <- gather(weather, day, value, X1:X31, na.rm = TRUE)

 # View the head
head(weather2)

_

Values are variable names

Our data suffer from a second common symptom of messy data: values are variable names. Specifically, values in the measure column should be variables (i.e. column names) in our dataset.
The spread() function from tidyr is designed to help with this. To remind you of how this function works, we've loaded another small dataset called df2 (which is the result of applying gather() to the original df from last exercise). Give the following a try before attempting the instructions below.
spread(df2, time, val)

Note how the values of the time column now become column names.

- Using the code provided, remove the first column of weather2, assigning to without_x.
- Spread the measure column of without_x and save the result to weather3
- View the result with head()

The tidyr package is already loaded

 # First remove column of row names
without_x <- weather2[, -1]

 # Spread the data
weather3 <- spread(without_x, measure, value)

 # View the head
head(weather3)

_

Clean up dates

Now that the weather dataset adheres to tidy data principles, the next step is to prepare it for analysis. We'll start by combining the year, month, and day columns and recoding the resulting character column as a date. We can use a combination of base R, stringr, and lubridate to accomplish this task.

- Load the stringr and lubridate packages
- Use stringr's str_replace() to remove the Xs from the day column of weather3
- Create a new column called date. Use the unite()function from tidyr to paste together the year, month, and day columns in order, using - as a separator (see ?unite if you need help)
- Coerce the date column using the appropriate function from lubridate
- Use the code provided (select()) to reorder columns, saving the result to weather5
- View the head of weather5

tidyr and dplyr are already loaded

 # Load the stringr and lubridate packages
library(stringr)

library(lubridate)

 # Remove X's from day column
weather3$day <- str_replace(weather3$day, "X", "")

 # Unite the year, month, and day columns
weather4 <- unite(weather3, date, year, month, day, sep = "-")

 # Convert date column to proper date format using lubridates's ymd()
weather4$date <- ymd(weather4$date)

 # Rearrange columns using dplyr's select()
weather5 <- select(weather4, date, Events, CloudCover:WindDirDegrees)

 # View the head of weather5
head(weather5)

A closer look at column types

It's important for analysis that variables are coded appropriately. This is not yet the case with our weather data. Recall that functions such as `as.numeric()` and `as.character()` can be used to *coerce* variables into different types.
It's important to keep in mind that coercions are not always successful, particularly if there's some data in a column that you don't expect. For example, the following will cause problems:
`as.numeric(c(4, 6.44, "some string", 222))`

If you run the code above in the console, you'll get a warning message saying that R introduced an `NA` in the process of coercing to numeric. This is because it doesn't know how to make a number out of a string (`"some string"`). Watch out for this in our weather data!

- Use `str()` to see how variables are stored in `weather5`
- View the first 20 rows of `weather5`. Keep an eye out for strange values!
- Try coercing the `PrecipitationIn` column of `weather5` to numeric without saving the result

 # View the structure of weather5
str(weather5)

 # Examine the first 20 rows of weather5. Are most of the characters numeric?
head(weather5, 20)

 # See what happens if we try to convert PrecipitationIn to numeric
as.numeric(weather5$PrecipitationIn)

_

Column type conversions

As you saw in the last exercise, `"T"` was used to denote a *trace*amount (i.e. too small to be accurately measured) of precipitation in the `PrecipitationIn` column. In order to coerce this column to numeric, you'll need to deal with this somehow. To keep things simple, we will just replace `"T"`with zero, as a string ("0").

- Use `str_replace()` from stringr to make the proper replacements in the `PrecipitationIn` column of `weather5`

- Run the call to mutate_at as-is to conveniently apply as.numeric() to all columns from CloudCover through WindDirDegrees (reading left to right in the data), saving the result to weather6
- View the structure of weather6 to confirm the coercions were successful

The dplyr and stringr packages are already loaded

 # Replace "T" with "0" (T = trace)
weather5$PrecipitationIn <- str_replaceweather5$PrecipitationIn, "T", "0")

 # Convert characters to numerics
weather6 <- mutate_at(weather5, vars(CloudCover:WindDirDegrees), funs(as.numeric))

 # Look at result
str(weather6)

_

Find missing values

Before dealing with missing values in the data, it's important to find them and figure out why they exist in the first place. If your dataset is too big to look at all at once, like it is here, remember you can use sum() and is.na() to quickly size up the situation by counting the number of NA values.
The summary() function may also come in handy for identifying which variables contain the missing values. Finally, the which() function is useful for locating the missing values within a particular column.

