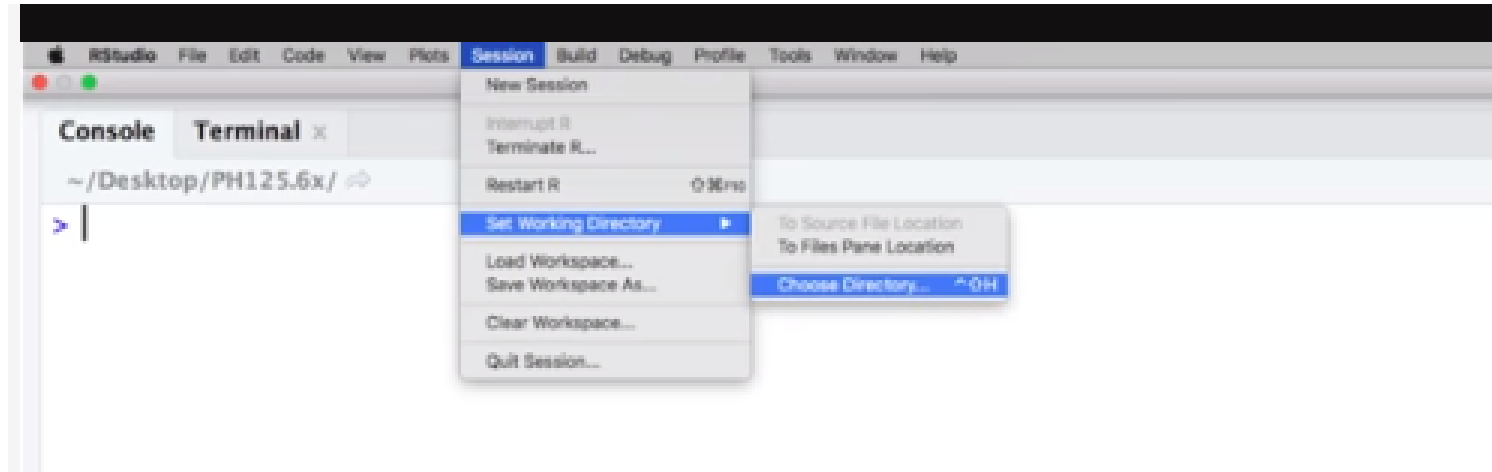


## HARVARDx 06 - Data Wrangling

### Preparing to Read in a File - Knowing the Working Directory

The first step is to find the file containing your data and know its location on your file system. For this reason and others, when you are working in R, it is important to know your working directory. This is the directory in which R will save or look for files by default. You can see your working directory by typing the following command--`getwd()`.

You can change a working directory using the function `setwd`. If you are working in R Studio, you can change it by clicking on Session:



Unless a full path is provided, they search for files in the working directory. For this reason, our recommended approach for beginners is that you create a directory for each analysis. To keep raw data files organized, we recommend creating a data directory inside your project directory, especially when the project involves more than one data file.

Because you may not have a data file handy yet, we provide an example data file in the DSLABS package. Once you download and install that the DSLABS package, files will be in the external data, `extdata`, directory that you can get by typing this command.

```
> system.file("extdata", package="dslabs")
[1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/dslabs/extdata"
```

To see the files, use `list`:

```
> path <- system.file("extdata", package="dslabs")
> list.files(path)
[1] "fertility-two-countries-example.csv"
[2] "life-expectancy-and-fertility-two-countries-example.csv"
[3] "murders.csv"
```

we are ready to import them into R. To make the code simpler, you can move this file to your working directory. You can do this through the file system directly. But you can also do it within R itself, using the file.copy() function. To do this, it will help to define a variable with the full path using the function file.path().

```
> filename <- "murders.csv"
> fullpath <- file.path(path, filename)
> fullpath
[1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/dslabs/extdata/murders.csv"
> file.copy(fullpath, getwd())
[1] TRUE

> file.exists(filename)
[1] TRUE
```

Reading in Data to R

Function	Format	Typical Suffix
read_table	white space separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must define delimiter	txt

Function	Format	Typical Suffix
read_excel	auto detect the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new format	xlsx

readr is the tidyverse library that includes functions for reading data stored in text file spreadsheets into R. The following functions are available to read in spreadsheet files-- read\_table, read\_csv, read\_csv2, read\_tsv, and read\_delim. What makes these different is the type of delimiter that it works with. The read\_excel package provides functions to read in data in the Microsoft Excel format. Note: you need the library(readxl) to read xls files.

The function excel\_sheets gives us the names of the sheets in an Excel file. These names can then be passed on to the sheet argument in the three functions that we just described to read in Excel files.

Note that the suffix usually tells us what type of file it is. But there's no guarantee that these always match. To be sure, we can open the file to take a look or use functions such as read\_lines that show us the first few lines of a file within R. You can do it like this.

```
> read_lines("murders.csv", n_max = 3)
[1] "state,abb,region,population,total" "Alabama,AL,South,4779736,135"
[3] "Alaska,AK,West,710231,19"
```

This also shows us if there's a header or not.

We can read in the file like this:

```
> dat <- read_csv(filename)
Parsed with column specification:
cols(
  state = col_character(),
  abb = col_character(),
  region = col_character(),
  population = col_integer(),
  total = col_integer()
)
```

```
> dat <- read_csv(fullpath)
Parsed with column specification:
cols(
  state = col_character(),
  abb = col_character(),
  region = col_character(),
  population = col_integer(),
  total = col_integer()
)
```

The object we created is a tibble with the content of the file.

```
> head(dat)
# A tibble: 6 x 5
  state      abb region population total
  <chr>    <chr> <chr>      <int> <int>
1 Alabama  AL    South    4779736 135
2 Alaska   AK    West     710231 19
3 Arizona  AZ    West    6392017 232
4 Arkansas AR    South    2915918 93
5 California CA    West    37253956 1257
6 Colorado CO    West     5029196 65
```

