

## 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):

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

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

```
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:

```
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
```

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:

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

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

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

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

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



# Getting data into R

---

# Basics of getting data into R

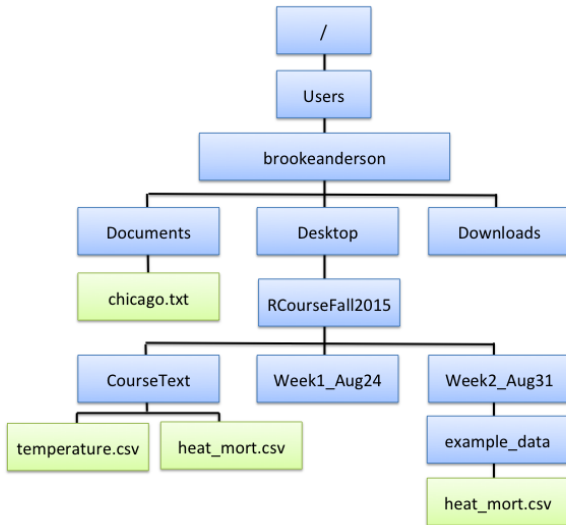
Basic approach:

- 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



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

```
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"
```



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
<code>~</code>	Home directory
<code>.</code>	Current working directory
<code>..</code>	One directory up from current working directory (parent directory)
<code>../..</code>	Two directories up from current working directory
<code>../data</code>	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:

```
## 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:

```
my_dir <- paste0("~/RProgrammingForResearch",  
                "data/measles_data")  
  
my_dir  
  
## [1] "~/RProgrammingForResearchdata/measles_data"  
  
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:

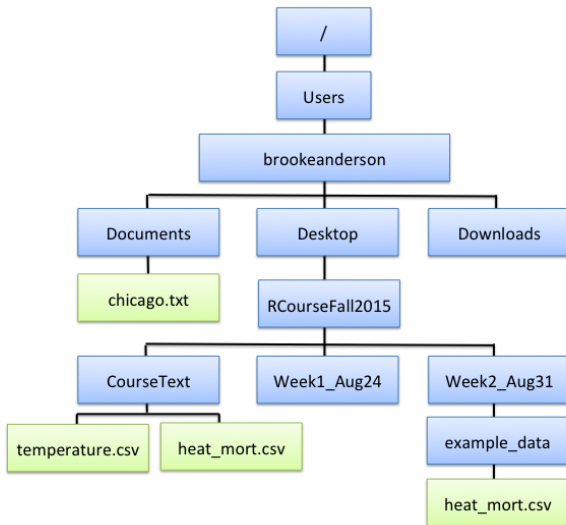
- *Relative pathname*: How to get to the file or directory from your current working directory
- *Absolute pathname*: How to get to the file or directory from anywhere on the computer

## 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:

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

Relative:

```
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:

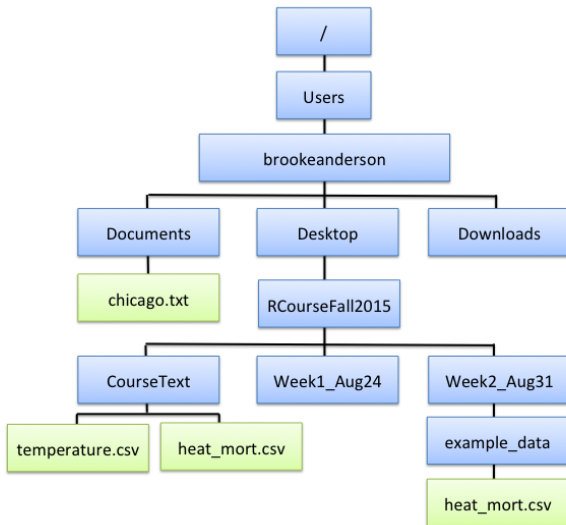
```
setwd("Ex/data")
```

## 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).

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)
- 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)
- 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.

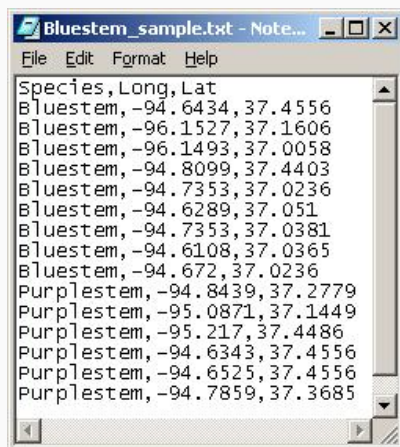
To start, we'll look at flat files:

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. . .