- Use sum() and is.na() to count the number of NAvalues in weather6
- Look at a summary() of weather6 to figure out how the missings are distributed among the different variables
- Use which() to identify the indices (i.e. row numbers) where Max.Gust.SpeedMPH is NA and save the result to ind (for *indices*)
- Use ind to look at the full rows of weather6 for which Max.Gust.SpeedMPH is missing

 # Count missing values
sum(is.na(weather6))

 # Find missing values
summary(weather6)

 # Find indices of NAs in Max.Gust.SpeedMPH
ind <- which(is.na(weather6$Max.Gust.SpeedMPH))

 # Look at the full rows for records missing Max.Gust.SpeedMPH
weather6[ind,]

_

An obvious error

Besides missing values, we want to know if there are values in the data that are too extreme or bizarre to be plausible. A great way to start the search for these values is with summary().
Once implausible values are identified, they must be dealt with in an intelligent and informed way. Sometimes the best way forward is obvious and other times it may require some research and/or discussions with the original collectors of the data.

- View a summary() of weather6
- Use which() to find the index of the erroneous element of weather6$Max.Humidity, saving the result to ind
- Use ind to look at the full row of weather6 for that day
- You discover an extra zero was accidentally added to this value. Correct it in the data

 # Review distributions for all variables
summary(weather6)

 # Find row with Max.Humidity of 1000
ind <- which(weather6$Max.Humidity == 1000, arr.ind=TRUE)

 # Look at the data for that day
weather6[ind,]

 # Change 1000 to 100
weather6$Max.Humidity[ind] <- 100

_

Another obvious error

You've discovered and repaired one obvious error in the data, but it appears that there's another. Sometimes you get lucky and can infer the correct or intended value from the other data. For example, if you know the minimum and maximum values of a particular metric on a given day...

- Use summary() to look at the value of *only* the Mean.VisibilityMilesvariable of weather6
- Determine the element of the value that is clearly erroneous in this column, saving the result to ind
- Use ind to look at the full row of weather6 for this day
- Inspect the values of other variables for this day to determine the correct value of Mean.VisibilityMiles, then make the appropriate fix

 # Look at summary of Mean.VisibilityMiles
summary(weather6$Mean.VisibilityMiles)

 # Get index of row with -1 value
ind <- which(weather6$Mean.VisibilityMiles == -1)

 # Look at full row
weather6[ind,]

 # Set Mean.VisibilityMiles to the appropriate value
weather6$Mean.VisibilityMiles[ind] <- 10

_

Check other extreme values

In addition to dealing with obvious errors in the data, we want to see if there are other extreme values. In addition to the trusty summary() function, hist() is useful for quickly getting a feel for how different variables are distributed.

- Check a summary() of weather6 one more time for extreme or unexpected values
- View a histogram for MeanDew.PointF

- Do the same for Min.TemperatureF
- And once more for Mean.TemperatureF to compare distributions

 # Review summary of full data once more
summary(weather6)

 # Look at histogram for MeanDew.PointF
hist(weather6$MeanDew.PointF)

 # Look at histogram for Min.TemperatureF
hist(weather6$Min.TemperatureF)

 # Compare to histogram for Mean.TemperatureF
hist(weather6$Mean.TemperatureF)

Finishing touches

Before officially calling our weather data clean, we want to put a couple of finishing touches on the data. These are a bit more subjective and may not be necessary for analysis, but they will make the data easier for others to interpret, which is generally a good thing.
There are a number of stylistic conventions in the R language. Depending on who you ask, these conventions may vary. Because the period (.) has special meaning in certain situations, we generally recommend using underscores (_) to separate words in variable names. We also prefer all lowercase letters so that no one has to remember which letters are uppercase or lowercase.
Finally, the events column (renamed to be all lowercase in the first instruction) contains an empty string ("") for any day on which there was no significant weather event such as rain, fog, a thunderstorm, etc. However, if it's the first time you're seeing these data, it may not be obvious that this is the case, so it's best for us to be explicit and replace the empty strings with something more meaningful.

- We've created a vector of column names in your workspace called new_colnames, all of which obey the conventions described above. Clean up the column names of weather6 by assigning new_colnames to names(weather6)
- Replace all empty strings in the events column of weather6with "None"
- One last time, print out the first 6 rows of the weather6 data frame to see the changes

 # Clean up column names
names(weather6) <- new_colnames
Replace empty cells in events column
weather6$events[weather6$events == ""] <- "None"

 # Print the first 6 rows of weather6
head(weather6)