So now we're ready to read in the data into R. From the suffix and the peek that we look, we know that we should use the `read_csv()` function. We do it like this.

```
> dat <- read_csv(filename)
Parsed with column specification:
cols(
  state = col_character(),
  abb = col_character(),
  region = col_character(),
  population = col_integer(),
  total = col_integer()
)
```

Note that when we run these functions, we receive a message letting us know what data types were used for each column. Also note that that, the object that we just created by reading in the file, is a tibble with the content of the file. You can see the first six lines using the head function like this.

```
> head(dat)
# A tibble: 6 x 5
  state      abb region population total
  <chr>    <chr> <chr>      <int> <int>
1 Alabama AL    South    4779736 135
2 Alaska  AK    West     710231 19
3 Arizona AZ    West    6392017 232
4 Arkansas AR    South    2915918 93
5 California CA   West    37253956 1257
6 Colorado CO    West     5029196 65
```

**Reading dbase files:** We have `read.table`, `read.csv`, and `read.delim`. There are a couple of important differences you should know about. To show this, we read the data using an R:BASE function. We call the object `dat2`, like this.

```
> dat2 <- read.csv(filename)
```

One difference is that now we have a `data.frame`, not a table. You can see it using the `class` function.

```
> class(dat2$abb)
[1] "factor"
> class(dat2$region)
[1] "factor"
```

The other difference is that the characters are converted to factors. Look at the class of the abbreviation column. Note, it's a factor. In the original file, there were characters. This can be avoided by setting the argument `stringsAsFactors` to `FALSE`.

```
> dat3 <- read.csv(filename, stringsAsFactors = FALSE)
> class(dat3$abb)
```

### Reading Data from the Internet

Another common place for data to result is on the internet. When these are data files, we can download them and then import them. Or we can read them indirectly from the web.

For example, we know that because our DS lab package is on GitHub, the file we downloaded for the package has a URL.

```
> url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"
```

The `read_csv` file can read these files directly. We use the URL instead of the file name when calling the function, like this.

```
> dat <- read_csv(url)
Parsed with column specification:
cols(
  state = col_character(),
  abb = col_character(),
  region = col_character(),
  population = col_integer(),
  total = col_integer()
)
```

Now, if you want to have a local copy of the file, you can use the `download.file` function, like this.

```
> download.file(url, "murders.csv")
trying URL 'https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv'
Content type 'text/plain; charset=utf-8' length 1684 bytes
=====
downloaded 1684 bytes
```

Two functions that are sometimes useful when downloading data from the internet are `tempdir()` and `tempfile()`. The first actually creates a directory with a name that is very unlikely not to be unique. Similarly, `tempfile` creates a character string, not a file, that is likely to be a unique file name. [Look at the file name that we get when we write `tempfile`.](#)

```
> tempfile()
[1] "/var/folders/c8/_3bwm84s56d31pnyry1cp5xc0000gp/T//RtmpU3ycGS/file2a529004c88"
```

So as an example, we'll use these commands to download a file, give it a temporary name, read it in, and then erase the files that we downloaded. We can do that using this code.



```
> tmp_filename <- tempfile()
  download.file(url, tmp_filename)
  dat <- read_csv(tmp_filename)
  file.remove(tmp_filename)
```

```
=====
downloaded 1684 bytes
```

```
> dat <- read_csv(tmp_filename)
Parsed with column specification:
cols(
  state = col_character(),
  abb = col_character(),
  region = col_character(),
  population = col_integer(),
  total = col_integer()
)
```

```
> file.remove(tmp_filename)
```

```
[1] TRUE
```

```
> head(dat)
```


```
# A tibble: 6 x 5
```

	state	abb	region	population	total
	<chr>	<chr>	<chr>	<int>	<int>
1	Alabama	AL	South	4779736	135
2	Alaska	AK	West	710231	19

### Tidy Data

If we go back to the original data provided by Gapminder, we see that it does not start out tidy. We include an example file with the data shown in this graph, mimicking the way it was originally saved in a spreadsheet. You can get to the file like this.

```
> path <- system.file("extdata", package="dslabs")
> filename <- file.path(path, "fertility-two-countries-example.csv")
> wide_data <- read_csv(filename)
Parsed with column specification:
cols(
  .default = col_double(),
  country = col_character()
)
See spec(...) for full column specifications.
```



After running that code, the object `wide_data` includes the same information as the object `tidy_data`, except it is in a different format--a wide format. Here are the first 9 columns:

```
> select(wide_data, country, `1960`:`1967`)
# A tibble: 2 x 9
  country    `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`
  <chr>      <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Germany    2.41    2.44    2.47    2.49    2.49    2.48    2.44    2.37
2 South Korea 6.16    5.99    5.79    5.57    5.36    5.16    4.99    4.85
```

Let's go over two important differences between the wide and tidy formats. First, in the wide format, each row includes several observations. Second, one of the variables-- the year--is stored in the header. The ggplot code we introduced earlier, no longer works if we feed it the wide data. For one, there is no year available. So to use the tidyverse we need to wrangle this data into tidy format.

## Reshaping Data

RAFAEL IRIZARRY: We've learned that having data in tidy format is what makes a tidy verse flow. After the first step in the data analysis process, importing data, a common next step is to reshape the data into a form that facilitates the rest of the analysis. The `tidyr` package includes several functions that are useful for tidying data. This package is included in the tidy verse.

One of the most used functions in the package is `gather`, which converts y data into tidy data. We'll get to the first and second argument of `gather` soon, but let's describe the third argument. The third argument of the `gather` function specifies the columns that will be gathered. The default behavior for the `gather` function is to gather all the columns. So in most cases, we have to specify the columns. In the example we've been examining, we want to gather the columns 1960, 1961, up to 2015. Those are the column names.



