Entering / cleaning data 1

Getting data into R

Basics of getting data into R

Basic approach:

- Download data to your computer
- Save the data in your R Project directory for the project you're using it for or in a subdirectory within that directory ("data" is a good name for this subdirectory)
- Read data into R (functions in `readr`: `read_csv`, `read_table`, `read_delim`, `read_fwf`, etc.)
- Check to make sure the data came in correctly (`dim`, `ncol`, `nrow`, `head`, `tail`, `str`, `colnames`)

Reading data into R

What kind of data can you get into R?

The sky is the limit. . .

- **Flat files**
- Files from other statistical packages (SAS, Excel, Stata, SPSS)
- Tables on webpages
- Data in a database (e.g., SQL)
- Data stored in XML and JSON
- Complex data formats (e.g., netCDF files from climate folks, MRI data stored in Analyze, NIfTI, and DICOM formats)
- Data through APIs (e.g., GoogleMaps, Twitter, many government agencies)
- Incredibly messy data using scan and readLines

Flat files

R can read in data from *a lot* of different formats. The only catch: you need to tell R how to do it.

To start, we'll look at **flat files**, which are plain text files (i.e., you can read them when you open them in a text editor, unlike a file in a binary format, like an Excel or Word file) with a two-dimenional structure (a row for each observation and a column for each variable).
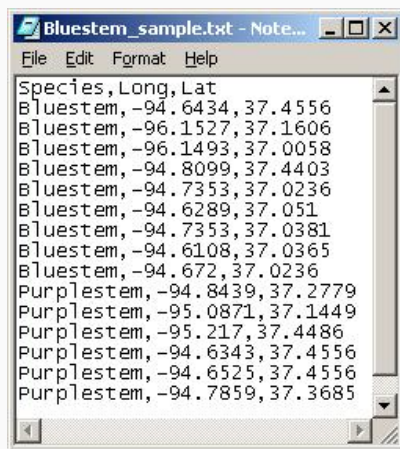
Types of flat files

There are two main types of flat files:

1. **Fixed width files**: Each column is a certain number of characters wide. (If you printed it out, you could draw vertical lines that separate the columns.)

2. **Delimited files**: In each row, a certain symbol (**delimiter**) separates the data into columns values for that observation.

   - ".csv": Comma-separated values
   - ".tab", ".tsv": Tab-separated values
   - Other possible delimiters: colon, semicolon, pipe ("|")

See if you can identify what types of files the following files are. . .

What type of file?


```
Bluestem_sample.txt - Note...
File  Edit  Format  Help
Species,Long,Lat
Bluestem,-94.6434,37.4556
Bluestem,-96.1527,37.1606
Bluestem,-96.1493,37.0058
Bluestem,-94.8099,37.4403
Bluestem,-94.7353,37.0236
Bluestem,-94.6289,37.051
Bluestem,-94.7353,37.0381
Bluestem,-94.6108,37.0365
Bluestem,-94.672,37.0236
Purplestem,-94.8439,37.2779
Purplestem,-95.0871,37.1449
Purplestem,-95.217,37.4486
Purplestem,-94.6343,37.4556
Purplestem,-94.6525,37.4556
Purplestem,-94.7859,37.3685
```
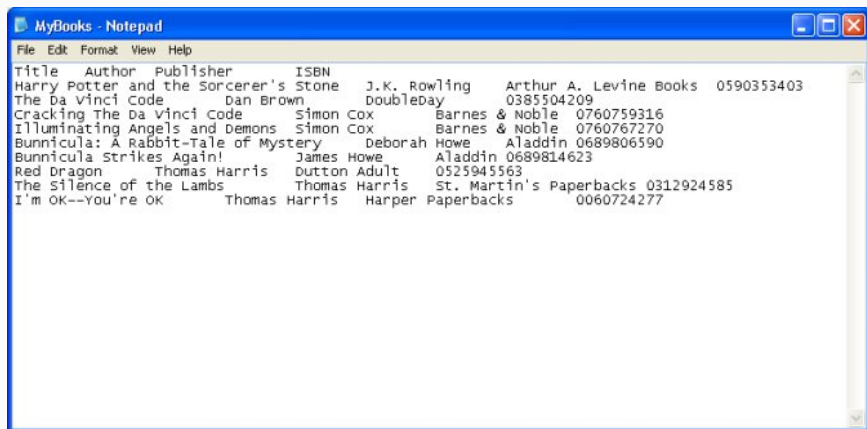
What type of file?
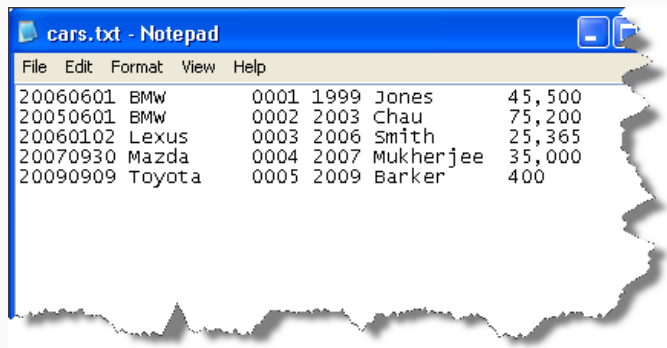
What type of file?

```
H|20110606|pizza.txt|
D|10|Chicken Pesto|20|23|30|5.5|7.4|9.9||
D|10|Meatball|10|53|60|6.5|8.4|10.9|
D|10|Fire Cracker|3|13|60|5.8|7.9|11.9|
D|10|Spinach|1|2|5|5.5|7.0|8.8|
D|10|BBQ Chicken|35|102|95|6.5|7.9|10.9|
D|10|Vegetarian|5|13|28|4.5|7.9|9.5|
D|10|Mexican|11|33|36|5.5|7.4|9.9|
D|10|The Monaco|22|53|7|5.5|7.5|8.9|
D|10|Chilli Prawn|5|5|6|5.5|7.4|9.9|
D|10|Chefs Special|8|18|40|5.8|7.8|9.8|
D|10|Marinara|3|17|41|5.5|7.4|9.0|
D|10|Supreme|50|52|58|5.5|7.4|9.2|
D|10|Margherita|9|19|87|5.0|7.0|8.0|
D|10|Napoli|60|85|66|5.2|7.2|9.2|
D|10|Caprice|31|32|38|5.5|7.4|9.3|
D|10|Ham and Pineapple|18|39|28|5.8|7.0|9.0|
T|16|
```

