Entering / cleaning data 1

Some odds and ends

Missing values

In R, `NA` is used to represent a missing value in a vector. This value can show up in numerical or character vectors (or in vectors of some other classes):

```r
c(1, 4, NA)
```

```
## [1]  1  4 NA
```

```r
c("Jane Doe", NA)
```

```
## [1] "Jane Doe" NA
```

The $ operator

We've talked about how you can combine vectors of the same length to create a dataframe.

To go the other direction (pull a column from a dataframe), you can use the $ operator.

For example, say you have the following dataset and want to pull the `color` column as a vector:

```
example_df <- data.frame(color = c("red", "blue"),
                         value = c(1, 2))
example_df
```

```
##   color value
## 1   red     1
## 2  blue     2
```

The $ operator

You can pull the `color` column as a vector using the name of the dataframe, the dollar sign, and then the name of the column:

```
example_df$color
```

```
## [1] red  blue
## Levels: blue red
```

```
class(example_df$color)
```

```
## [1] "factor"
```

(Note: You can use tab completion in RStudio after you put in `example_df$`.)

Order of evaluation

In R, you can “nest” functions within a single call. Just like with math, the order that the functions are evaluated moves from the inner set of parentheses to the outer one.

For example, to print the structure of the dataframe from the previous example after creating it, you can run:

```r
str(data.frame(color = c("red", "blue"), value = c(1, 2)))
```

```
## 'data.frame':    2 obs. of  2 variables:
##  $ color: Factor w/ 2 levels "blue","red": 2 1
##  $ value: num  1 2
```

paste and paste0

If you want to paste together several character strings to make a length-one character vector, you can use the paste function to do that:

```
paste("abra", "ca", "dabra")
```

```
## [1] "abra ca dabra"
```

By default, spaces are used to separate each original character string in the final string.

paste and paste0

If you want to remove these spaces, you can use the sep argument in the paste function:

```r
paste("abra", "ca", "dabra", sep = "")
```

```
## [1] "abracadabra"
```

A short-cut function is paste0, which is identical to running paste with the argument sep = "":

```r
paste0("abra", "ca", "dabra")
```

```
## [1] "abracadabra"
```

Getting data into R

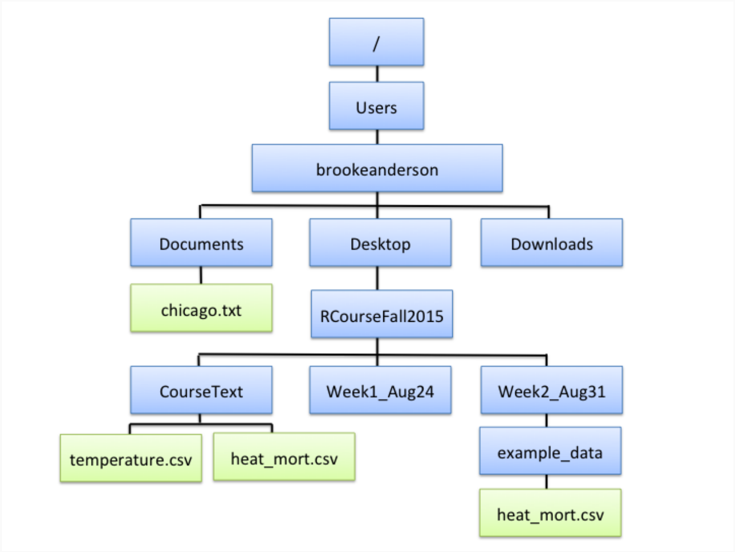
Basics of getting data into R

Basic approach:

- Download data to your computer
- Make sure R is working in the directory with your data (`getwd`, `setwd`)
- 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`)

Directories

Computer directory structure



Directories

You can check your working directory anytime using `getwd()`:

```r
getwd()
```

```
## [1] "/Users/_gbanders/r_course/RProgrammingForResearch/slides"
```

Directories

You can use `setwd()` to change your directory.

To get to your home directory (for example, mine is "/Users/brookeanderson"), you can use the abbreviation `~`.