Now let's explain what the first argument of the gather function does. ***The first argument sets the name of the column that will hold the variable that are currently kept in the y data column names. In our case, it makes sense to set the name of this column to year, but we can name it anything. The second argument sets the column name for the column that will hold the values in the column cells.*** In this case, we'll call it fertility since that's the data that is in those cells. Know that nowhere in this file does it tell us that this is fertility data. We know this from the file name. This is not the best way to store data but it's the way this data was given to us.

Now the gathering code looks like this.

```
> new_tidy_data <- wide_data %>%  
+   gather(year, fertility, `1960`:`2015`)
```

We're going to create a new tidy data sets object, call it new underscore tidy underscore data, and now all we do is apply the gather function to the y data. We can see that the data have been converted to tidy format would columns year and fertility. Look at the first six rows.

```
> head(new_tidy_data)  
# A tibble: 6 x 3  
  country    year fertility  
  <chr>    <chr>    <dbl>  
1 Germany  1960      2.41  
2 South Korea 1960      6.16  
3 Germany  1961      2.44  
4 South Korea 1961      5.99  
5 Germany  1962      2.47  
6 South Korea 1962      5.79
```

Note that the only column that was not gathered was the countries column. That's because we asked for all the other ones to be gathered. So a somewhat quicker way to write this code is to specify which columns not to gather, rather than all the columns that will be gathered. So the code will look simply like this.

```
> new_tidy_data <- wide_data %>%  
+   gather(year, fertility, -country)
```

The object looks a lot like the original tidy data we showed earlier. There's just one minor difference. Can you spot it? Look at the data type for the Year column. It's a integer in our original tidy data table. In our new tidy data, the one we just gathered, it's a character. The gather function assumes that column names are characters, so we need a bit more wrangling before we're ready to make a plot. We need to convert this column to numbers.

```
> class(tidy_data$year)
[1] "integer"
> class(new_tidy_data$year)
[1] "character"
```

We can use as.numeric if we want, but the gather function actually has an argument for that. It's the convert argument. So the code would be like this.

```
> new_tidy_data <- wide_data %>%
+   gather(year, fertility, -country, convert = TRUE)
> class(new_tidy_data$year)
[1] "integer"
```

And once we do this, the class of the Year column is an integer. Now that the data is tidy, we can use the same ggplot commands to generate the plot we saw earlier. Like this.

```
> new_tidy_data <- wide_data %>%
+   gather(year, fertility, -country, convert = TRUE)
> class(new_tidy_data$year)
[1] "integer"
```

```
> new_tidy_data %>%
+   ggplot(aes(year, fertility, color = country)) +
+   geom_point()
```

So we've shown how to gather wide data into tidy data. Now, as we will see in later example, it is sometimes useful for data wrangling purposes to convert tidy data into the wide format data. We often use this as an intermediate step in tidying up data. The spread function is basically the inverse of gather. The first argument tells spread which variables will be used as the column names. The second argument specifies which variables to use to fill out the cells. So the code would look like this.

```

> new_wide_data <- new_tidy_data %>% spread(year, fertility)
> select(new_wide_data, country, `1960`:`1967`)
# A tibble: 2 x 9
  country    `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`
  <chr>      <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Germany    2.41   2.44   2.47   2.49   2.49   2.48   2.44   2.37
2 South Korea 6.16   5.99   5.79   5.57   5.36   5.16   4.99   4.85

```

This converts tiny data back into the wide format--as you can see here. This diagram can help you remember, how these two functions work.

## Reshaping Data

```

#
# DS5 Exercise to show how to use gather, separate, and unite to tidy
# data when data is spread across multiple columns
#
library(tidyverse)

filename <- "leafterc.csv"
d <- read_csv(filename)

select(d, 1:5)

# When we look at this table, we can see that it is in y format.
# Also note that there are values for two variables with the column
# names encoding which column represents which variable.

# A tibble: 2 x 5
# country `1960_fertility` `1960_life_expe~` `1961_fertility` `1961_life_exp~`
# <chr>      <dbl>          <dbl>          <dbl>          <dbl>
# Germany    2.41          69.3          2.44          69.8
# South ~    6.16          53           5.99          53.8

# We can start the data wrangling with the gather function,
# but we should no longer use the column name Year for the new columns,

```

```

# since it also contains the variable type. We will call it key.
# That's the default of this function. So we write this piece of code
# to gather the data.

dat <- d %>% gather(key, value, -country)
head(dat)

# A tibble: 6 x 3
#   country      key      value
#   <chr>      <chr>    <dbl>
# Germany  1960_fertility  2.41
# South Korea 1960_fertility  6.16
# Germany  1960_life_expectancy 69.3
# South Korea 1960_life_expectancy 53
# Germany  1961_fertility  2.44
# South Korea 1961_fertility  5.99

# The first challenge to achieve this is to separate the key column
# into the year and the variable type. We can add a third column to catch this and let
# the separate function know which column to fill in with missing values--
# NAs, in this case-- when there is no third value.

dat %>% separate(key, c("year", "first_variable_name", "second_variable_name"), "_", fill = "right")

# However, if we read the separate file, we find that a better approach
# is to merge the last two variables when there's an extra separation
# using the argument extra, like this.

dat %>% separate(key, c("year", "variable_name"), sep = "_", extra = "merge")

# However, we're not done yet. We need to create a column for each
# variable. As we've learned, the spread function can do this. So now,
# to create tidy data, we're actually using the spread function.
# So we write this piece of code, and when we run it,