What type of file?


```
MyBooks - Notepad
File Edit Format View Help
Title    Author    Publisher        ISBN
Harry Potter and the Sorcerer's Stone    J.K. Rowling    Arthur A. Levine Books  0590353403
The Da Vinci Code        Dan Brown        DoubleDay        0385504209
Cracking The Da Vinci Code    Simon Cox    Barnes & Noble  0760759316
Illuminating Angels and Demons  Simon Cox    Barnes & Noble  0760767270
Bunnicula: A Rabbit-Tale of Mystery    Deborah Howe    Aladdin 0689806590
Bunnicula Strikes Again!    James Howe    Aladdin 0689814623
Red Dragon    Thomas Harris    Dutton Adult    0525945563
The Silence of the Lambs    Thomas Harris    St. Martin's Paperbacks 0312924585
I'm OK--You're OK        Thomas Harris    Harper Paperbacks        0060724277
```

What type of file?

Title,Subtitle,Larger Work,Contributor #1,Contributor #2,Contributor #3,Contributor
#4,Genre,Publisher,Published Location,Date
Published,Instrumentation,Key,Location,Indiana Connection,Sheet Music
Consortium,Notes,Complete
"""A""" You're Adorable",The alphabet song,,Buddy Kaye,Sidney Lippman,Fred Wise,,Popular
standard,Laurel Music Corporation,"New York, NY",1948,Voice and
piano/guitar or ukulele,C Major,,None,Yes,Perry Como pictured on cover,
"Aba Daba Honey Moon, The",,"Two Weeks with Love"" Motion Picture",Arthur Fields,Walter
Donovan,,"Popular Standard, Movie Selection",Leo Feist Inc.,"New
York, NY",1942,Voice and Piano,C Minor,,None,Yes,,
Abi Bezunt,,"""Mamele"" Motion Picture",Abraham Ellstein,Molly Picon,,,"Popular Standard,
Movie Selection",Metro Music Co.,"New York, NY",1939,Voice and
Piano,E Minor,,None,No,Molly Picon pictured on cover,
Abdul the Bulbul Ameer,,,Bob Kaai,Jim Smock,,,Popular Standard,Calumet Music Co.
,"Chicago, IL",1935,"Voice, Piano, Hawaiian Guitar, Ukulele",G
Major,,None,Yes,Ben Pollack pictured on cover,
About A Quarter to Nine,,"""Go Into Your Dance"" Motion Picture",Harry Warren,Al
Dubin,,"Popular Standard, Movie Selection",M. Witmark & Sons,"New York,
NY",1935,"Voice, Piano, Guitar, Ukelele",E Minor,,None,No,Al Jolson and Ruby Keeler
pictured on cover,
Absent,,,John. W. Metcalf,Catherine Young Glen,,,Popular Standard,Arthur P.
Schmidt,"Boston, MA",1899,Voice and Piano,G Major,,None,Yes,,
The Academy Two-Step,,Barclay Walker,,,,Popular Standard,Carlin & Lennox,"Indianapolis,
IN",,Piano,F Major,,Composer,No,,
Ac-cent-tchu-ate the Positive,Mister In Between,"""Here Come the Waves"" Motion
Picture",Harold Arlen,Johnny Mercer,,,"Popular Standard, Movie
Selection",Edwin H. Morris & Co.,"New York, NY",1944,"Voice, Piano, Guitar",F
Major,,None,Yes,Bing Crosby and Betty Hutton pictured on cover,
Across the Alley From the Alamo,,,Joe Greene,,,,Popular Standard,Leslie Music

What type of file?

```
1000233    Miralda      John
1000234    Faley        Nick
1000235    Baylog       Cathy
1000236    Gallardo     Mike
1000237    Christian    Daniel
1000238    Baufield     Daniel
1000239    Frazier      Robert
1000240    Garrido      Edward
1000241    Williams     Zachary
1000242    Morel        David
           Padilla      Damian
1000244    Rosenberg    Wayne
1000245    Blanchard    Phong S
1000246    Wiggins      David
1000247    Miller       Jeffrey
1000248    Coon         Terry
1000249    Chretien     Walter
1000250    Myers        Timothy

1000233    Miralda      John
1000234    Faley        Nick
1000235    Baylog       Cathy
```

Types of flat files

Flat files will often end in file extensions like ".txt", ".csv", ".fwf", and ".tsv".