For example, if you want to change into your home directory and print its name, you could run:

```
setwd("~")
getwd()
```

```
## [1] "/Users/brookeanderson"
```

Directories

Remember that, since ~ is a shortcut for my home directory, the following two calls would give me the same result:

```
setwd("~")
getwd()
```

```
## [1] "/Users/brookeanderson"
```

```
setwd("/Users/brookeanderson")
getwd()
```

```
## [1] "/Users/brookeanderson"
```

Directories

The most straightforward way to read in data is often to put it in your working directory and then read it in using the file name. If you're working in the directory with the file you want, you should see the file if you list files in the working directory:

```
list.files()
```

```
##  [1] "CourseNotes_Week1.pdf"
##  [2] "CourseNotes_Week1.Rmd"
##  [3] "CourseNotes_Week10.pdf"
##  [4] "CourseNotes_Week10.Rmd"
##  [5] "CourseNotes_Week11.pdf"
##  [6] "CourseNotes_Week11.Rmd"
##  [7] "CourseNotes_Week12.pdf"
##  [8] "CourseNotes_Week12.Rmd"
##  [9] "CourseNotes_Week13.pdf"
## [10] "CourseNotes_Week13.Rmd"
## [11] "CourseNotes_Week14.pdf"
```

Directories

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()`.

Getting around directories

There are a few abbreviations you can use to represent certain relative or absolute locations when you're using `setwd()`:

Shorthand	Meaning
`~`	Home directory
`.`	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

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:

```r
## 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
my_dir <- paste0("~/RProgrammingForResearch",
                 "data/measles_data")
my_dir
```

```
## [1] "~/RProgrammingForResearchdata/measles_data"
```

```r
setwd(my_dir)
```

Relative versus absolute pathnames

When you want to reference a directory or file, you can use one of two types of pathnames:

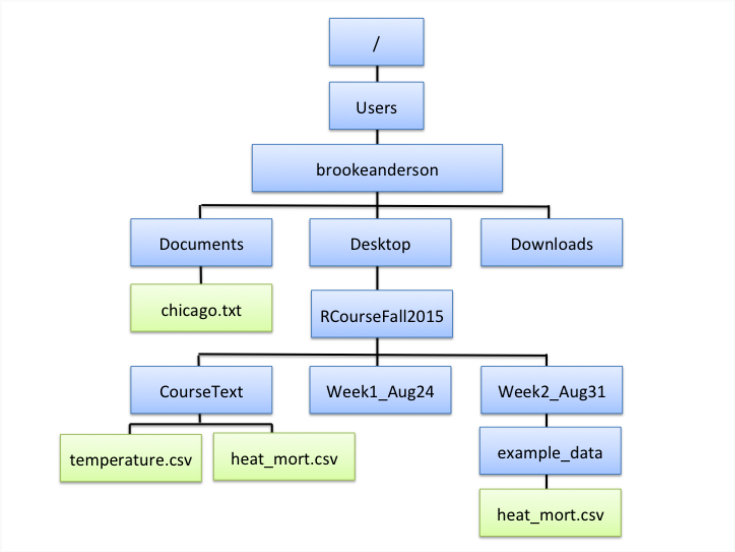
- *Relative pathname*: How to get to the file or directory from your current working directory
- *Absolute pathname*: How to get to the file or directory from anywhere on the computer

Relative versus absolute pathnames

If directions worked like pathnames, here is how you would tell someone to get to this building:

- *Relative pathname*: Turn right on Center, then turn right when you get to the intersection, then go to the second building on your left (only works if you started on Prospect going west right before the intersection with Center).
- *Absolute pathname*: Go to 350 W. Lake Street, Fort Collins, CO, USA.