```

```

dat %>% separate(key, c("year", "variable_name"), sep = "_", extra = "merge") %>% spread(variable_name, value)

# we now get a fertility and a life expectancy column. Alternatively, we
# could use separate and unite like this:

dat %>% separate(key, c("year", "first_variable_name", "second_variable_name"), "_", fill = "right") %>% unite(variable_name,
first_variable_name, second_variable_name, sep = "_")

# A tibble: 10 x 4
#   country    year variable_name    value
#   <chr>      <chr> <chr>          <dbl>
# Germany    1960 fertility_NA     2.41
# South Korea 1960 fertility_NA     6.16
# Germany    1960 life_expectancy 69.3
# South Korea 1960 life_expectancy 53
# Germany    1961 fertility_NA     2.44
# South Korea 1961 fertility_NA     5.99
# Germany    1961 life_expectancy 69.8
# South Korea 1961 life_expectancy 53.8
# Germany    1962 fertility_NA     2.47
# South Korea 1962 fertility_NA     5.79

# ...and then spread the columns with this code:
dat %>%
  separate(key, c("year", "first_variable_name", "second_variable_name"), "_", fill = "right") %>%
  unite(variable_name, first_variable_name, second_variable_name, sep = "_") %>%
  spread(variable_name, value)

# A tibble: 6 x 4
#   country    year fertility_NA life_expectancy
#   <chr>      <chr>      <dbl>          <dbl>
# Germany    1960         2.41           69.3
# Germany    1961         2.44           69.8

```

```
# Germany      1962      2.47      NA
# South Korea 1960      6.16      53
# South Korea 1961      5.99     53.8
# South Korea 1962      5.79      NA
```

## Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout.

**gather()**(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor\_key = FALSE)

gather() moves column names into a **key** column, gathering the column values into a single **value** column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

```
gather(table4a, `1999`, `2000`,
        key = "year", value = "cases")
```

**spread()**(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL)

spread() moves the unique values of a **key** column into the column names, spreading the values of a **value** column across the new columns.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

key value

```
spread(table2, type, count)
```

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

## Combining Tables / Joins

```
# Examples and explanatory Text on joins
```



```

@
library(dbplyr)
library(tidyverse)

# Read in the tables using the csv function
tab1 <- read_csv("murders")
tab2 <- read_csv("electoral-votes")

# Now perform a normal left join
left_join(tab1, tab2)

# We can also use the pipe, like this:
tab1 %>% left_join(tab2)

# Now a right join
tab1 %>% right_join(tab2)

# Use inner join to get just the rows common to both tables
inner_join(tab1, tab2)
# A full join is a union of the tables like this
full_join(tab1, tab2)

# The semi join function lets us keep the part of the first table
# for which we have information in the second. It does not add the
# columns of the second.
semi_join(tab1, tab2)

# The function anti join is the opposite of semi join. It keeps the
# elements of the first table for which there is no information in
# the second.
anti_join(tab1, tab2)

```

x				y			
A	B	C		A	B	C	
a	t	1		a	t	3	
b	u	2		b	u	2	
c	v	3		d	w	1	

Use a **"Mutating Join"** to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

Use a **"Filtering Join"** to filter one table against the rows of another.

A	B	C
a	t	1
b	u	2

**semi\_join(x, y, by = NULL, ...)**  
Return rows of x that have a match in y.  
USEFUL TO SEE WHAT WILL BE JOINED.

A	B	C
c	v	3

**anti\_join(x, y, by = NULL, ...)**  
Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA

**left\_join(x, y, by = NULL, copy=FALSE, suffix=c(".x", ".y"),...)**  
Join matching values from y to x.

A	B	C	D
a	t	1	3
b	u	2	2
d	w	NA	1

**right\_join(x, y, by = NULL, copy = FALSE, suffix=c(".x", ".y"),...)**  
Join matching values from x to y.

A	B	C	D
a	t	1	3
b	u	2	2

**inner\_join(x, y, by = NULL, copy = FALSE, suffix=c(".x", ".y"),...)**  
Join data. Retain only rows with matches.

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA
d	w	NA	1

**full\_join(x, y, by = NULL, copy=FALSE, suffix=c(".x", ".y"),...)**  
Join data. Retain all values, all rows.



## Set Operators

```
# Set Operators
#
# Another set of commands useful for combining data are the set operators.
# When applied to vectors, these behave as their name suggests-- union,
# intersect, et cetera. And we're going to see examples soon. However, if
# the tidyverse, or, more specifically, dplyr is loaded, these functions
# can be used on data frames, as opposed to just on vectors.
```

```
library(tidyverse)
library(dplyr)
```

```

intersect(1:10, 6:15)

# But with dplyr loaded, we can also do this for tables. It'll take the
# intersection of rows for tables having the same column names.

tab1 <- tab[1:5,]
tab2 <- tab[3:7,]
intersect(tab1, tab2)

# So if we take the first five rows of tab and rows three through seven of
# tabs, and we take the intersection, it will give us rows three, four,
# and five,

# Similarly, union takes the union. If you apply it to vectors, you get
# the union like this. But with dplyr loaded, we can also do this for
# tables having the same column names.

tab1 <- tab[1:5,]
tab2 <- tab[3:7,]
union(tab1, tab2)

# #We can also take set differences using the function setdiff. Unlike
# intersect and union, this function is not symmetric. For example, note
# that you get two different answers if you switch the arguments.

setdiff(1:10, 6:15)
setdiff(6:16, 1:10)

# And again, with dplyr loaded, we can apply this to data frames.
# Look what happens when we take the setdiff of tab one and tab two.

tab1 <- tab[1:5,]
tab2 <- tab[3:7,]
setdiff(tab1, tab2)

```

```
# Finally, the function setequal tells us if two sets are the same
# regardless of order. So for example, if I do set equals of one through
# five and one through six, I get false, because they're not the same vectors.
# But if I take set equals of one through five and five through one,
# I get true, because if you ignore order, these are the same vectors.
# With dplyer loaded, we can use this on data frames, as well.
# When applied to data frames that are not equal, regardless of order,
```

```
setequal(1:5, 1:6)
setequal(1:5, 5:1)
setequal(tab1, tab2)
```

