**Here's what clean data look like**

100xp

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

**Instructions**

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

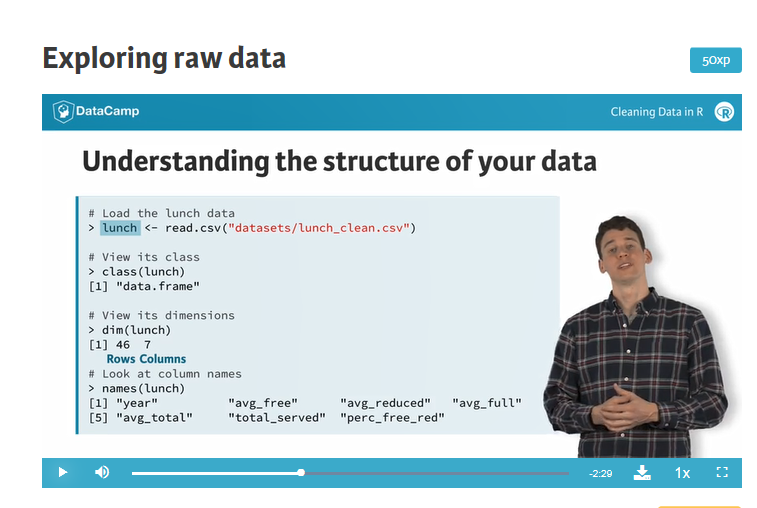
head(weather\_clean)

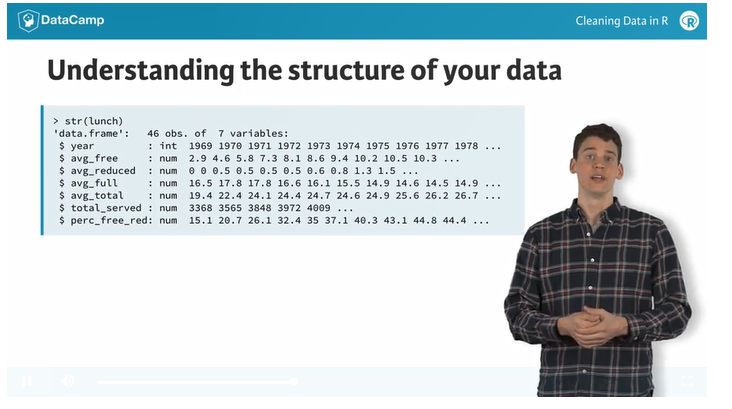
# View the last 6 rows of data

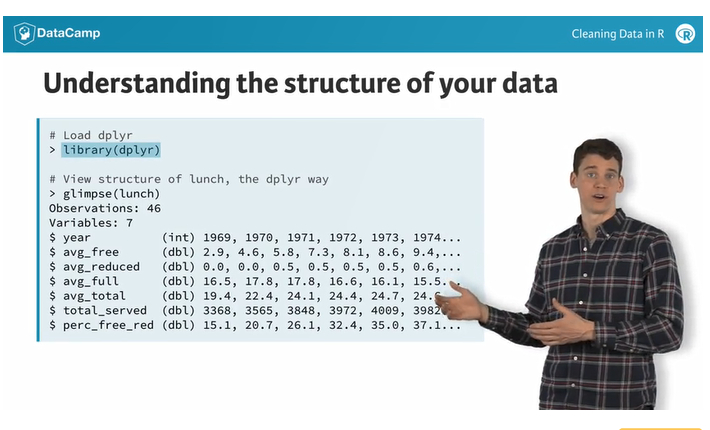
tail(weather\_clean)

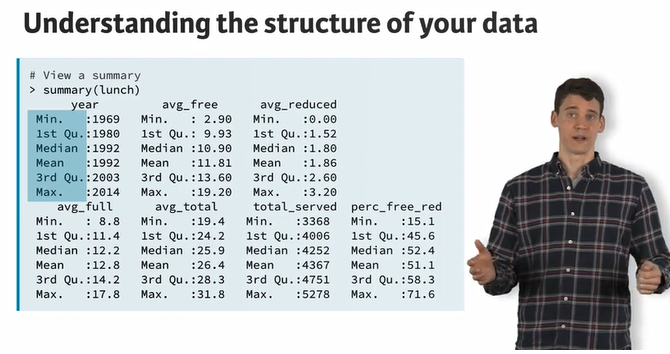
# View a condensed summary of the data

str(weather\_clean)









# Getting a feel for your data

100xp

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.