Computer directory structure



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 using the two types of pathnames:

Absolute:

```r
setwd("/Users/brookeanderson/RProgrammingForResearch/data")
```

Relative:

```r
setwd("data")
```

Relative versus absolute pathnames

Here are some other examples of relative pathnames:

If data is a subdirectory of your current parent directory:

```
setwd("../data")
```

If data is a subdirectory of your home directory:

```
setwd("~/data")
```

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

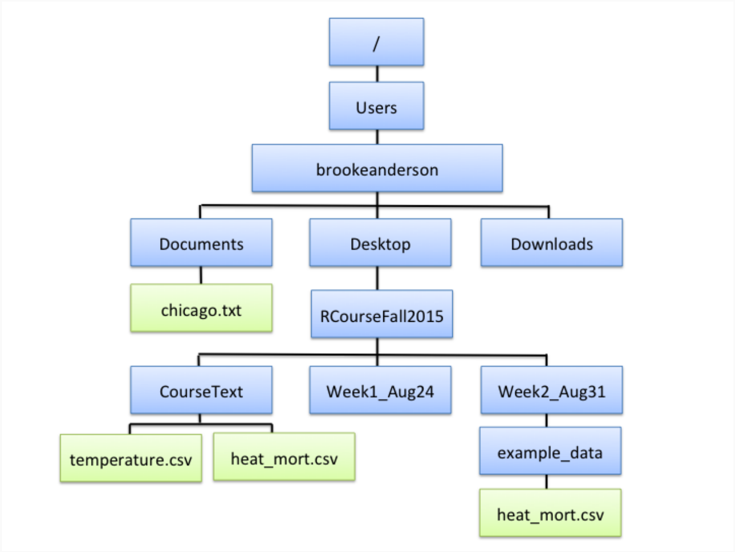
```
setwd("Ex/data")
```

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).

Computer directory structure



Relative versus absolute pathnames

If you want to read in data from a file that is not in your working directory, there are three options:

- Move the data into your working directory (this can be done outside of R).
- Change your working directory so that you are working in the directory that has the data (e.g., with `setwd()`).
- Use a pathname, rather than a simple filename, to refer to the data file (this will be the recommended method for most of this course, although we'll practice other methods today).

In-course exercise

We'll take a break now to do the first part of the in-course exercise (Sections 2.6.1 and 2.6.2).

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 (e.g., the table near the end of this page)
- Data in a database (e.g., SQL)
- Data stored in XML and JSON
- Really crazy data formats used in other disciplines (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`

Types of flat files

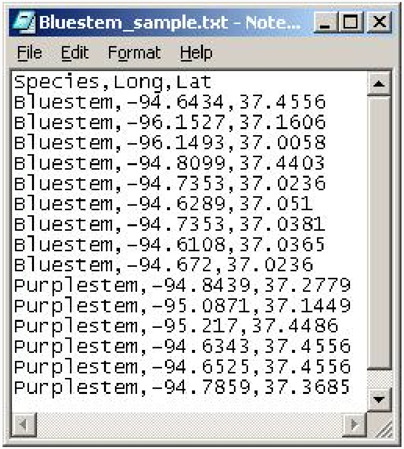
R can read in data from *a lot* of different formats. The only catch: you need to tell R how to do it.

To start, we'll look at flat files:

1. Fixed width files
2. Delimited files
    - ".csv": Comma-separated values
    - ".tab", ".tsv": Tab-separated values
    - Other possible delimiters: colon, semicolon, pipe ("|")

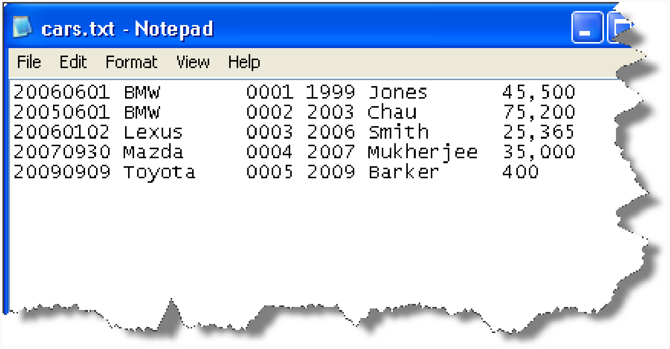
See if you can identify what types of files the following files are. . .

What type of file?



