

40 Days and 40 Nights

Martin Morgan¹ L. Shawn Matott²

2020-04-28

¹Roswell Park Comprehensive Cancer Center, Martin.Morgan@RoswellPark.org
²Roswell Park Comprehensive Cancer Center

Contents

Motivation	7
Introduction	7
What to expect	7
1 Basics	11
1.1 Day 1 (Monday) Zoom orientation	11
1.2 Day 2: Vectors and variables	21
1.3 Day 3: <code>factor()</code> , <code>Date()</code> , and <code>NA</code>	23
1.4 Day 4: Working with variables	27
1.5 Day 5 (Friday) Zoom check-in	30
1.6 Day 6: <i>R</i> scripts	39
1.7 Day 7: Saving data	41
2 The data frame	45
2.1 Day 8 (Monday) Zoom check-in	45
2.2 Day 9: Creation and manipulation	54
2.3 Day 10: <code>subset()</code> , <code>with()</code> , and <code>within()</code>	61
2.4 Day 11: <code>aggregate()</code> and an initial work flow	64
2.5 Day 12 (Friday) Zoom check-in	69
2.6 Day 13: Basic visualization	82
2.7 Day 14: Functions	84

3 Packages and the ‘tidyverse’	91
3.1 Day 15 (Monday) Zoom check-in	91
3.2 Day 16 Key tidyverse packages: readr and dplyr	99
3.3 Day 17 Visualization with ggplot2	103
3.4 Day 18 Worldwide COVID data	120
3.5 Day 19 (Friday) Zoom check-in	127
3.6 Day 20 Exploring the course of pandemic in different regions . .	127
3.7 Day 21	133
4 Machine learning	135
4.1 Day 22 (Monday) Zoom check-in	135
4.2 Day 23	150
4.3 Day 24	150
4.4 Day 25	150
4.5 Day 26 (Friday) Zoom check-in	151
4.6 Day 27	151
4.7 Day 28	151
5 Bioinformatics with Bioconductor	153
5.1 Day 29 (Monday) Zoom check-in	153
5.2 Day 30	153
5.3 Day 31	153
5.4 Day 32	153
5.5 Day 33 (Friday) Zoom check-in	153
5.6 Day 34	153
5.7 Day 35	153
6 Collaboration	155
6.1 5 Days (Monday) Zoom check-in	155
6.2 4 Days	155
6.3 3 Days	155
6.4 2 Days	155
6.5 Today! (Friday) Zoom check-in	155

CONTENTS	5
-----------------	----------

Frequently asked questions	157
-----------------------------------	------------

Motivation

This is a WORK IN PROGRESS.

This course was suggested and enabled by Adam Kisailus and Richard Hershberger. It is available for Roswell Park graduate students.

Introduction

The word ‘quarantine’ is from the 1660’s and refers to the fourty days (Italian *quaranta giorni*) a ship suspected of carrying disease was kept in isolation.

What to do in a quarantine? The astronaut Scott Kelly spent nearly a year on the International Space Station. In a New York Times opinion piece he says, among other things, that ‘you need a hobby’, and what better hobby than a useful one? Let’s take the opportunity provided by COVID-19 to learn R for statistical analysis and comprehension of data. Who knows, it may be useful after all this is over!

What to expect

We’ll meet via zoom twice a week, Mondays and Fridays, for one hour. We’ll use this time to make sure everyone is making progress, and to introduce new or more difficult topics. Other days we’ll have short exercises and activities that hopefully provide an opportunity to learn at your own speed.

We haven’t thought this through much, but roughly we might cover:

- Week 1: We’ll start with the basics of installing and using R. We’ll set up *R* and *RStudio* on your local computer, or if that doesn’t work use a cloud-based RStudio. We’ll learn the basics of *R* – numeric, character, logical, and other vectors; variables; and slightly more complicated representations of ‘factors’ and dates. We’ll also use *RStudio* to write a

script that allows us to easily re-create an analysis, illustrating the power concept of *reproducible research*.