## Instructions

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](https://www1.imperial.ac.uk/publichealth/departments/ebs/projects/eresh/majidezzati/healthmetrics/metabolicriskfactors/).

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

**Take Hint (-30xp)**

* script.R



1

2

3

4

5

6

7

8

# Check the class of bmi

class(bmi)

# Check the dimensions of bmi

dim(bmi)

# View the column names of bmi

names(bmi)

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Reset To Sample Code

Ctrl+Shift+Enter

Submit Answer

* R Console
* Slides



1

2

3

4

5

> class(bmi)

[1] "data.frame"

> dim(bmi)

[1] 199 30

> names(bmi)

[1] "Country" "Y1980" "Y1981" "Y1982" "Y1983" "Y1984" "Y1985"

[8] "Y1986" "Y1987" "Y1988" "Y1989" "Y1990" "Y1991" "Y1992"

[15] "Y1993" "Y1994" "Y1995" "Y1996" "Y1997" "Y1998" "Y1999"

[22] "Y2000" "Y2001" "Y2002" "Y2003" "Y2004" "Y2005" "Y2006"

[29] "Y2007" "Y2008"

**Viewing the structure of your data**

100xp

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](https://www.datacamp.com/courses/dplyr-data-manipulation-r-tutorial) 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.

**Instructions**

* 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**

100xp

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.

**Instructions**

* 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, n=6)

# View the first 15 rows

head(bmi, n=15)

# View the last 6 rows

tail(bmi, n = 6)

# View the last 10 rows

tail(bmi, n=10)

**Visualizing your data**

100xp

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.

**Instructions**

For the bmi dataset:

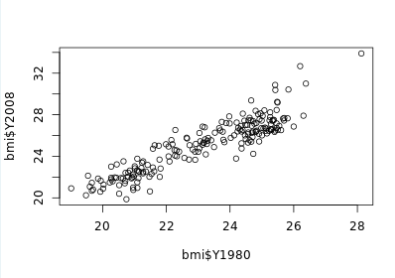
* Use hist() to look at the distribution of average BMI across all countries in 2008
* Use plot() to see how each country's average BMI in 1980 (x-axis) compared with its BMI in 2008 (y-axis)