## **Webscraping**

```
# Video and sample code for webscraping
```

```
library(tidyverse)
library(rvest)
```

```
# Import the web page
url <- "https://en.wikipedia.org/wiki/Murder_in_the_United_States_by_state"
h <- read_html(url)
```

```
# The class of the object is XML document
class(h)
```

```
# Here we know that the information is stored in an HTML table.
# You can see this in a line of code of the HTML document we showed earlier.
# <table class="wikitable sortable">
```

```
# The rvest package includes functions to extract nodes from HTML documents.
# The function html_nodes, plural, extracts all nodes of that type.
# And html_node extracts just the first node of that type.
# This gives us just the html code for that table
```

```

tab <- h %>% html_nodes("table")
tab <- tab[[2]]
tab

# In the code we just showed, you can definitely see a pattern,
# and writing code to extract just the data is very doable.
# In fact rvest includes a function precisely for this--
# for converting HTML tables into data frames.
tab <- tab %>% html_table
class(tab)

# We are now much closer to having a usable data table.
# Let's change the names of the columns, which are a little bit long,
# and then take a look. / You can see that we already
# have a data frame very close to what we want.
tab <- tab %>% setNames(c("state", "population", "total", "murders", "gun_ownership", "total_rate", "murder_rate",
"gun_murder_rate"))
head(tab)

# However, we still have some data wrangling to do.
# For example, notice that some of the columns that are supposed to be numbers
# are actually characters.
# And what makes it even worse is that some of them have commas,
# so it makes it harder to convert to numbers.
# Before we continue with this, we're going to learn a little bit more
# about general approaches to extracting information from websites.

```

Why did we use the `html_nodes()` command instead of the `html_node` command?

- ☒ The `html_node` command only selects the first node of a specified type. In this example the first “table” node is a legend table and not the actual data we are interested in. ✓
- ☒ The `html_nodes` command allows us to specify what type of node we want to extract, while the `html_node` command does not.
- ☐ It does not matter; the two commands are interchangeable.
- ☐ We used `html_nodes` so that we could specify the second “table” element using the `tab[[2]]` command. ✓

## CSS Elements

# Video and sample code for css and determining CSS elements to use

```
library(tidyverse)
library(rvest)
```

```
# The default look of webpage made with the most basic HTML is quite unattractive.
# The aesthetically pleasing pages we see today are made using CSS.
# CSS is used to add style to webpages.
# The fact that all pages for a company have the same style is usually a result that
# they all use the same CSS file. The general way these CSS files work is by defining
# how each of the elements of a webpage will look. The title, headings, itemized lists,
# tables, and links, for example, each receive their own style including font, color,
# size, and distance from the margin, among others.

# To do this CSS leverages patterns used to define these elements, referred to as selectors.
# An example of pattern we used in a previous video is table but there are many many more.
# If we want to grab data from a webpage and we happen to know a selector that is unique
# to the part of the page, we can use the html_nodes function.
```



```

# However, knowing which selector to use can be quite complicated. To demonstrate this we
# will try to extract the recipe name, total preparation time, and list of ingredients
# from this guacamole recipe. Looking at the code for this page, it seems that the task is
# impossibly complex. However, selector gadgets actually make this possible. SelectorGadget
# is piece of software that allows you to interactively determine what CSS selector you need
# to extract specific components from the webpage. If you plan on scraping data other than
# tables, we highly recommend you install it. A Chrome extension is available which permits
# you to turn on the gadget highlighting parts of the page as you click through, showing the
# necessary selector to extract those segments.

# For the guacamole recipe page we already have done this and determined that we need the
# following selectors:

h <- read_html("http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609")
recipe <- h %>% html_node(".o-AssetTitle__a-HeadlineText") %>% html_text()
prep_time <- h %>% html_node(".o-RecipeInfo__a-Description--Total") %>% html_text()
ingredients <- h %>% html_nodes(".o-Ingredients__a-ListItemText") %>% html_text()

# You can see how complex the selectors are. In any case we are now ready to extract what
# we want and create a list:

guacamole <- list(recipe, prep_time, ingredients)
guacamole

# Since recipe pages from this website follow this general layout, we can use this code to
# create a function that extracts this information:

get_recipe <- function(url){
  h <- read_html(url)
  recipe <- h %>% html_node(".o-AssetTitle__a-HeadlineText") %>% html_text()
  prep_time <- h %>% html_node(".o-RecipeInfo__a-Description--Total") %>% html_text()
  ingredients <- h %>% html_nodes(".o-Ingredients__a-ListItemText") %>% html_text()
  return(list(recipe = recipe, prep_time = prep_time, ingredients = ingredients))
}

```

```
# and then use it on any of their webpages:
```

```
get_recipe("http://www.foodnetwork.com/recipes/food-network-kitchen/pancakes-recipe-1913844")
```

```
# There are several other powerful tools provided by rvest. For example, the functions html_form,  
# set_values, and submit_form permit you to query a webpage from R. This is a more advanced topic  
# not covered here.
```

## Working with Strings

```
# Video and sample code for string functions
```

```
library(tidyverse)  
library(rvest)
```

```
# Since web pages and other formal documents use commas in numbers  
# to improve readability.  
# For example, we write the number 4,853,875 like this.  
# And it's easier to read than writing it like this with no commas.  
# Because this is such a common task, there's already a function,  
# parse_number(), that readily does this conversion.
```

```
# Now, what happens if your string includes double quotes?  
#   For example, if you want to write 10 inches like this--  
#   "10"  
# for this we have to use single quotes.  
# You can't use this code, because this is just the string  
# 10 followed by a double quote.  
# If you type this into R you get an error,  
# because you have an unclosed double quote.  
# So to avoid this, we can use the single quotes like this.  
# To make sure that it's working, in R we can use the function cat.  
# The function cat lets us see what the string actually looks like.
```

```

# So if we type cat(s), we will see 10" as we wanted.

# Now, what do we want to use string to be 5 feet written like this--
# 5'
# 5 and then the single quote.
# In this case we use the double quotes like this--
# "5'"
# you can see it works by using cat().
# So we've learn how to write five feet and 10 inches separately.
# But what if we want to write them together to represent
# five feet and 10 inches, like this?
# In this case, neither the single or a double quote will work.

# We have to escape the character like this:
s <- '5\'10"'
cat(s)

cat(" LeBron James is 6'8\" ")

# The following all generate errors because they are improperly formed
cat(' LeBron James is 6'8" ')
cat(` LeBron James is 6'8" `)
cat(" LeBron James is 6\'8" ")