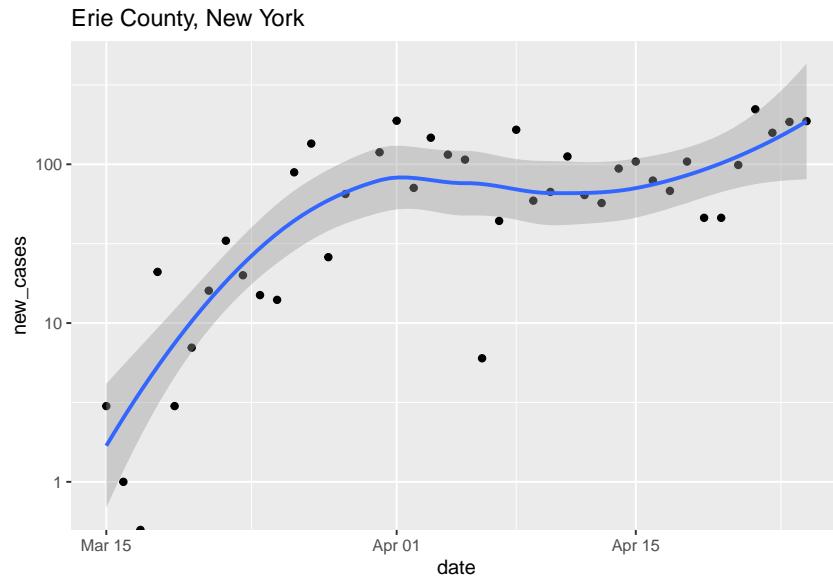
```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes_per_activity <- c(20, 30, 60, 60, 60)
minutes_per_activity >= 60
## [1] FALSE FALSE TRUE TRUE TRUE
activity[minutes_per_activity >= 60]
## [1] "conference call" "webinar"           "walk"
```

- Week 2: The `data.frame`. This week is all about *R*'s `data.frame`, a versatile way of representing and manipulating a table (like an Excel spreadsheet) of data. We'll learn how to create, write, and read a `data.frame`; how to go from data in a spreadsheet in Excel to a `data.frame` in *R*; and how to perform simple manipulations on a `data.frame`, like creating a subset of data, summarizing values in a column, and summarizing values in one column based on a grouping variable in another column.

```
url = "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
cases <- read.csv(url)
erie <- subset(cases, county == "Erie" & state == "New York")
tail(erie)
##           date county    state fips cases deaths
## 77398 2020-04-21   Erie New York 36029  2147   161
## 80188 2020-04-22   Erie New York 36029  2233   174
## 82989 2020-04-23   Erie New York 36029  2450   179
## 85793 2020-04-24   Erie New York 36029  2603   184
## 88609 2020-04-25   Erie New York 36029  2773   199
## 91427 2020-04-26   Erie New York 36029  2954   205
```

- Week 3: Packages for extending *R*. A great strength of *R* is its extensibility through packages. We'll learn about CRAN, and install and use the 'tidyverse' suite of packages. The tidyverse provides us with an alternative set of tools for working with tabular data, and We'll use publicly available data to explore the spread of COVID-19 in the US. We'll read, filter, mutate (change), and select subsets of the data, and group data by one column (e.g., 'state') to create summaries (e.g., cases per state). We'll also start to explore data visualization, creating our first plots of the spread of COVID-19.

```
library(dplyr)
library(ggplot2)
## ...additional commands
```



- Week 4: Machine learning. This week will develop basic machine learning models for exploring data.
- Week 5: Bioinformatic analysis with Bioconductor. *Bioconductor* is a collection of more than 1800 *R* packages for the statistical analysis and comprehension of high-throughput genomic data. We'll use *Bioconductor* to look at COVID-19 genome sequences, and to explore emerging genomic data relevant to the virus.
- Week 6: COVID-19 has really shown the value of open data and collaboration. In the final week of our quarantine, we'll explore collaboration; developing independent and group projects that synthesize the use of *R* to explore data. We'll learn tools of collaboration including git and github, and develop 'best practices' for robust, reproducible research. We'll learn about writing 'markdown' reports to share our project with others.