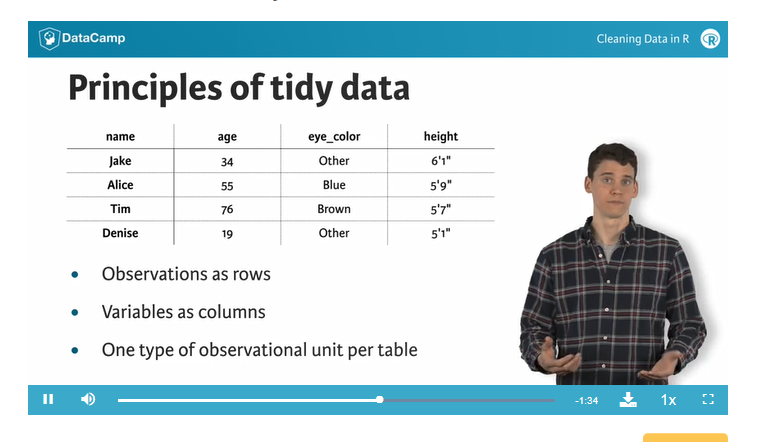
# Histogram of BMIs from 2008

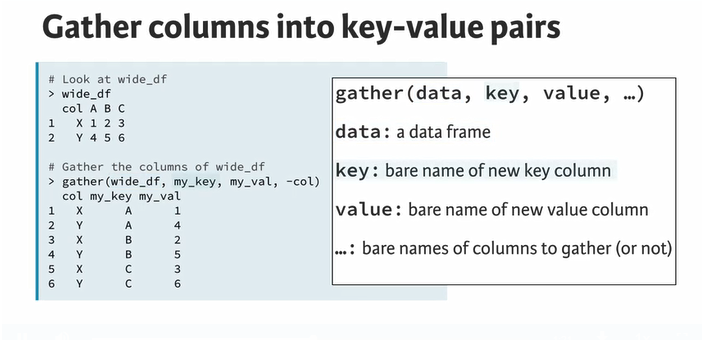
hist(bmi$Y2008)

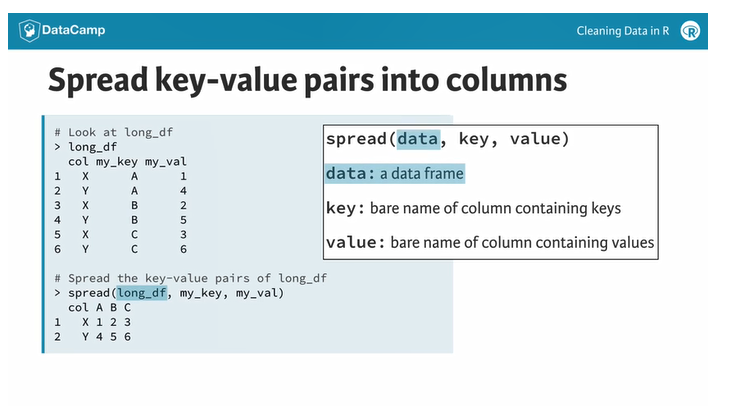
# Scatter plot comparing BMIs from 1980 to those from 2008

plot(bmi$Y1980, bmi$Y2008)









**Gathering columns into key-value pairs**

100xp

The most important function in tidyr is gather(). It should be used when you have columns that are not variables and you want to collapse them into key-value pairs.

The easiest way to visualize the effect of gather() is that it makes wide datasets long. As you saw in the video, running the following command on wide\_df will make it long:

gather(wide\_df, my\_key, my\_val, -col)

Experiment with this in the console before attempting the exercise.

**Instructions**

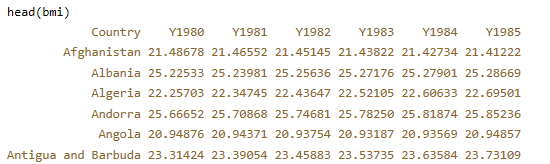
* Apply the gather() function to bmi, saving the result to bmi\_long. This will create two new columns:
  + year, containing as values what are currently column headers
  + bmi\_val, the actual BMI values
* View the first 20 rows of bmi\_long

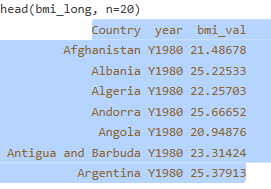
# Apply gather() to bmi and save the result as bmi\_long

bmi\_long <- gather(bmi, year, bmi\_val, -Country)

# View the first 20 rows of the result

head(bmi\_long, n=20)





**Spreading key-value pairs into columns**

100xp

The opposite of gather() is spread(), which takes key-values pairs and spreads them across multiple columns. This is useful when values in a column should actually be column names (i.e. variables). It can also make data more compact and easier to read.

The easiest way to visualize the effect of spread() is that it makes long datasets wide. As you saw in the video, running the following command will make long\_df wide:

spread(long\_df, my\_key, my\_val)

Experiment with this in the console before attempting the exercise.