To figure out the structure of a flat file, start by opening it in a text editor. RStudio can also be used as a text editor to open and explore flat files (right click on the file name and then choose "Open With" and "RStudio").

Reading in flat files

R can read any of the types of files we just looked at by using one of the functions from the readr package:

File type	General function
Delimited	read_delim
Fixed width	read_fwf

You will just need to be able to clearly tell R *how* to read the file in, including what type of flat file it is and what delimiter it uses.

Reading in flat files

For example, the file "AWOIS_Wrecks_KnownYear.tab" is a flat delimited file with tabs as delimiters containing the subset of the Office of the Coast Survey's Automated Wreck and Obstruction Information System (AWOIS) for which the year the vessel sank is known.

You can download this file by going to this link and using the "Raw" button in the top right hand corner (right click and select "Download Linked File").

Reading in flat files

If save this file in your working directory, to read it in and assign it the name shipwrecks, you can run:

```
library("readr")
shipwrecks <- read_delim("AWOIS_Wrecks_KnownYear.tab",
                         delim = "\t")

# Check out a subset of the data
library("dplyr")
slice(.data = select(.data = shipwrecks, c(2, 4, 5, 9)), 1:4)

## # A tibble: 4 x 4
##     VESSLTERMS    LATDEC LONDEC YEARSUNK
##     <chr>          <dbl>  <dbl>    <dbl>
## 1 SUBCHASER 187    37.3  -75.5     1918
## 2 BIRCH LAKE       37.3  -75.6     1943
## 3 PACIFIC          37.3  -75.6     1925
## 4 UNKNOWN          37.3  -75.7     1916
```

15

readr family of functions

Some of the interesting options with the readr family of functions are:

Option	Description
skip	How many lines of the start of the file should you skip?
col_names	What would you like to use as the column names?
col_types	What would you like to use as the column types?
n_max	How many rows do you want to read in?
na	How are missing values coded?

Reading in flat files

The "daily_show_guests.csv" file you worked with in the previous In-Course Exercise is a delimited flat file with commas as the delimiters. **It also has four lines of information about the data, before the actual data begins.**

```
1  ## Obtained from GitHub page of FiveThirtyEight under the
2  ## Creative Commons Attribution 4.0 International License
3  ## https://github.com/fivethirtyeight/data/tree/master/daily-show-guests
4  ##
5  YEAR,GoogleKnowlege_Occupation,Show,Group,Raw_Guest_List
6  1999,actor,1/11/99,Acting,Michael J. Fox
7  1999,Comedian,1/12/99,Comedy,Sandra Bernhard
8  1999,television actress,1/13/99,Acting,Tracey Ullman
```

Reading in flat files

You can handle this by using the skip option to tell R to skip the first four lines:

```r
read_delim("daily_show_guests.csv", delim = ",", skip = 4)
```

```
## # A tibble: 2,693 x 5
##     YEAR GoogleKnowlege_~ Show  Group
##    <dbl> <chr>            <chr> <chr>
##  1  1999 actor            1/11~ Acti~
##  2  1999 Comedian         1/12~ Come~
##  3  1999 television actr~ 1/13~ Acti~
##  4  1999 film actress     1/14~ Acti~
##  5  1999 actor            1/18~ Acti~
##  6  1999 actor            1/19~ Acti~
##  7  1999 Singer-lyricist  1/20~ Musi~
##  8  1999 model            1/21~ Media
##  9  1999 actor            1/25~ Acti~
## 10  1999 stand-up comedi~ 1/26~ Come~
## # ... with 2,683 more rows, and 1 more variable:
## #   Raw_Guest_List <chr>
```

`readr` family of functions

Many members of the `readr` package that read delimited files are doing the same basic thing. The only difference is what defaults they have for the delimiter (`delim`).

Some key members of the `readr` family for delimited data:

Function	Delimiter
`read_csv`	comma
`read_csv2`	semi-colon
`read_table2`	whitespace
`read_tsv`	tab

readr family of functions

For any type of delimited flat files, you can also use the more general
read_delim function to read in the file. However, you will have to specify
yourself what the delimiter is (e.g., delim = "," for a comma-separated
file).

For example, the following two calls do the same thing:

```
read_delim("daily_show_guests.csv", delim = ",", skip = 4)
read_csv("daily_show_guests.csv", skip = 4)
```

readr family of functions

The readr package also includes some functions for reading in fixed width files:

- read_fwf
- read_table

These allow you to specify field widths for each fixed width field, but they will also try to determine the field-widths automatically.

Reading data from other files types

You can also read data in from a variety of other file formats, including:

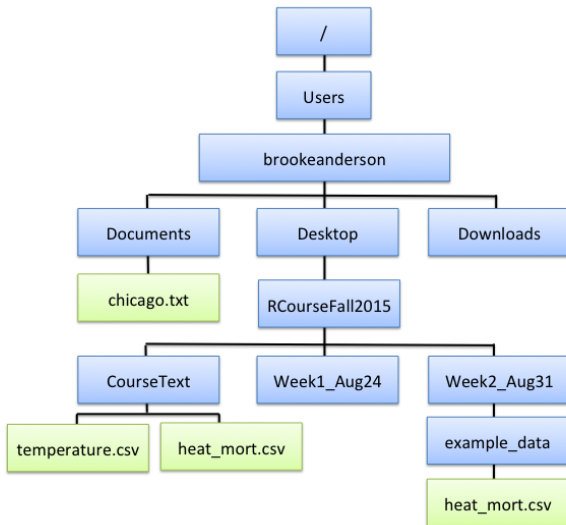| File type	Function	Package
Excel	read_excel	readxl
SAS	read_sas	haven
SPSS	read_spss	haven
Stata	read_stata	haven

We'll take break here to work on the start of the In-Course Exercise (Sections 2.7.1 and 2.7.2).