Chapter 1

Basics

1.1 Day 1 (Monday) Zoom orientation

1.1.1 Logistics (10 minutes)

Course material

- Available at <https://mtmorgan.github.io/QuaRantine>

Cadence

- Monday and Friday group zoom sessions – these will review and troubleshoot previous material, and outline goals for the next set of independent activities.
- Daily independent activities – most of your learning will happen here!

Communicating

- We'll use Microsoft Teams (if most participants have access to the course)
- Visit Microsoft Teams and sign in with your Roswell username (e.g., MA38727@RoswellPark.org) and the password you use to check email, etc. Join the 'QuaRantine' team.

1.1.2 Installing *R* and *RStudio* (25 minutes, Shawn)

What is R?

- A programming language for statistical computing, data analysis and scientific graphics.
- Open-source with a large (and growing) user community.
- Currently in the top 10 most popular languages according to the tiobe index.

What is RStudio?

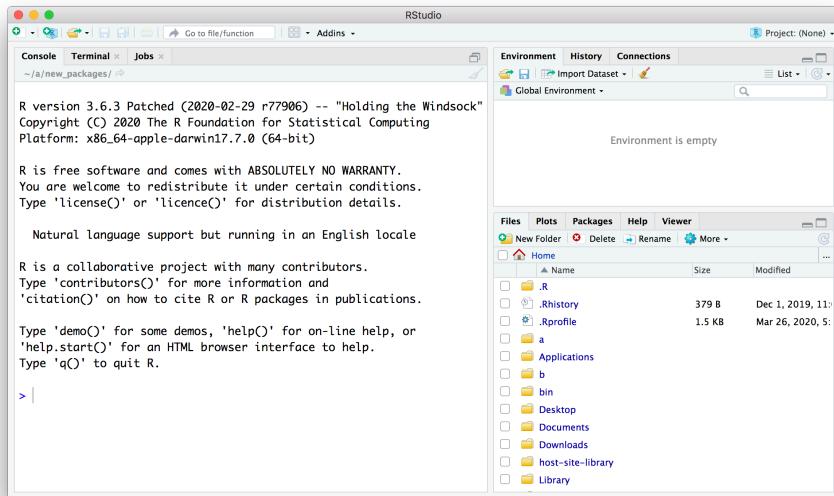
- RStudio provides an integrated editor and shell environment to make R programming easier. Some of the more useful features include:
 - Syntax highlighting and color coding
 - Easy switching between shell and editor
 - Dynamic help and docs

Installing *R* and *RStudio*

- Two ways to “get” RStudio:
 - Install on your laptop or desktop
 - * Download the free desktop installer here
 - Use the rstudio.cloud resource
 - * Visit rstudio.cloud, sign-up, and sign-on

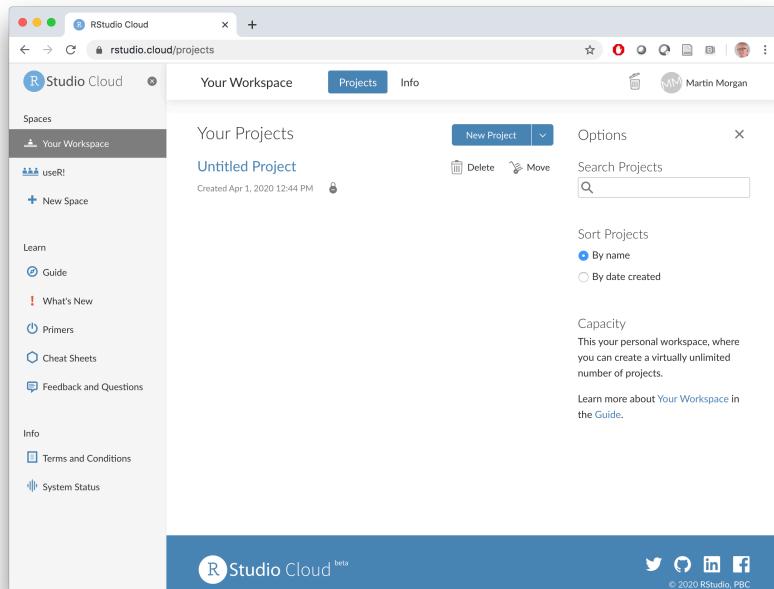
The preferred approach for this course is to try to install R and RStudio on your own computer