**Instructions**

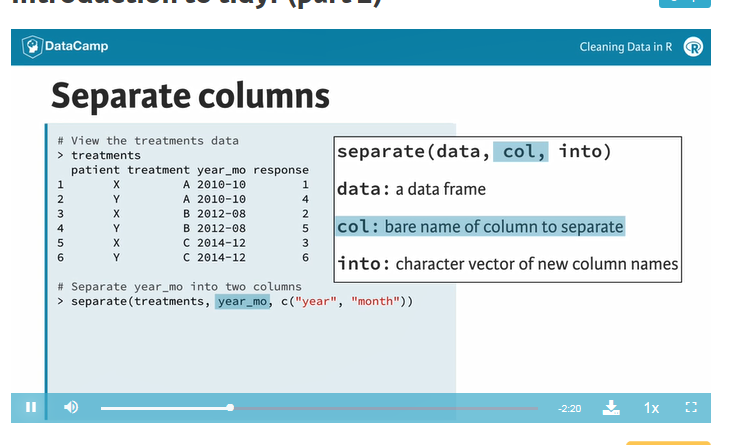
* Use spread() to reverse the operation that you performed in the last exercise with gather(). In other words, make bmi\_long wide again, saving the result to bmi\_wide
* View the head of bmi\_wide

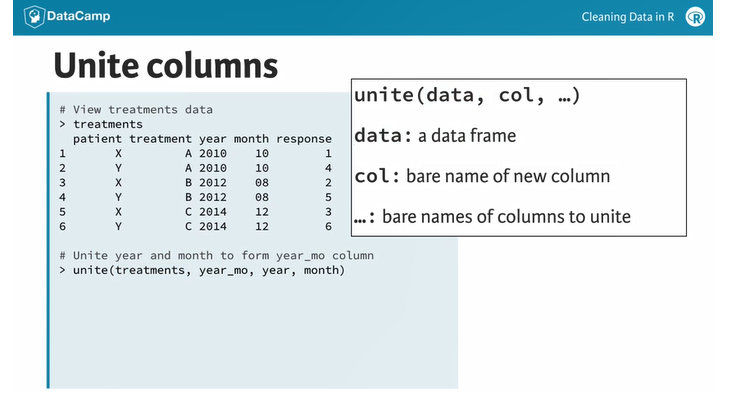
# Apply spread() to bmi\_long

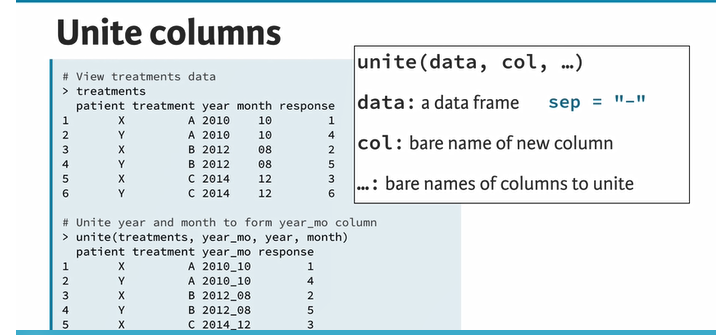
bmi\_wide <- spread(bmi\_long, year, bmi\_val)

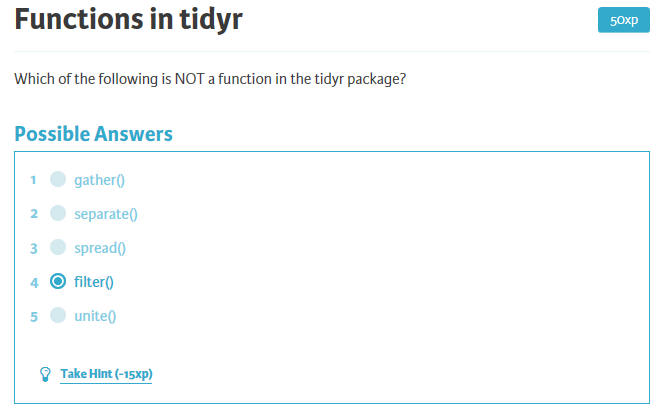
# View the head of bmi\_wide

head(bmi\_wide)









**Separating columns**

100xp

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.