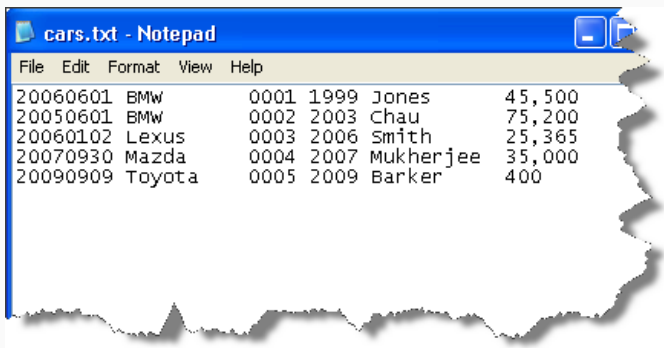
## What type of file?



A screenshot of a Notepad window titled "Bluestem\_sample.txt - Note...". The window has a menu bar with "File", "Edit", "Format", and "Help". The text inside the window is as follows:

```
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
```

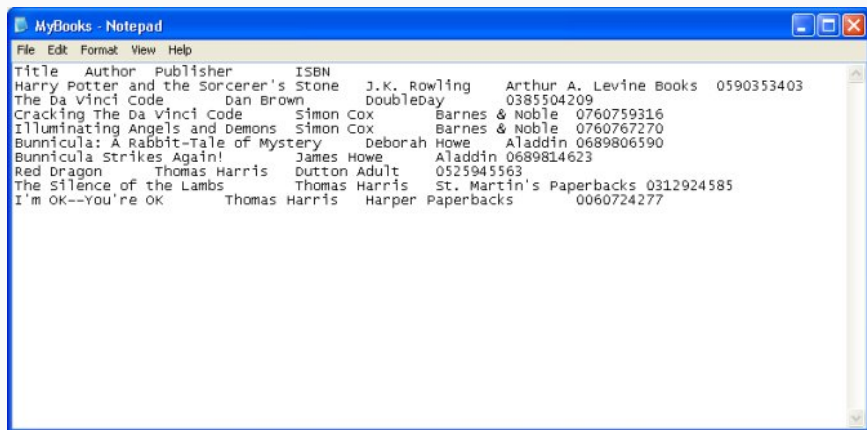
# What type of file?



## 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?



# What type of file?

File Edit Format View Help

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 Bezant,, ""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

## Reading in flat files

R can read any of these types of files using one of the `read.table` and `read.fwf` functions. Find out more about those functions with:

```
?read.table
```

```
?read.fwf
```

## read.table family of functions

Some of the interesting options with the `read.table` family of functions are:

Option	Description
<code>sep</code>	What is the delimiter in the data?
<code>skip</code>	How many lines of the start of the file should you skip?
<code>header</code>	Does the first line you read give column names?
<code>as.is</code>	Should you bring in strings as characters, not factors?
<code>nrows</code>	How many rows do you want to read in?
<code>na.strings</code>	How are missing values coded?



## read.table family of functions

All members of the `read.table` family are doing the same basic thing. The only difference is what defaults they have for the separator (`sep`) and the decimal point (`dec`).

Members of the `read.table` family:

Function	Separator	Decimal point
<code>read.csv</code>	comma	period
<code>read.csv2</code>	semi-colon	comma
<code>read.delim</code>	tab	period
<code>read.delim2</code>	tab	comma

## read\_\* family of functions

The `read.table` family of functions are part of base R. There is a newer package called `readr` that has a family of `read_*` functions. These functions are very similar, but have some more sensible defaults.

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

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

## read\_\* family of functions

Functions in the `read_*` family include

- `read_csv`, `read_tsv` (specific delimiters)
- `read_delim`, `read_table` (generic)
- `read_fwf`
- `read_log`
- `read_lines`

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 new 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 `read_*` 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/cmriivers/",  
             "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	<code>read_excel</code>	<code>readxl</code>
SAS	<code>read_sas</code>	<code>haven</code>
SPSS	<code>read_spss</code>	<code>haven</code>
Stata	<code>'read_stata</code>	<code>haven</code>



## Saving / loading R objects

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

```
save(ebola, file = "Ebola.RData")  
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"  
## [12] "CourseNotes_Week14.Rmd"  
## [13] "CourseNotes_Week15.pdf"
```

## Saving / loading R objects

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

```
rm(ebola)
```

```
ls()
```

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

```
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

# Data cleaning

---

Common data-cleaning tasks include:

Task	dplyr function
Renaming columns	<code>rename</code>
Filtering to certain rows	<code>filter</code>
Selecting certain columns	<code>select</code>
Adding or changing columns	<code>mutate</code>

# 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>
```

In cleaning up the data, we'll use dplyr functions, so we need to load that package:

```
library(dplyr)
```

## 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:

```
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 = GoogleKnowledge_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>
```

## 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, ...)
```

## 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

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
##           <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>
```