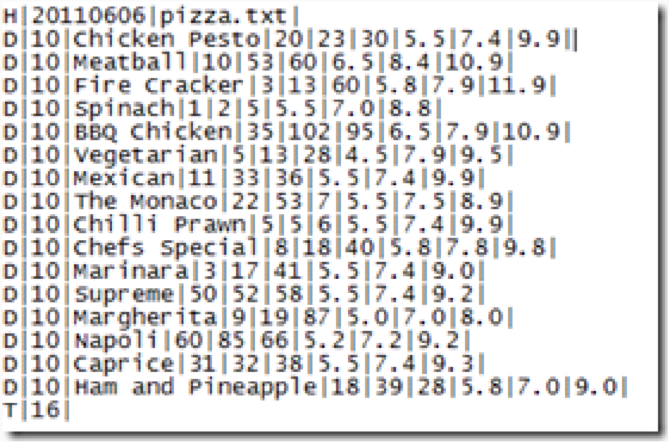
```
Species,Long,Lat
Bluestem,-94.6434,37.4556
Bluestem,-96.1527,37.1606
Bluestem,-96.1493,37.0058
Bluestem,-94.8099,37.4403
Bluestem,-94.7353,37.0236
Bluestem,-94.6289,37.051
Bluestem,-94.7353,37.0381
Bluestem,-94.6108,37.0365
Bluestem,-94.672,37.0236
Purplestem,-94.8439,37.2779
Purplestem,-95.0871,37.1449
Purplestem,-95.217,37.4486
Purplestem,-94.6343,37.4556
Purplestem,-94.6525,37.4556
Purplestem,-94.7859,37.3685
```

What type of file?



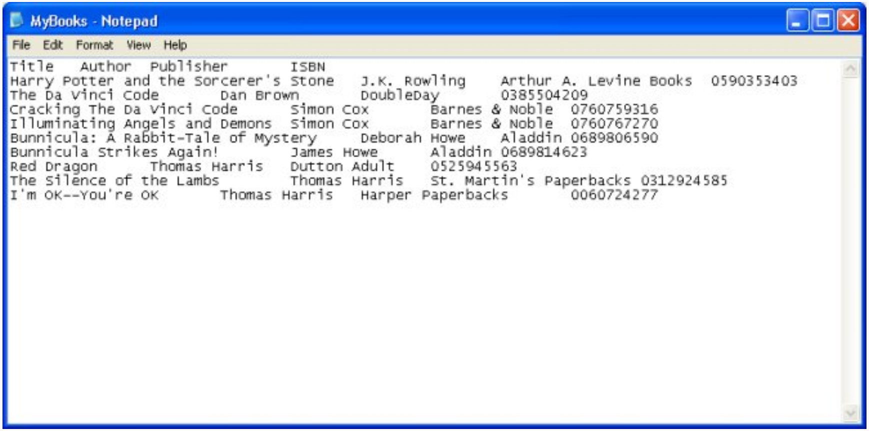
```
20060601 BMW         0001 1999 Jones        45,500
20050601 BMW         0002 2003 Chau         75,200
20060102 Lexus       0003 2006 Smith        25,365
20070930 Mazda       0004 2007 Mukherjee    35,000
20090909 Toyota      0005 2009 Barker       400
```

What type of file?