**Instructions**

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

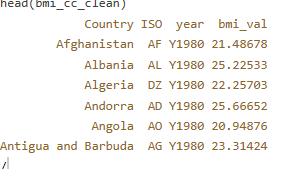
# Apply separate() to bmi\_cc

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**

100xp

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.

**Instructions**

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!

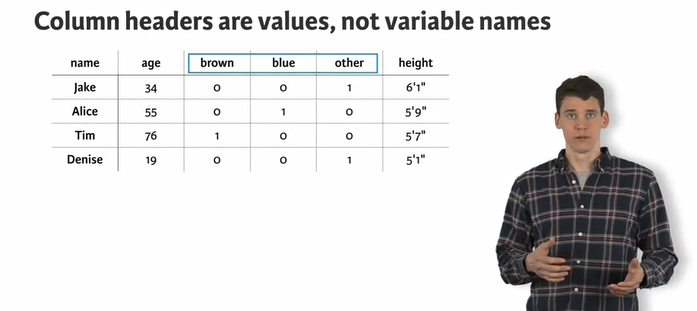
* Apply the unite() function to bmi\_cc\_clean
  + Reunite the Country and ISO columns into a single column called Country\_ISO
  + Separate each country name and code with a dash (-)
  + Save the result as bmi\_cc
* View the head of the result

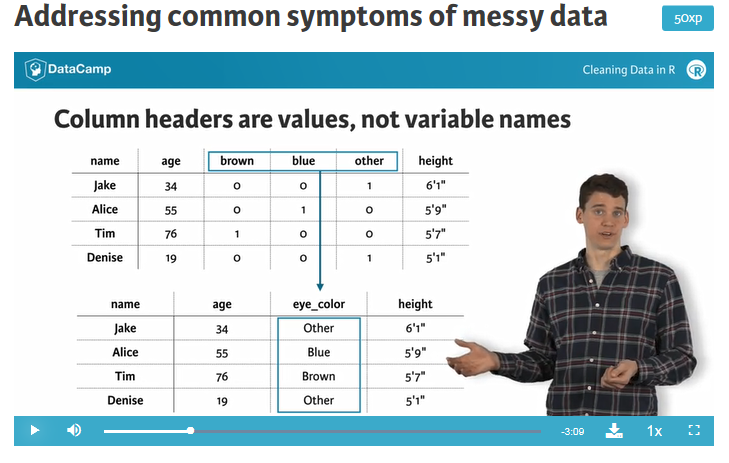
# Apply unite() to bmi\_cc\_clean

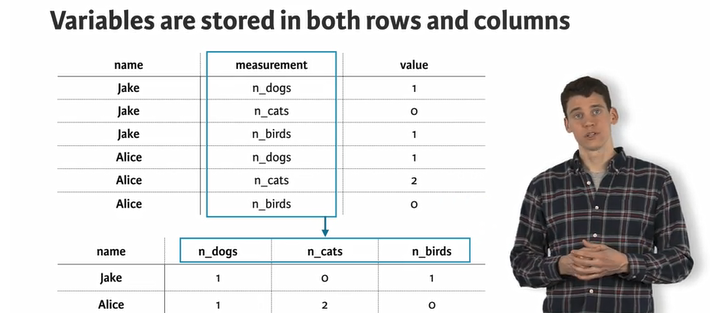
bmi\_cc <- unite(bmi\_cc\_clean, Country\_ISO, Country, ISO, sep = "-")