```

## Search and Replace with regex

```

library(tidyverse)
library(dslabs)
library(readr)
library(readxl)
install.packages("htmlwidgets")

data("reported_heights")
setwd("C:/Users/jamesw/Documents/ds-education")

```

```

# String exercises - the problems set where heights are not in inches

not_inches <- function(x, smallest = 50, tallest = 84) {
  inches <- suppressWarnings(as.numeric(x))
  ind <- is.na(inches) | inches < smallest | inches > tallest
  ind
}

problems <- reported_heights %>% filter(not_inches(height)) %>% .$height
length(problems)

pattern <- "^[4-7]'\d{1,2}\"$"

problems[c(2, 10, 11, 12, 15)] %>% str_view(pattern)

str_subset(problems, "inches")
str_subset(problems, "'")

# We will simplify the pattern by no longer using the inches symbol at the end
# so 5'4 will denote 5 feet 4 inches. Now the pattern looks like this:

pattern <- "^[4-7]'\d{1,2}$"

# Before we run this, we will do some string replacement to replace "feet" and "inches"
# with the feet symbols.

problems %>% str_replace("feet|ft|foot", "'") %>%
  str_replace("inches|in|'|\\"", "'") %>%
  str_detect(pattern) %>% sum

# Another problem is spaces - we did not get matches where spaces exist
# Spaces can be represented by \s so we can change the pattern to the following

```

```

pattern_2 <- "^[4-7]'\s\d{1,2}\"$"
str_subset(problems, pattern_2)

# We want a pattern to permit spaces but not require them. The asterisk indicates
# one or more instances. Se we can improve the pattern by adding the * after \s

pattern_2 <- "^[4-7]'\s*\d{1,2}\"$"

# There are two other qualifiers. For none or once, we use ?
# For one or more, we use +
# Note the differences using this code:

yes <- c("5", "6", "5'10", "5 feet", "4'11")
data.frame(string = c("AB", "A1B", "A11B", "A111B", "A1111B"),
           none_or_more = str_detect(yes, "A1*B"),
           none_or_once = str_detect(yes, "A1?B"),
           once_or_more = str_detect(yes, "A1+B"))

# New pattern
not_inches <- function(x, smallest = 50, tallest = 84) {
  inches <- suppressWarnings(as.numeric(x))
  ind <- is.na(inches) | inches < smallest | inches > tallest
  ind
}

problems <- reported_heights %>% filter(not_inches(height)) %>% .$height
length(problems)

pattern <- "^[4-7]'\s*'\s*\d{1,2}$"
problems %>% str_replace("feet|ft|foot", "'") %>%
  str_replace("inches|in|'|\'", "'") %>%
  str_detect(pattern) %>% sum

# Why not use str_replace_all? It could have unintentional consequences

```

## Regex Exercises

```
library(tidyverse)
library(dslabs)
library(readr)
library(readxl)
install.packages("htmlwidgets")
```

```
data("reported_heights")
setwd("C:/Users/jamesw/Documents/ds-education")
```

```
schools <- c("U. Kentucky", "Univ New Hampshire", "Univ. of Massachusetts", "University Georgia", "U California", "California State University")
```

```
schools %>%
  str_replace("Univ\\.?.?|U\\.?.?", "University ") %>%
  str_replace("^University of |^University ", "University of ")
```

```
schools %>%
  str_replace("^Univ\\.?.?\\s|^U\\.?.?\\s", "University ") %>%
  str_replace("^University of |^University ", "University of ")
```

```
schools %>%
  str_replace("^Univ\\.\\.\\.\\s|^U\\.\\.\\.\\s", "University") %>%
  str_replace("^University of |^University ", "University of ")
```

```
schools %>%
  str_replace("^Univ\\.?.?\\s|^U\\.?.?\\s", "University") %>%
  str_replace("University ", "University of ")
```

## OUTPUTS

```
> schools <- c("U. Kentucky", "Univ New Hampshire", "Univ. of Massachusetts", "University Georgie", "U California", "California State University")
```



```

> schools %>%
+   str_replace("Univ\\.?.?|U\\.?.?", "University ") %>%
+   str_replace("^University of |^University ", "University of ")
[1] "University of Kentucky"
[2] "University of New Hampshire"
[3] "University of Massachusetts"
[4] "University of Georgia"
[5] "University of California"
[6] "California State University"
> schools <- c("U. Kentucky", "Univ New Hampshire", "Univ. of Massachusetts", "University Georgia", "U California", "California
State University")
> schools %>%
+   str_replace("Univ\\.?.?|U\\.?.?", "University ") %>%
+   str_replace("^University of |^University ", "University of ")
[1] "University of Kentucky"
[2] "University of New Hampshire"
[3] "University of Massachusetts"
[4] "University of Georgia"
[5] "University of California"
[6] "California State University"
>
> schools %>%
+   str_replace("^Univ\\.?.?\\s|^U\\.?.?\\s", "University ") %>%
+   str_replace("^University of |^University ", "University of ")
[1] "University of Kentucky" "University of New Hampshire"
[3] "University of Massachusetts" "University of Georgia"
[5] "University of California" "California State University"
>
> schools %>%
+   str_replace("^Univ\\.?.?\\s|^U\\.?.?\\s", "University") %>%
+   str_replace("^University of |^University ", "University of ")
[1] "UniversityKentucky" "Univ New Hampshire"
[3] "Universityof Massachusetts" "University of Georgia"
[5] "U California" "California State University"

```

```

>
> schools %>%
+   str_replace("^Univ\\.?.?\\s|^U\\.?.?\\s", "University") %>%
+   str_replace("University ", "University of ")
[1] "UniversityKentucky"          "UniversityNew Hampshire"
[3] "Universityof Massachusetts"  "University of Georgie"
[5] "UniversityCalifornia"        "California State University"