Directory structure

Computer directory structure

Working directory

The **working directory** is the directory within your directory structure from which your R session is currently working.
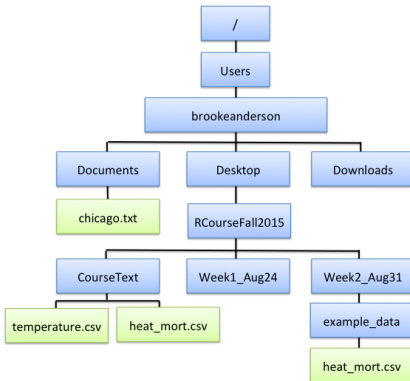
When you use **R Projects** to organize your work and files in R, anytime you open one of your R Projects, your working directory will automatically be that project's directory.

Working directory

To confirm this, open one of your R Projects and print out your working directory using the function getwd():

```
getwd()
```

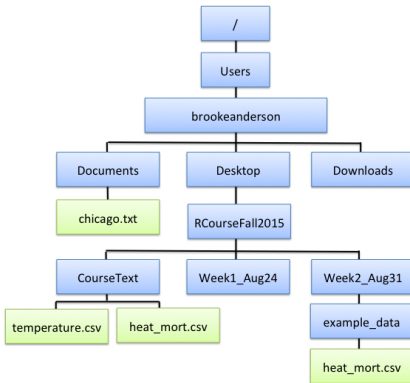```
[1] "/Users/brookeanderson/Desktop/RCourseFall2015"
```

## Working directory

You can use the list.files() function to list all the files in your current working directory:
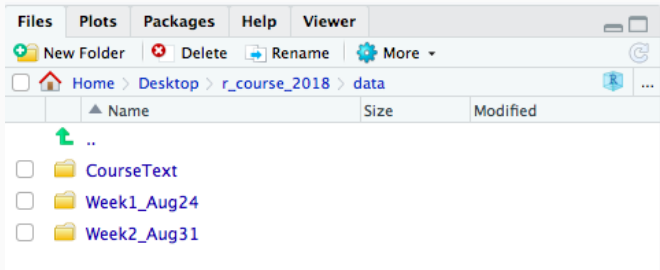
```r
list.files()
```

```
[1] "CourseText"    "Week1_Aug24"    "Week2_Aug31"
```

Working directory

The "Files" pane in RStudio (often on the lower right) will also show you the files available in your current working directory.

This should line up with what you get if you run list.files().

Working directory

When you run list.files(), if there is a name without a file extension, it's probably the name of a **subdirectory** of your current working directory. To list the files in one of these subdirectories, you can use list.files with that subdirectory's name.

```
list.files("CourseText")
```

```
[1]  "temperature.csv"      "heat_mort.csv"
```
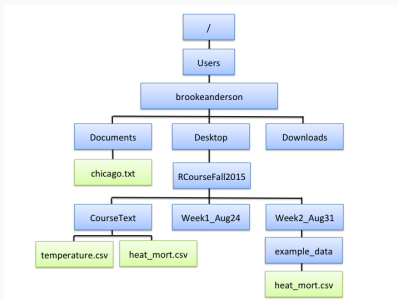
Relative versus absolute pathnames

When you want to reference a directory or file that is not in your working directory, you need to give R the directions for how to find the file. You can use one of two types of pathnames:

- *Relative pathname*: How to get to the file or directory from your current working directory
- *Absolute pathname*: How to get to the file or directory from anywhere on the computer

Relative versus absolute pathnames

Say your current working directory was
/Users/brookeanderson/RProgrammingForResearch and you wanted
to get into the subdirectory data. Here are examples of referencing that
subdirectory using the two types of pathnames:

Absolute:

"/Users/brookeanderson/RProgrammingForResearch/data"

Relative:

"data"

Relative versus absolute pathnames

Both methods of writing filenames have their own advantages and disadvantages:

- *Relative pathname*: Which file you are indicating depends on which working directory you are in, which means that your code will break if you try to re-run it from a different working directory. However, relative pathways in your code make it easier for you to share a working version of a project with someone else. For most of this course, we will focus on using relative pathnames, especially when you start collaborating.
- *Absolute pathname*: No matter what working directory you're in, it is completely clear to your computer which file you mean when you use an absolute pathname. However, your code will not work on someone else's computer without modifications (because the structure of their computer's full directory will be different).

Relative versus absolute pathnames

I **strongly** recommend saving your data files somewhere in the directory structure of the R Project in which you're working and then using **relative pathnames** to reference that file when you need to read it in.

This practice establishes good habits for making your research computationally reproducible.

Getting around directories

There are a few abbreviations you can use to represent certain relative locations:

Shorthand	Meaning
.	Current working directory
..	One directory up from current working directory (parent directory)
../..	Two directories up from current working directory
../data	The 'data' subdirectory of the parent directory

One other useful abbreviation is ~, which is shorthand for your **home directory**. On Windows, this will take you to your "Documents" folder.

Relative versus absolute pathnames

Here are some examples of relative pathnames that use these abbreviations:

If `data` is a subdirectory of your current parent directory (i.e., from your working directory, you need to go "up and over"):

`"../data"`

If `data` is a subdirectory of the subdirectory `Ex` of your current working directory:

`"Ex/data"`

Reading in an online flat file

Once you understand this idea of giving R directions to a file to read it, you shouldn't be too surprised that R can also do this for flat files that are hosted online.

In this case, the file isn't even on your computer, so you need to give R the directions to find it online. You do that by putting the file's online address as the file name. For example, to read in the shipwreck data directly from GitHub, you can run:

```r
shipwreck_url <- paste0("https://raw.githubusercontent.com/",
                        "geanders/RProgrammingForResearch/",
                        "master/data/",
                        "AWOIS_Wrecks_KnownYear.tab")
shipwrecks <- read_tsv(shipwreck_url)
```

Taking advantage of `paste0`

You can create an object with your directory name using paste0, and then use that to set your directory. We'll take a lot of advantage of this for reading in files.

The convention for paste0 is:

```
## Generic code
[object name] <- paste0("[first thing you want to paste]",
                        "[what you want to add to that]",
                        "[more you want to add]")
```

Taking advantage of `paste0`

Here's an example:

```r
shipwreck_url <- paste0("https://raw.githubusercontent.com/",
                        "geanders/RProgrammingForResearch/",
                        "master/data/",
                        "AWOIS_Wrecks_KnownYear.tab")
shipwrecks <- read_tsv(shipwreck_url)
```

We'll take a break now to do the next part of the in-course exercise (Section 2.7.3).

Data cleaning

Cleaning data

Common data-cleaning tasks include:

Task	dplyr function
Renaming columns	rename
Selecting certain columns	select
Adding or changing columns	mutate
Limiting to certain rows	slice
Filtering to certain rows	filter
Arranging rows	arrange

The "tidyverse"

Today, we'll talk about using functions from the dplyr and lubridate packages, which are both part of the "tidyverse", like the readr package.

To use these functions, you'll need to load those packages:

```
library("dplyr")
library("lubridate")
```

Cleaning data

As an example, let's look at the Daily Show data:

```r
daily_show <- read_csv("data/daily_show_guests.csv",
                       skip = 4)

head(daily_show, 3)

## # A tibble: 3 x 5
##     YEAR GoogleKnowlege_~ Show  Group
##    <dbl> <chr>            <chr> <chr>
## 1  1999 actor            1/11~ Acti~
## 2  1999 Comedian         1/12~ Come~
## 3  1999 television actr~ 1/13~ Acti~
## # ... with 1 more variable: Raw_Guest_List <chr>
```

Re-naming columns

A first step is often re-naming columns. It can be hard to work with a column name that is:

- long
- doesn't following the naming rules for R objects
- includes upper case

Several of the column names in `daily_show` have some of these issues:

```
colnames(daily_show)
```

```
## [1] "YEAR"
## [2] "GoogleKnowlege_Occupation"
## [3] "Show"
## [4] "Group"
## [5] "Raw_Guest_List"
```

Renaming columns

To rename these columns, use rename. The basic syntax is:

```
## Generic code
rename(dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
```

If you want to change column names in the saved object, be sure you reassign the object to be the output of rename.

Renaming columns

To rename columns in the `daily_show` data, then, use:

```r
daily_show <- rename(daily_show,
                     year = YEAR,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
head(daily_show, 3)

## # A tibble: 3 x 5
##    year job           date    category guest_name
##   <dbl> <chr>         <chr>   <chr>    <chr>
## 1  1999 actor         1/11/~  Acting   Michael J. ~
## 2  1999 Comedian      1/12/~  Comedy   Sandra Bern~
## 3  1999 television a~ 1/13/~  Acting   Tracey Ullm~
```

Renaming columns

As a quick check, what is the difference between these two calls?

```r
# 1.
rename(daily_show,
       year = YEAR,
       job = GoogleKnowlege_Occupation,
       date = Show,
       category = Group,
       guest_name = Raw_Guest_List)
```

```r
# 2.
daily_show <- rename(daily_show,
                     year = YEAR,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
```

Selecting columns

Next, you may want to select only some columns of the dataframe. You can use the select function for this.

In this case, the basic structure of this command is:

```
## Generic code
select(dataframe, column_name_1, column_name_2, ...)
```

Where column_name_1, column_name_2, etc., are the names of the columns you want to keep.

Selecting columns

For example, to select all columns except year (since that information is already included in date), run:

```
select(daily_show, job, date, category, guest_name)
```

```
## # A tibble: 2,693 x 4
##    job              date    category guest_name
##    <chr>            <chr>   <chr>    <chr>
##  1 actor            1/11/99 Acting   Michael J. Fox
##  2 Comedian         1/12/99 Comedy   Sandra Bernha~
##  3 television act~  1/13/99 Acting   Tracey Ullman
##  4 film actress     1/14/99 Acting   Gillian Ander~
##  5 actor            1/18/99 Acting   David Alan Gr~
##  6 actor            1/19/99 Acting   William Baldw~
##  7 Singer-lyricist  1/20/99 Musician Michael Stipe
##  8 model            1/21/99 Media    Carmen Electra
##  9 actor            1/25/99 Acting   Matthew Lilla~
## 10 stand-up comed~  1/26/99 Comedy   David Cross
```

Selecting columns

As a reminder, we could have selected these columns using position, too:

```
select(.data = daily_show, 2:5)
```

The select function has some cool extra options, including:

- Selecting all columns that start with a certain pattern
- Selecting all columns that end with a certain pattern
- Selecting all columns that contain a certain pattern

Selecting columns

The select function also provides some time-saving tools. For example, in the last example, we wanted all the columns except one. Instead of writing out all the columns we want, we can use - with the columns we don't want to save time:

```
daily_show <- select(daily_show, -year)
head(daily_show, 3)

## # A tibble: 3 x 4
##   job                date   category guest_name
##   <chr>              <chr>  <chr>    <chr>
## 1 actor              1/11/~ Acting   Michael J. Fox
## 2 Comedian           1/12/~ Comedy   Sandra Bernha~
## 3 television actre~  1/13/~ Acting   Tracey Ullman
```

Selecting columns

Another cool trick with select is that, if you want to keep several columns in a row, you can use a colon (:) with column names (rather than column position numbers) to select those columns:

```
daily_show <- select(daily_show, job:guest_name)
```

This call says that we want to select all columns from the one named "job" to the one named "guest_name".

Add or change columns

You can change a column or add a new column using the `mutate` function. That function has the syntax:

```
# Generic code
mutate(dataframe,
       changed_column = function(changed_column),
       new_column = function(other arguments))
```

- If you want to just **change** a column (in place), use its original name on the left of the equation.
- If you want to **add** a new column, use a new column name (you pick what that name will be) on the left of the equation (this will be the name of the new column).

Add or change columns

For example, the job column in daily_show sometimes uses upper case and sometimes does not:

```
head(unique(daily_show$job), 10)
```

```
## [1] "actor"              "Comedian"
## [3] "television actress" "film actress"
## [5] "Singer-lyricist"    "model"
## [7] "stand-up comedian"  "actress"
## [9] "comedian"           "Singer-songwriter"
```

Add or change columns

There is a package called stringr for working with character strings.

We could use the str_tolower function from the stringr package to
make all listings lowercase:

```
library("stringr")
daily_show <- mutate(daily_show, job = str_to_lower(job))
head(daily_show, 3)

## # A tibble: 3 x 4
##   job               date   category guest_name
##   <chr>             <chr>  <chr>    <chr>
## 1 actor             1/11/~ Acting   Michael J. Fox
## 2 comedian          1/12/~ Comedy   Sandra Bernha~
## 3 television actre~ 1/13/~ Acting   Tracey Ullman
```

Vectorized operations in R

[Vectorized operations]

We'll take a break now and do section 2.6.4 of the In-Course Exercise.

Dates in R

A common task when changing or adding columns is to change the class of some of the columns. This is especially common for dates, which will often be read in as a character vector when reading data into R.

Vector classes

Here are a few common vector classes in R:

Class	Example
character	"Chemistry", "Physics", "Mathematics"
numeric	10, 20, 30, 40
factor	Male [underlying number: 1], Female [2]
Date	"2010-01-01" [underlying number: 14,610]
logical	TRUE, FALSE [underlying numbers: 1, 0]

Vector classes

To find out the class of a vector, you can use class():

```
class(daily_show$date)
```

```
## [1] "character"
```

Note: You can use str to get information on the classes of all columns in a dataframe. It's also printed at the top of output when you print a tibble.

Vector classes

For some of these vector classes, there are specific packages for working
with vectors in that class:

Class	Package
character	stringr
factor	forcats
Date	lubridate

`lubridate` **package**

The `lubridate` package is helpful for working with vectors of dates or date-times.

You will see dates represented in many different ways. For example, October might be included in data as "October", "Oct", or "10". Further, the way the elements are separated can vary.

Computers are very literal, so this ambiguity can be confusing for them.

lubridate package

The lubridate package has a number of functions for converting character strings into dates (or date-times). To decide which one to use, you just need to know the order of the elements of the date in the character string.

For example, here are some commonly-used lubridate functions:

lubridate function	Order of date elements
ymd	year-month-day
dmy	day-month-year
mdy_hm	month-day-year-hour-minute
ymd_hms	year-month-day-hour-minute-second

(Remember, you can use vignette("lubridate") and ?lubridate to get help with the lubridate package.)

lubridate **package**

In many cases you can use functions from the lubridate package to parse dates pretty easily.

For example, if you have a character string with the date in the order of *year-month-day*, you can use the ymd function from lubridate to convert the character string to the Date class. For example:

```r
library("lubridate")
my_date <- ymd("2008-10-13")
class(my_date)

## [1] "Date"
```

lubridate **package**

The functions in lubridate are pretty good at working with different ways of expressing date and time elements intelligently:

```r
mdy("10-31-2017")
```

```
## [1] "2017-10-31"
```

```r
dmy("31 October 2017")
```

```
## [1] "2017-10-31"
```

lubridate **package**

There are lubridate functions that can parse date-times, too:

```
ymd_hms("2017/10/31--17:33:10")
```

```
## [1] "2017-10-31 17:33:10 UTC"
```

```
mdy_hm("Oct. 31, 2017 5:33PM", tz = "MST")
```

```
## [1] "2017-10-31 17:33:00 MST"
```

Converting to `Date` class

We can use the `mdy` function from `lubridate` to convert the `date` column in the `daily_show` dataset to a Date class:

```
daily_show <- mutate(daily_show, date = mdy(date))
head(daily_show, 3)

## # A tibble: 3 x 4
##   job            date       category guest_name
##   <chr>          <date>     <chr>    <chr>
## 1 actor          1999-01-11 Acting   Michael J. F~
## 2 comedian       1999-01-12 Comedy   Sandra Bernh~
## 3 television ac~ 1999-01-13 Acting   Tracey Ullman

class(daily_show$date)

## [1] "Date"
```

Converting to `Date` class

Once you have an object in the `Date` class, you can do things like plot by date, calculate the range of dates, and calculate the total number of days the dataset covers:

```r
range(daily_show$date)
```

```
## [1] "1999-01-11" "2015-08-05"
```

```r
diff(range(daily_show$date))
```

```
## Time difference of 6050 days
```

lubridate **package**

The lubridate package also includes functions to pull out certain elements of a date. For example, we could use wday to create a new column with the weekday of each show:

```
daily_show <- mutate(daily_show,
                     show_day = wday(date, label = TRUE))
head(select(daily_show, date, show_day), 3)

## # A tibble: 3 x 2
##   date       show_day
##   <date>     <ord>
## 1 1999-01-11 Mon
## 2 1999-01-12 Tue
## 3 1999-01-13 Wed
```

Other functions in `lubridate` for pulling elements from a date include:

- `mday`: Day of the month
- `yday`: Day of the year
- `month`: Month
- `quarter`: Fiscal quarter
- `year`: Year

Extracting rows of a dataframe

Slicing to certain rows

Last week, you learned how to use the slice function from the dplyr package to limit a dataframe to certain rows by row position:

```
library("dplyr")
slice(.data = daily_show, 1:3)
```

Slicing to certain rows

For example, to print the first three rows of the `daily_show` data, you can run:

```
slice(daily_show, 1:3)
```

```
## # A tibble: 3 x 5
##   job      date       category guest_name show_day
##   <chr>    <date>     <chr>    <chr>      <ord>
## 1 actor    1999-01-11 Acting   Michael J~ Mon
## 2 comedian 1999-01-12 Comedy   Sandra Be~ Tue
## 3 televis~ 1999-01-13 Acting   Tracey Ul~ Wed
```

Randomly sampling rows

There are some other functions you can use to extract rows from a tibble dataframe, all from the "dplyr" package.

For example, if you'd like to extract a random subset of *n* rows, you can use the sample_n function, with the size argument set to *n*.

To extract two random rows from the daily_show dataframe, run:

```
sample_n(tbl = daily_show, size = 2)
```

```
## # A tibble: 2 x 5
##   job   date       category guest_name  show_day
##   <chr> <date>     <chr>    <chr>       <ord>
## 1 actor 2010-03-16 Acting   Jude Law    Tue
## 2 actor 2004-03-18 Acting   Ethan Hawke Thu
```

Extracting distinct rows

If you have a dataset where some observations are exactly the same, you can reduce the data to only unique observations using the distinct function.

For example, there seem to be two listings for the show that featured Hootie & the Blowfish:

```
select(slice(.data = daily_show, 64:65), date, guest_name)

## # A tibble: 2 x 2
##   date       guest_name
##   <date>     <chr>
## 1 1999-03-11 Hootie & the Blowfish, Billy Crystal
## 2 1999-03-11 Hootie & the Blowfish, Billy Crystal
```

Extracting distinct rows

If we run the distinct function on these rows, we can limit them to a single, unduplicated row:

```
hootie_shows <- select(slice(.data = daily_show, 64:65), date, g
distinct(.data = hootie_shows)

## # A tibble: 1 x 2
##   date       guest_name
##   <date>     <chr>
## 1 1999-03-11 Hootie & the Blowfish, Billy Crystal
```

Arranging rows

There is also a function, arrange, you can use to re-order the rows in a dataframe based on the values in one of its columns. The syntax for this function is:

```
# Generic code
arrange(dataframe, column_to_order_by)
```

If you run this function to use a character vector to order, it will order the rows alphabetically by the values in that column. If you specify a numeric vector, it will order the rows by the numeric value.

For example, we could reorder the daily_show data alphabetically by the
values in the category column with the following call:

```
daily_show <- arrange(daily_show, category)
head(daily_show, 3)

## # A tibble: 3 x 5
##    job     date       category guest_name   show_day
##    <chr>   <date>     <chr>    <chr>        <ord>
## 1 profes~ 2001-10-03 Academic Stephen S.~  Wed
## 2 profes~ 2001-12-03 Academic Nadine Str~  Mon
## 3 histor~ 2003-11-04 Academic Michael Be~  Tue
```

Arranging rows

If you want the ordering to be reversed (e.g., from "z" to "a" for character vectors, from higher to lower for numeric, latest to earliest for a Date), you can include the desc function.

For example, to reorder the daily_show data by descending date (latest to earliest), you can run:

```
daily_show <- arrange(daily_show, desc(date))
head(daily_show, 3)

## # A tibble: 3 x 5
##   job       date       category guest_name show_day
##   <chr>     <date>     <chr>    <chr>      <ord>
## 1 comedian  2015-08-05 Comedy   Louis C.K. Wed
## 2 actor     2015-08-04 Acting   Denis Lea~ Tue
## 3 stand-u~  2015-08-03 Comedy   Amy Schum~ Mon
```

Filtering to certain rows

Next, you might want to filter the dataset down so that it only includes
certain rows. You can use filter to do that. The syntax is:

```
## Generic code
filter(dataframe, logical expression)
```

The **logical expression** gives the condition that a row must meet to be
included in the output data frame. For example, you might want to pull:

- Rows from 2015
- Rows where the guest was an academic
- Rows where the job is not missing

Filtering to certain rows

For example, the == logical operator tests if two values are equal. So if
you want to create a data frame that only includes guests who were
scientists, you can run:

```
scientists <- filter(daily_show, category == "Science")
head(scientists)

## # A tibble: 6 x 5
##   job     date       category guest_name show_day
##   <chr>   <date>     <chr>    <chr>      <ord>
## 1 astrop~ 2015-04-23 Science  Neil deGra~ Thu
## 2 surgeon 2014-10-06 Science  Atul Gawan~ Mon
## 3 astrop~ 2013-09-04 Science  Mario Livio Wed
## 4 astrop~ 2013-03-06 Science  Neil deGra~ Wed
## 5 primat~ 2012-04-16 Science  Jane Gooda~ Mon
## 6 astrop~ 2012-02-27 Science  Neil deGra~ Mon
```

Common logical and relational operators in R

To build a logical statement to use in `filter`, you'll need to know some of R's logical and relational operators:

Operator	Meaning	Example
==	equals	`category == "Acting"`
!=	does not equal	`category != "Comedy`
%in%	is in	`category %in% c("Academic", "Science")`
is.na()	is NA	`is.na(job)`
&	and	`(year == 2015) & (category == "Academic")`
|	or	`(year == 2015) | (category == "Academic")`

Common logical and relational operators in R

The == operator will check each element of a vector against each corresponding element of another vector to see if it's equal.

The result will be a vector:

```r
c(1, 2, 3) == c(1, 2, 4)
```

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

Common logical and relational operators in R

By contrast, the `%in%` operator will check each element of a vector to see if it's a value that is included in a second vector.

In this case, the two vectors don't have to have the same length:

```
c(1, 2, 3) %in% c(1, 5)
```

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

This logical expressions is asking *Is the first element of the first vector, 1, in the set given by the second vector, 1 and 5? Is the second element of the first vector, 2, in the set given by the second vector? Etc.*

Common logical and relational operators in R

The **bang operator** (!) negates a logical expression:

```r
c(1, 2, 3) == c(1, 2, 5)
```

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

```r
!(c(1, 2, 3) == c(1, 2, 5))
```

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

```r
is.na(c(1, 2, NA))
```

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

```r
!is.na(c(1, 2, NA))
```

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

dplyr versus base R

Just so you know, all of these actions also have alternatives in base R:

dplyr	Base R equivalent
rename	Reassign colnames
select	Square bracket indexing ([,])
slice	Square bracket indexing ([,])
filter	subset
mutate	Replacement assignment expression

You will see these alternatives used in older code examples.

We'll take a break now and do section 2.6.5 of the In-Course Exercise.

Piping

Piping

If you look at the format of these dplyr functions, you'll notice that they all take a dataframe as their first argument:

```
# Generic code
rename(dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
select(dataframe, column_name_1, column_name_2)
slice(dataframe, starting_row:ending_row)
arrange(dataframe, column_to_order_by)
filter(dataframe, logical statement)
mutate(dataframe,
       changed_column = function(changed_column),
       new_column = function(other arguments))
```

Piping

Classically, you would clean up a dataframe in R by reassigning the
dataframe object at each step:

```r
daily_show <-read_csv("../data/daily_show_guests.csv",
                    skip = 4)
daily_show <- rename(daily_show,
                    job = GoogleKnowlege_Occupation,
                    date = Show,
                    category = Group,
                    guest_name = Raw_Guest_List)
daily_show <- select(daily_show, -YEAR)
daily_show <- mutate(daily_show, job = str_to_lower(job))
daily_show <- filter(daily_show, category == "Science")
```

Piping

"Piping" lets you clean this code up a bit. It can be used with any function that inputs a dataframe as its first argument. It "pipes" the dataframe created right before the pipe (%>%) into the function right after the pipe.

Piping

With piping, the same data cleaning looks like:

```r
daily_show <- read_csv("../data/daily_show_guests.csv",
                       skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")
```

Piping

Piping tip #1: As you are trying to figure out what "piped" code like this is doing, try highlighting from the start of the code through just part of the pipe and run that. For example, try highlighting and running just from read_csv through the before the %>% in the line with select, and see what that output looks like.

```r
daily_show <- read_csv("../data/daily_show_guests.csv",
                       skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")
```

Piping

Piping tip #2: When you are writing an R script that uses piping, first write it and make sure you have it right **without** assigning it to an R object (i.e., no <-). Often you'll use piping to clean up an object in R, but if you have to work on the piping code, you end up with different versions of the object, which will cause frustrations.

```r
read_csv("../data/daily_show_guests.csv", skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")
```

Piping

Piping tip #3: There is a keyboard shortcut for the pipe symbol:

```
Command-Shift-m
```

We'll take a break now and do section 2.6.6 of the In-Course Exercise.

Odds and ends

Being lazy

[Argument matching by position]

Being lazy

[Leaving quotation marks off package names]

[Non-standard evaluation inside tidyverse calls]

Infix operators

[In-fix operators are just functions in disguise]