- Windows Users:
 - Download R for Windows and run the installer. Avoid, if possible, installing as administrator.
 - Download RStudio for Windows and run the installer.
 - Test the installation by launching RStudio. You should end up with a window like the screen shot below.
- Mac Users:
 - Download R for macOS (OS X 10.11, El Capitan, and later) or older macOS and run the installer.
 - Download RStudio for macOS and run the installer.
 - Test the installation by launching RStudio. You should end up with a window like the screen shot below.

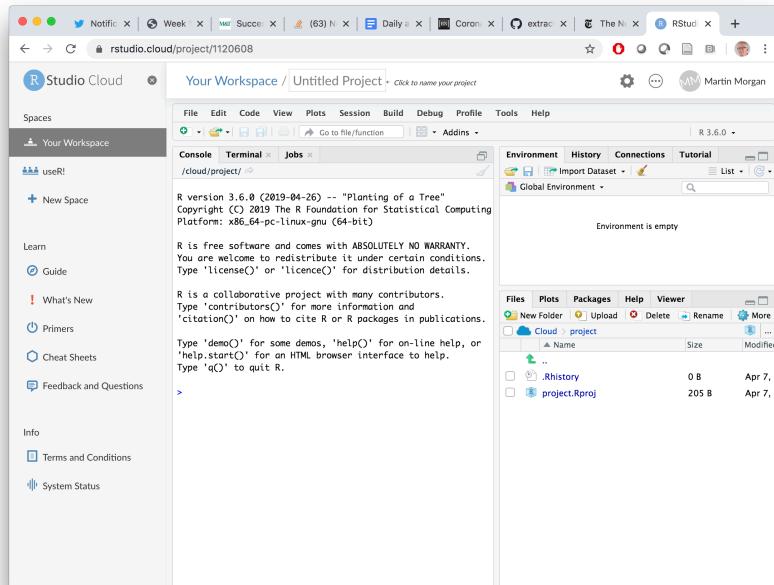


An ALTERNATIVE, if installing on your own computer does not work:

- Do the following only if you are NOT ABLE TO INSTALL R and RStudio.
- Visit rstudio.cloud. Click the ‘Get Started’ button, and create an account (I used my gmail account...). You should end up at a screen like the following.



- Click on the ‘New Project’ button, to end up with a screen like the one below. Note the ‘Untitled Project’ at the top of the screen; click on it to name your project, e.g., ‘QuaRantine’.





Breakout Room

At this point you should have RStudio running either via your desktop installation or through rstudio.cloud. If not, please let us know via the chat window and we'll invite you to a breakout room to troubleshoot your installation.

1.1.3 Basics of *R* (25 minutes)

R as a simple calculator

```
1 + 2  
## [1] 3
```

R Console Output

Enter this in the console:

```
2 + 3 * 5  
## [1] 17
```

Q: what's the [1] all about in the output?

A: It's the index of the first entry in each line.

This is maybe a better example:

```
1:30
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

Displaying help in the R Console

? <command-name>

- Some examples:

```
? cat
? print
```

Variables

Naming variables in *R*

- A variable name can contain letters, numbers, and the dot . or underline _ characters. Variables should start with a letter.
- Try entering these in the console:

```
y = 2
try.this = 33.3
oneMoreTime = "woohoo"

• Now try these:
2y = 2
_z = 33.3
function = "oops, my bad"
```

R is case sensitive (*R* != *r*)

```
R = 2
r = 3
R == r
## [1] FALSE
```

Variable Assignment

- You may use = or <- (and even ->) to assign values to a variable.