```

## Regex Groups in R

```

library(tidyverse)
library(dslabs)
library(readr)
library(readxl)
install.packages("htmlwidgets")

data("reported_heights")
setwd("C:/Users/jamesw/Documents/ds-education")

# Groups are a powerful aspect of regex that permits the extraction of values.
# Groups are defined using parentheses.
# They don't affect the pattern matching per se.
# Instead, it permits tools to identify specific parts of the pattern
# so we can extract them.
# So, for example, we want to change height
# like 5.6 to five feet, six inches.
# To avoid changing patterns such as 70.2, we'll
# require that the first digit be between four and seven--
# we can do that using the range operation--
# and that the second be none or more digits.
# We can do that using backslash, backslash d star.
# Let's start by defining a simple pattern that matches this.

pattern_without_groups <- "^[4-7],\\d*$"

```

```

# We want to extract the digits so that we can then form
# the new version using a single quote.
# These are two groups, so we encapsulate them with parentheses like this.

pattern_with_groups <- "^[4-7]),(\\d*)$"

yes <- c("5,9", "5,11", "6,", "6,1")
no <- c("5'9", "", "2,8", "6.1.1")
s <- c(yes, no)
str_detect(s, pattern_without_groups)
str_detect(s, pattern_with_groups)

# Once we define groups, we can use a function str_match
# to extract the values these groups define, like this.
# Look what happens if we write this code.

str_match(s, pattern_with_groups)

#      [,1]  [,2] [,3]
# [1,] "5,9"  "5"  "9"
# [2,] "5,11" "5"  "11"
# [3,] "6,"   "6"  ""
# [4,] "6,1"  "6"  "1"
# [5,] NA     NA   NA
# [6,] NA     NA   NA
# [7,] NA     NA   NA
# [8,] NA     NA   NA

# Note that the second and third columns contain feet and inches respectively.
# The first is the original pattern that was matched.
# If no match occurred, we see an N/A.
# Now we can understand the difference between the function str_extract
# and str_match.

```

```

# str_extract extracts only strings that match a pattern,
# not the values defined by the groups.

# Here's what happens with string extract.

str_extract(s, pattern_with_groups)

# [1] "5,9" "5,11" "6," "6,1" NA NA NA NA

# Another powerful aspect of groups is that you
# can refer to the extracted value in regex when searching and replacing.
# The regex special character for the i-th group is backslash, backslash, i.
# So backslash, backslash, 1 is the value extracted from the first group,
# and backslash, backslash, 2 is the value from the second group, and so on.
# So as a simple example, note that the following code
# will replace a comma by a period, but only if it is between two digits.

str_replace(s, pattern_with_groups, "\\1'\\2")

# [1] "5'9" "5'11" "6'" "6'1" "5'9" ",," "2,8" "6.1.1"

# Now we're ready to define a pattern that helps us convert all the x.y, x,y,
# and x y's to our preferred format.

# We need to adapt pattern underscore with groups to be a bit more flexible
# and capture all these cases. The pattern now looks like this.

pattern_with_groups <- "^[4-7]\\s*[,\\.\\s+]\\s*(\\d*)$"

# The caret means start of the string.
# Then four to seven means one digit between four and seven--four, five, six, or seven.
# Then the backslash, backslash, s, star means none or more white spaces.
# The next pattern means the fifth symbol is either comma, or dot, or at least one space.

```

```

# Then we have none or more white spaces again.
# Then we have none or more digits, and then the end of the string.

str_subset(problems, pattern_with_groups) %>%
  str_replace(pattern_with_groups, "\\1'\\2") %>% head

# EXERCISES

problems <- c("5.3", "5,5", "6 1", "5 .11", "5, 12")
pattern_with_groups <- "^[4-7])[,\\.](\\d*)$"
str_replace(problems, pattern_with_groups, "\\1'\\2")

# In the above, You forgot to check for any spaces in your regex pattern.
# While the first two entries of "problems" have commas and periods correctly
# replaced, the last three entries are not identified as part of the pattern
# and are not replaced.

problems <- c("5.3", "5,5", "6 1", "5 .11", "5, 12")
pattern_with_groups <- "^[4-7])[,\\.\\s](\\d*)$"
str_replace(problems, pattern_with_groups, "\\1'\\2")

# The new regex pattern now checks for one character,
# either a comma, period or space, between the first digit
# and the last one or two digits, and replaces it with an
# apostrophe (`). However, because your last two problem
# strings have additional space between the digits, they are not corrected.

```

## Testing and Improving

```

library(tidyverse)
library(dslabs)
library(readr)
library(readxl)
install.packages("htmlwidgets")

```

```

data("reported_heights")
setwd("C:/Users/jamesw/Documents/ds-education")

# Let's write a function that captures all the entries that
# can't be converted into numbers, remembering
# that some are in centimeters.
# We'll deal with those later.

not_inches_or_cm <- function(x, smallest = 50, tallest = 84) {
  inches <- suppressWarnings(as.numeric(x))
  ind <- !is.na(inches) &
    ((inches >= smallest & inches <= tallest) |
     (inches/2.54 >= smallest & inches/2.54 <= tallest))
  !ind
}

problems <- reported_heights %>% filter(not_inches_or_cm(height)) %>% .$height
length(problems)

# Here, we leverage the pipe-- one of the advantages of using a stringr.
# We use the pipe to concatenate the different replacements
# that we have just performed.

# Then we define the pattern and then, we go and try to see how many we match.

converted <- problems %>%
  str_replace("feet|foot|ft", "'") %>%
  str_replace("inches|in|'|\"", "") %>%
  str_replace("^[4-7])\\s*[.,\\s+]\\s*(\\d*)$", "\\1'\\2")

pattern <- "^[4-7]\\s*'\\s*\\d{1,2}$" # Note curly brackets near the end
index <- str_detect(converted, pattern)
mean(index)

```



```
# [1] 0.615
# We are matching over half now. Let's go after the rest of the cases
# One problem is that people exactly 5 or 6 feet did not enter inches

# Some of the inches were entered with decimal points.
# For example, 5 feet and 7.5 inches. Our pattern only looks for two digits.
# We also have cm, European punctuation, etc.
# It may not be worthwhile to fix them
```

### Using Groups and Quantifiers

Four clear patterns of entries have arisen along with some other minor problems:

1. Many students measuring exactly 5 or 6 feet did not enter any inches. For example, 6' - our pattern requires that inches be included.
2. Some students measuring exactly 5 or 6 feet entered just that number.
3. Some of the inches were entered with decimal points. For example 5'7.5". Our pattern only looks for two digits.
4. Some entries have spaces at the end, for example 5 ' 9.
5. Some entries are in meters and some of these use European decimals: 1.6, 1,7.
6. Two students added cm.
7. One student spelled out the numbers: Five foot eight inches.