## 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 4  
##           job      date category  
##       <chr>   <chr>   <chr>  
## 1 neurosurgeon 4/28/03  Science  
## 2 scientist    1/13/04  Science  
## 3 physician    6/15/04  Science  
## 4 doctor       9/6/05   Science  
## 5 astronaut    2/13/06  Science  
## 6 Astrophysicist 1/30/07  Science  
## # ... with 1 more variables: guest_name <chr>
```

## Common logical operators in R

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

Operator	Meaning	Example
<code>==</code>	equals	<code>category == "Acting"</code>
<code>!=</code>	does not equal	<code>category != "Comedy"</code>
<code>%in%</code>	is in	<code>category %in% c("Academic", "Science")</code>
<code>is.na()</code>	is NA	<code>is.na(job)</code>
<code>!is.na()</code>	is not NA	<code>!is.na(job)</code>
<code>&amp;</code>	and	<code>year == 2015 &amp; category == "Academic"</code>
<code> </code>	or	<code>year == 2015   category == "Academic"</code>

# 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))
```



## 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

We could use the `tolower` function to make all listings lowercase:

```
mutate(daily_show, job = tolower(job))
```

```
## # 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
```

```
## # ... with 2,683 more rows and 1 more variables:
```

# Piping



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

```
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)  
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:

```
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 = tolower(job))  
daily_show <- filter(daily_show, category == "Science")
```

“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 = GoogleKnowledge_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = tolower(job)) %>%
  filter(category == "Science")
```

```
## Parsed with column specification:
```

```
## cols(
##   YEAR = col_integer(),
##   GoogleKnowlege_Occupation = col_character(),
##   Show = col_character(),
##   Group = col_character()
```

## 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
filter	subset
mutate	Use \$ to change / create columns

You will see these alternatives used in older code examples.



## Dates in R

---

One final common task in cleaning data is to change the class of some of the columns. This is especially common for dates, which will usually be read in as characters or factors.

# 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.

## Converting to Date class

To convert a vector to the Date class, you can use `as.Date()`:

```
daily_show <- mutate(daily_show,  
                      date = as.Date(date, format = "%m/%d/%y"))  
head(daily_show, 3)
```

```
## # A tibble: 3 x 4  
##           job           date category  
##           <chr>       <date>    <chr>  
## 1 neurosurgeon 2003-04-28 Science  
## 2   scientist 2004-01-13 Science  
## 3   physician 2004-06-15 Science  
## # ... 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:

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

```
## [1] "2003-04-28" "2015-04-23"
```

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

```
## Time difference of 4378 days
```

## Converting to Date class

The only tricky thing is learning the abbreviations for the `format` option. You use this option to specify the format of the date **before** you change it to a Date class. Here are some common date format abbreviations:

Abbreviation	Meaning
%m	Month as a number (e.g., 1, 05)
%B	Full month name (e.g., August)
%b	Abbreviated month name (e.g., Aug)
%y	Two-digit year (e.g., 99)
%Y	Four-digit year (e.g., 1999)

## Converting to Date class

Here are some examples:

Current format of date	format =
10/23/2008	"%m/%d%Y"
08-10-23	"%y-%m-%d"
Oct. 23 2008	"%b. %d %Y"
October 23, 2008	"%B %d, %Y"



## lubridate package

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

The ymd function from lubridate can be used regardless of the format, as long as the date elements are in the order: year, month, day. For example:

```
library(lubridate)
ymd("2008-10-13")
```

```
## [1] "2008-10-13"
```

```
ymd("'08 Oct 13")
```

```
## [1] "2008-10-13"
```

```
ymd("'08 Oct 13")
```

```
## [1] "2008-10-13"
```

The lubridate package has similar functions for other date orders or for date-times. For example:

- `dmy`
- `mdy`
- `ymd_h`
- `ymd_hm`

## lubridate package

We could have used these to transform the date in `daily_show`:

```
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(date = mdy(date)) %>%  
  filter(category == "Science")  
head(daily_show, 2)
```

```
## # A tibble: 2 x 4
```

```
##           job           date category  
##      <chr>      <date>      <chr>  
## 1 neurosurgeon 2003-04-28  Science  
## 2   scientist 2004-01-13  Science
```

## 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:

```
mutate(daily_show,  
       show_day = wday(date, label = TRUE)) %>%  
  select(date, show_day, guest_name) %>%  
  slice(1:5)
```

```
## # A tibble: 5 x 3  
##       date show_day      guest_name  
##   <date>   <ord>         <chr>  
## 1 2003-04-28 Mon      Dr Sanjay Gupta  
## 2 2004-01-13 Tues      Catherine Weitz  
## 3 2004-06-15 Tues      Hassan Ibrahim  
## 4 2005-09-06 Tues      Dr. Marc Siegel  
## 5 2006-02-13 Mon Astronaut Mike Mullane
```

Functions in `lubridate` for pulling elements from a date include:

- `wday`
- `mday`
- `yday`
- `month`
- `quarter`
- `year`