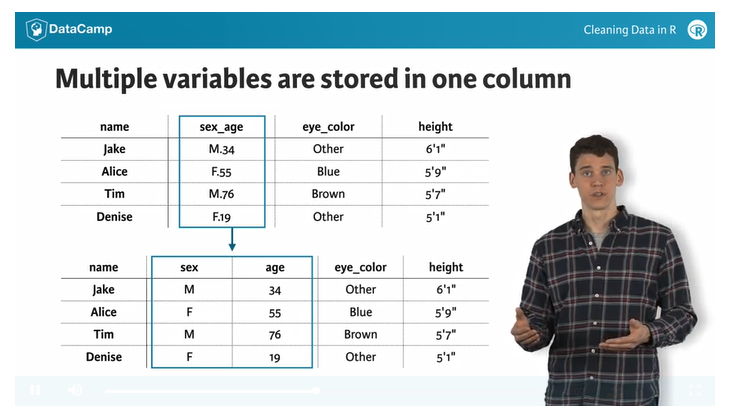
# View the head of the result

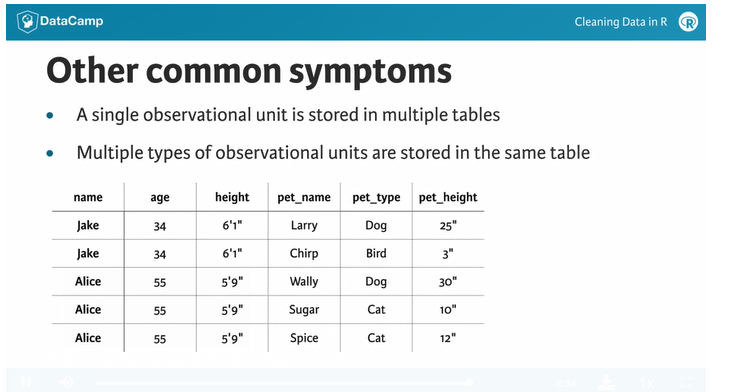
head(bmi\_cc)











**Column headers are values, not variable names**

100xp

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.

**Instructions**

* 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

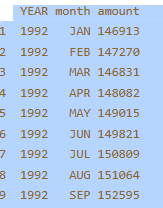
census2 <- gather(census, month, amount , -YEAR)

# Arrange rows by YEAR using dplyr's arrange

census2 <- arrange(census2, YEAR)

# View first 20 rows of census2

head(census2, n=20)



# Variables are stored in both rows and columns

100xp

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?

## Instructions

* 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

100xp

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")).

## Instructions

* 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

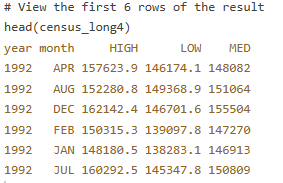
head(census\_long3)

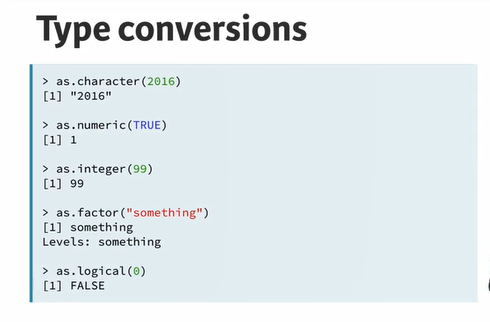
# Separate the yr\_month column into two

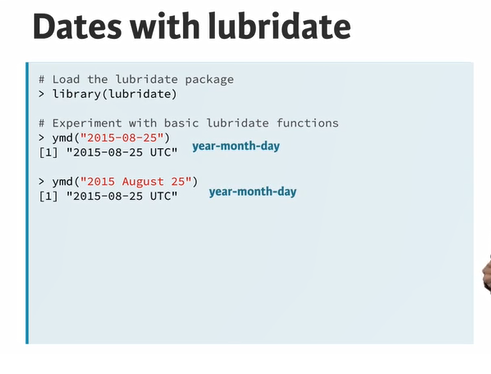
census\_long4 <- separate(census\_long3, yr\_month, c("year", "month"))

# View the first 6 rows of the result

head(census\_long4)







Change the object within each call of the class() function to make it evaluate to the following (in order):

* character
* numeric
* integer
* factor
* 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

100xp

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.