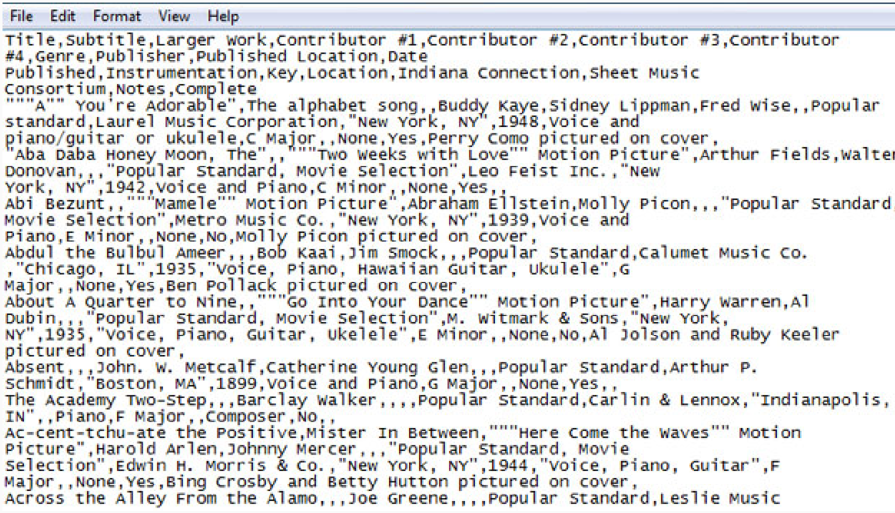
```
H|20110606|pizza.txt|
D|10|Chicken Pesto|20|23|30|5.5|7.4|9.9||
D|10|Meatball|10|53|60|6.5|8.4|10.9|
D|10|Fire Cracker|3|13|60|5.8|7.9|11.9|
D|10|Spinach|1|2|5|5.5|7.0|8.8|
D|10|BBQ Chicken|35|102|95|6.5|7.9|10.9|
D|10|Vegetarian|5|13|28|4.5|7.9|9.5|
D|10|Mexican|11|33|36|5.5|7.4|9.9|
D|10|The Monaco|22|53|7|5.5|7.5|8.9|
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?



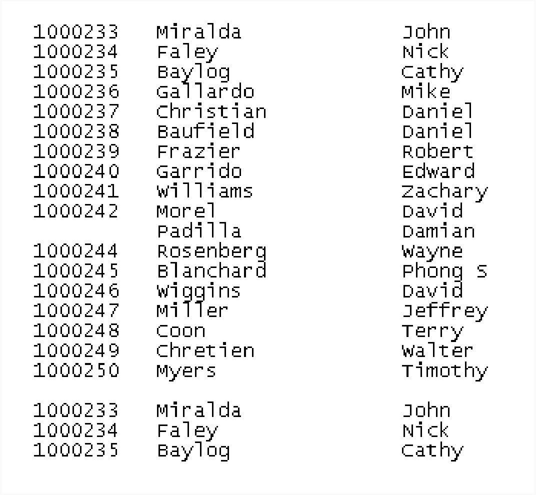
```
Title	Author	Publisher	ISBN
Harry Potter and the Sorcerer's Stone	J.K. Rowling	Arthur A. Levine Books	0590353403
The Da Vinci Code	Dan Brown	DoubleDay	0385504209
Cracking The Da Vinci Code	Simon Cox	Barnes & Noble	0760759316
Illuminating Angels and Demons	Simon Cox	Barnes & Noble	0760767270
Bunnicula: 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,Walte
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

To figure out the structure of a flat file, start by opening it in a text editor.

As an alternative, you can open it in RStudio (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 (e.g., `read_table`, `read_fwf`). Find out more about those functions with:

```r
library(readr)
?read_table
?read_fwf
```

`read.table` family of functions

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

Option	Description
`delim`	What is the delimiter in the data?
`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?

`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 separator (`delim`).

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

Function	Separator
`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).

`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.

readr family of functions

You will also see code that uses functions like read.csv and read.table to read in flat files. These are from the read.table family of functions, which are part of base R. The readr functions are very similar, but have some more sensible defaults, including in determining column classes.

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

```
## Parsed with column specification:
## cols(
##   YEAR = col_integer(),
##   GoogleKnowlege_Occupation = col_character(),
##   Show = col_character(),
##   Group = col_character(),
##   Raw_Guest_List = col_character()
## )
```

`read_*` family of functions

Compared to the `read.table` family of functions, the `read_*` functions:

- Work better with large datasets: faster, includes progress bar
- Have more sensible defaults (e.g., characters default to characters, not factors)

The "tidyverse"

The `readr` package is part of the "tidyverse"– a collection of recent and developing packages for R, many written by Hadley Wickham.