It is not necessarily clear that it is worth writing code to handle all these cases since they might be rare enough. However, some give us an opportunity to learn some more regex techniques so we will build a fix.

### Case 1

For case 1, if we add a '0 to, for example, convert all 6 to 6'0, then our pattern will match. This can be done using groups using the following code:

```
yes <- c("5", "6", "5")
no <- c("5'", "5' '", "5'4")
```

```
s <- c(yes, no)
str_replace(s, "^[4-7])$", "\\1'0")
```

The pattern says it has to start (^), be followed with a digit between 4 and 7, and then end there (\$). The parenthesis defines the group that we pass as \\1 to the replace regex.

## Cases 2 and 4

We can adapt this code slightly to handle case 2 as well which covers the entry 5'. Note that the 5' is left untouched by the code above. This is because the extra ' makes the pattern not match since we have to end with a 5 or 6. To handle case 2, we want to permit the 5 or 6 to be followed by no or one symbol for feet. So we can simply add '{0,1}' after the ' to do this. We can also use the none or once special character ?. As we saw previously, this is different from \* which is none or more. We now see that this code also handles the fourth case as well:

```
str_replace(s, "^[56])'?$", "\\1'0")
```

Note that here we only permit 5 and 6 but not 4 and 7. This is because heights of exactly 5 and exactly 6 feet tall are quite common, so we assume those that typed 5 or 6 really meant either 60 or 72 inches. However, heights of exactly 4 or exactly 7 feet tall are so rare that, although we accept 84 as a valid entry, we assume that a 7 was entered in error.

## Case 3

We can use quantifiers to deal with case 3. These entries are not matched because the inches include decimals and our pattern does not permit this. We need allow the second group to include decimals and not just digits. This means we must permit zero or one period . followed by zero or more digits. So we will use both ? and \*. Also remember that for this particular case, the period needs to be escaped since it is a special character (it means any character except a line break).

So we can adapt our pattern, currently `^[4-7]\\s*'\s*\d{1,2}$` to permit a decimal at the end:

```
pattern <- "^[4-7]\\s*'\s*(\\d+\\.?\d*)$"
```

## Case 5

Case 5, meters using commas, we can approach similarly to how we converted the `x.y` to `x'y`. A difference is that we require that the first digit is 1 or 2:

```
yes <- c("1,7", "1, 8", "2, " )
no <- c("5,8", "5,3,2", "1.7")
s <- c(yes, no)
str_replace(s, "^[12])\\s*,\\s*(\\d*)$", "\\1\\.\\2")
```

We will later check if the entries are meters using their numeric values.

## Trimming

In general, spaces at the start or end of the string are uninformative. These can be particularly deceptive because sometimes they can be hard to see:

```
s <- "Hi "  
cat(s)  
identical(s, "Hi")
```

This is a general enough problem that there is a function dedicated to removing them: `str_trim`.

```
str_trim("5 ' 9 ")
```

## To upper and to lower case

One of the entries writes out numbers as words: Five foot eight inches. Although not efficient, we could add 12 extra `str_replace` to convert zero to 0, one to 1, and so on. To avoid having to write two separate operations for Zero and zero, One and one, etc., we can use the `str_to_lower` function to make all words lower case first:

```
s <- c("Five feet eight inches")  
str_to_lower(s)
```

## Putting it into a function

We are now ready to define a procedure that handles converting all the problematic cases.

We can now put all this together into a function that takes a string vector and tries to convert as many strings as possible to a single format. Below is a function that puts together the previous code replacements:

```

convert_format <- function(s){
  s %>%
    str_replace("feet|foot|ft", "'") %>% #convert feet symbols to '
    str_replace_all("inches|in|'|\"|cm|and", "") %>% #remove inches and other symbols
    str_replace("^[4-7]\\s*[.\\s+\\s*(\\d*)$", "\\1'\\2") %>% #change x.y, x,y x y
    str_replace("^[56]]'?$", "\\1'0") %>% #add 0 when to 5 or 6
    str_replace("^[12]\\s*[.\\s*(\\d*)$", "\\1\\.\\2") %>% #change european decimal
    str_trim() #remove extra space
}

```

We can also write a function that converts words to numbers:

```

words_to_numbers <- function(s){
  str_to_lower(s) %>%
    str_replace_all("zero", "0") %>%
    str_replace_all("one", "1") %>%
    str_replace_all("two", "2") %>%
    str_replace_all("three", "3") %>%
    str_replace_all("four", "4") %>%
    str_replace_all("five", "5") %>%
    str_replace_all("six", "6") %>%
    str_replace_all("seven", "7") %>%
    str_replace_all("eight", "8") %>%
    str_replace_all("nine", "9") %>%
    str_replace_all("ten", "10") %>%
    str_replace_all("eleven", "11")
}

```

Now we can see which problematic entries remain:

```

converted <- problems %>% words_to_numbers %>% convert_format
remaining_problems <- converted[not_inches_or_cm(converted)]
pattern <- "^[4-7]\\s*'\\s*\\d+\\.?\\d*$"
index <- str_detect(remaining_problems, pattern)
remaining_problems[!index]

```