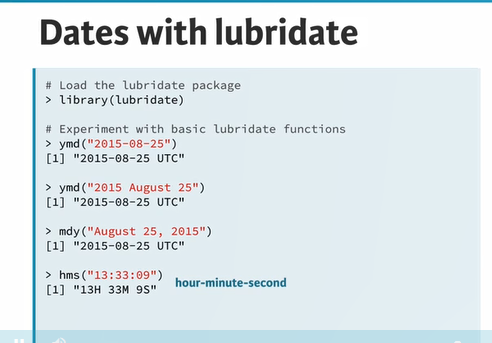
Only certain coercions are allowed, but the rules for what works are generally pretty intuitive. For example, trying to convert a character string to a number **gives an error**: as.numeric("some text").

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.

## Instructions

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



# Common type conversions

100xp

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.

## Instructions

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

100xp

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.

## Instructions

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)

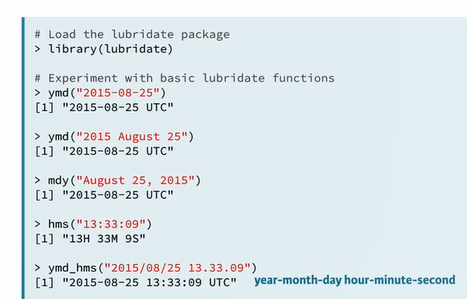
students2$dob <- ymd(students2$dob)

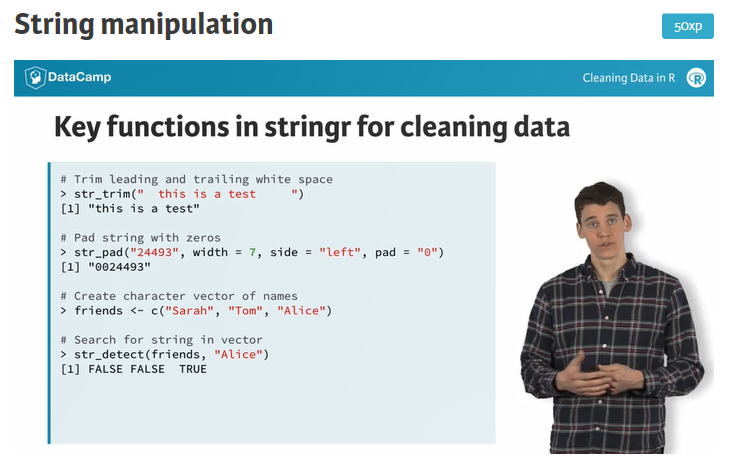
# Coerce nurse\_visit to a date and time

students2$nurse\_visit <- ymd\_hms(students2$nurse\_visit)

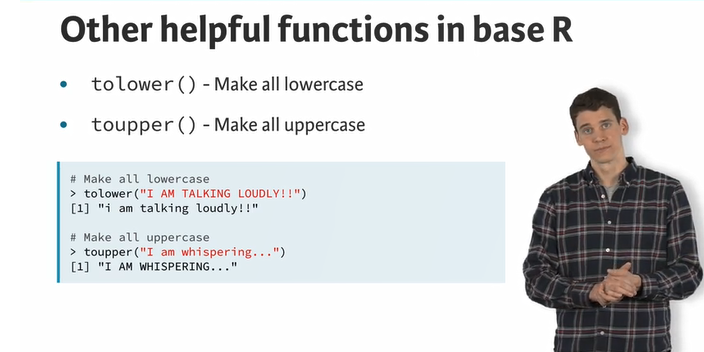
# Look at students2 once more with str()

str(students2)









# Trimming and padding strings

100xp

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"

## Instructions

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