The “tidyverse”

"A giant among data nerds"

https://priceonomics.com/hadley-wickham-the-man-who-revolutionized-r/

Reading in online flat files

If you're reading in data from a non-secure webpage (i.e., one that starts with `http`), if the data is in a "flat-file" format, you can just read it in using the web address as the file name:

```
url <- paste0("http://www2.unil.ch/comparativegenometrics",
              "/docs/NC_006368.txt")
ld_genetics <- read_tsv(url)
ld_genetics[1:5, 1:4]
```

```
## # A tibble: 5 x 4
##     pos    nA    nC    nG
##   <int> <int> <int> <int>
## 1   500   307   153   192
## 2  1500   310   169   207
## 3  2500   319   167   177
## 4  3500   373   164   168
## 5  4500   330   175   224
```

Reading in online flat files

With the `readr` family of functions, you can also read in data from a secure webpage (e.g., one that starts with `https`). This allows you to read in data from places like GitHub and Dropbox public folders:

```
url <- paste0("https://raw.githubusercontent.com/cmrivers/",
              "ebola/master/country_timeseries.csv")
ebola <- read_csv(url)
ebola[1, 1:3]
```

```
## # A tibble: 1 x 3
##       Date   Day Cases_Guinea
##      <chr> <int>        <int>
## 1 1/5/2015   289         2776
```

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`

Saving / loading R objects

You can save an R object you've created as an `.RData` file using `save()`:

```r
save(ebola, file = "Ebola.RData")
list.files(pattern = "E")
```

```
## [1] "Ebola.RData"
```

This saves to your current working directory (unless you specify a different location).

Saving / loading R objects

Then you can re-load the object later using `load()`:

```r
rm(ebola)
ls()
```

```
## [1] "daily_show"        "dirpath_shortcuts"
## [3] "example_df"        "ld_genetics"
## [5] "my_dir"            "read_funcs"
## [7] "url"
```

```r
load("Ebola.RData")
ls()
```

```
## [1] "daily_show"        "dirpath_shortcuts"
## [3] "ebola"             "example_df"
## [5] "ld_genetics"       "my_dir"
## [7] "read_funcs"        "url"
```

Saving R objects

One caveat for saving R objects: some people suggest you avoid this if possible, to make your research more reproducible.

Imagine someone wants to look at your data and code in 30 years. R might not work the same, so you might not be able to read an `.RData` file. However, you can open flat files (e.g., `.csv`, `.txt`) and R scripts (`.R`) in text editors– you should still be able to do this regardless of what happens to R.

Potential exceptions:

- You have an object that you need to save that has a structure that won't work well in a flat file
- Your starting dataset is really, really large, and it would take a long time for you to read in your data fresh every time

In-course exercise

We'll take another break here to work on the next part of the In-Course Exercise (Section 2.6.3).

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.

The “tidyverse”

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

```r
library(dplyr)
library(lubridate)
```

Cleaning data

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

```
daily_show <- read_csv("../data/daily_show_guests.csv",
                       skip = 4)
head(daily_show, 3)
```

```
## # A tibble: 3 x 5
##    YEAR GoogleKnowlege_Occupation    Show  Group
##   <int>                     <chr>   <chr>  <chr>
## 1  1999                     actor 1/11/99 Acting
## 2  1999                  Comedian 1/12/99 Comedy
## 3  1999        television actress 1/13/99 Acting
## # ... with 1 more variables: 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
- includes spaces
- includes upper case

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

```r
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:

```
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
##   <int>              <chr>   <chr>    <chr>
## 1  1999              actor 1/11/99   Acting
## 2  1999           Comedian 1/12/99   Comedy
## 3  1999 television actress 1/13/99   Acting
## # ... with 1 more variables: guest_name <chr>
```

Renaming columns

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

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

```
# 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 `select` for this. 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
##                   <chr>   <chr>    <chr>
##  1                actor 1/11/99   Acting
##  2             Comedian 1/12/99   Comedy
##  3 television actress 1/13/99   Acting
##  4        film actress 1/14/99   Acting
##  5                actor 1/18/99   Acting
##  6                actor 1/19/99   Acting
##  7     Singer-lyricist 1/20/99 Musician
##  8                model 1/21/99    Media
##  9                actor 1/25/99   Acting
## 10  stand-up comedian 1/26/99   Comedy
```

Selecting columns

As a reminder, we could have selected these columns using square bracket indexing, too:

```
daily_show[ , 2:5]
```

However, the `select` function will fit in nicely with other data-cleaning functions from "tidyverse" packages, plus 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:

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

```
## # A tibble: 3 x 4
##                   job    date category
##                 <chr>   <chr>    <chr>
## 1               actor 1/11/99   Acting
## 2            Comedian 1/12/99   Comedy
## 3 television actress 1/13/99   Acting
## # ... with 1 more variables: guest_name <chr>
```

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:

```r
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 name 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:

```r
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

We could use the `tolower` function 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
##                    <chr>   <chr>    <chr>
## 1                  actor 1/11/99   Acting
## 2               comedian 1/12/99   Comedy
## 3 television actress 1/13/99   Acting
## # ... with 1 more variables: guest_name <chr>
```

In-course exercise

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

Dates in R

`Date` class

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

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 from `dplyr` functions.

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 `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

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.

The functions in `lubridate` are pretty good at working with these different options intelligently:

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

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

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

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

lubridate package

Some more examples:

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

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

```r
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
##                   <chr>     <date>    <chr>
## 1                 actor 1999-01-11   Acting
## 2              comedian 1999-01-12   Comedy
## 3 television actress 1999-01-13   Acting
## # ... with 1 more variables: guest_name <chr>
```

```
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 = T
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     Tues
## 3 1999-01-13      Wed
```

`lubridate` package

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

Filtering and logical operators

Slicing to certain rows

Last week, you learned how to use square bracket indexing to limit a dataframe to certain rows by row number:

```
daily_show[1:3, ]
```

The `dplyr` package has a function you can use to do this, called `slice`. That function has the syntax:

```
# Generic code
slice(dataframe, starting_row:ending_row)
```

where `starting_row` is the row number of the first row you want to keep and `ending_row` is the row number of the last line you want to keep.

Slicing to certain rows

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

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

```
## # A tibble: 3 x 5
##                   job       date category
##                 <chr>     <date>    <chr>
## 1               actor 1999-01-11   Acting
## 2            comedian 1999-01-12   Comedy
## 3 television actress 1999-01-13   Acting
## # ... with 2 more variables: guest_name <chr>,
## #   show_day <ord>
```

Arranging rows

There is also a function, `arrange`, you can use to re-order the rows in a dataframe. The syntax for this function is:

```r
# 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.

Arranging rows

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
##       <chr>     <date>    <chr>             <chr>
## 1 professor 2001-10-03 Academic  Stephen S. Morse
## 2 professor 2001-12-03 Academic   Nadine Strossen
## 3 historian 2003-11-04 Academic Michael Beschloss
## # ... with 1 more variables: show_day <ord>
```

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
##                <chr>     <date>    <chr>
## 1          comedian 2015-08-05   Comedy
## 2             actor 2015-08-04   Acting
## 3 stand-up comedian 2015-08-03   Comedy
## # ... with 2 more variables: guest_name <chr>,
## #   show_day <ord>
```

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 statement)
```

The `logical statement` 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, 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
##             <chr>     <date>    <chr>
## 1 astrophysicist 2015-04-23  Science
## 2        surgeon 2014-10-06  Science
## 3 astrophysicist 2013-09-04  Science
## 4 astrophysicist 2013-03-06  Science
## 5  primatologist 2012-04-16  Science
## 6 astrophysicist 2012-02-27  Science
## # ... with 2 more variables: guest_name <chr>,
## #   show_day <ord>
```

Common logical operators in R

To build a logical statement to use in `filter`, you'll need to know some of R's logical 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)`
`!is.na()`	is not NA	`!is.na(job)`
`&`	and	`year == 2015 & category == "Academic"`
`|`	or	`year == 2015 | category == "Academic"`

`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`	Use $ to change / create columns

You will see these alternatives used in older code examples.

In-course exercise

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

Piping

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:

```
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`

In-course exercise

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