```
x <- 2 + 3 * 5
y = 2 + 3 * 6
2 + 3 * 7 -> z
cat(x, y, z)
## 17 20 23
```

R's four basic 'atomic' data types

- Numeric (includes integer, double, etc.)
 - 3.14, 1, 2600
- Character (string)
 - "hey, I'm a string"
 - 'single quotes are ok too'
- Logical
 - TRUE or FALSE (note all caps)
- NA
 - not assigned (no known value)

Use `class()` to query the class of data:

```
a <- 5
class(a)
## [1] "numeric"
```

Use `as.` to coerce a variable to a specific data type

```
a <- as.integer(5)
class(a)
## [1] "integer"
```

```
d <- as.logical(a)
d
## [1] TRUE
class(d)
## [1] "logical"
```

Using Logical Operators

Equivalence test (`==`):

```
1 == 2
## [1] FALSE
```

Not equal test (\neq):

```
1 != 2
## [1] TRUE
```

less-than ($<$) and greater-than ($>$):

```
18 > 44
## [1] FALSE
3 < 204
## [1] TRUE
```

Logical Or (\mid):

```
(1 == 2) | (2 == 2)
## [1] TRUE
```

Logical And ($\&$):

```
(1 == 2) & (2 == 2)
## [1] FALSE
```

Objects and Vectors in R

Objects

- R stores everything, variables included, in ‘objects’.

```
x <- 2.71

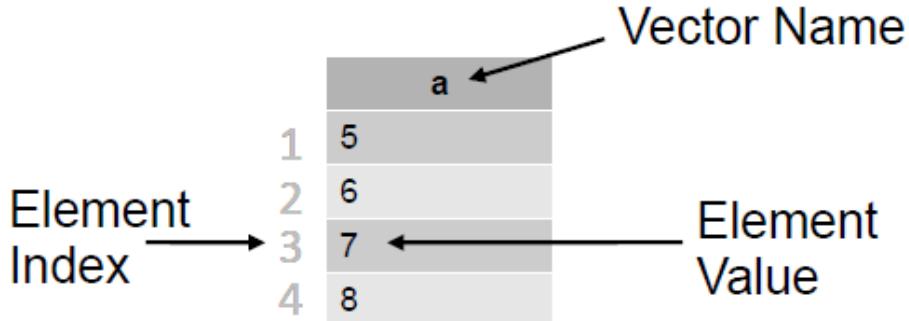
# print the value of an object
print(x)
## [1] 2.71

# determine class or internal type of an object
class(x)
## [1] "numeric"

# TRUE if an object has not been assigned a value
is.na(x)
## [1] FALSE
```

Vectors

- ‘Vectors’ and ‘data frames’ are the bread and butter of R
- Vectors consist of several elements of the same class
 - e.g. a vector of heart rates, one per patient



Data frames (`data.frame`)

- Data frames are structures that can contain columns of various types
 - e.g. height, weight, age, heart rate, etc.
 - Handy containers for experimental data
 - Analogous to spreadsheet data
 - More on Data Frames throughout the week!

Working with Vectors

Creating a Vector

- Use the `c()` function

```
name <- c("John Doe", "Jane Smith", "MacGillicuddy Jones", "Echo Shamus")
age <- c(36, 54, 82, 15)
favorite_color <- c("red", "orange", "green", "black")

## print the vectors
name
## [1] "John Doe"           "Jane Smith"          "MacGillicuddy Jones"
## [4] "Echo Shamus"
age
## [1] 36 54 82 15
favorite_color
## [1] "red"    "orange"  "green"   "black"
```

Accessing vector data

- Use numerical indexing
- R uses 1-based indexing
 - 1st vector element has index of 1
 - 2nd has an index of 2
 - 3rd has an index of 3
 - and so on

```
name[1]
## [1] "John Doe"
age[3]
## [1] 82
```

- R supports “slicing” (i.e. extracting multiple items)

```
favorite_color[c(2, 3)]
## [1] "orange" "green"
```

- Negative indices are omitted

```
age[-2]
## [1] 36 82 15
```

Some Useful Vector Operations

- `length()`: number of elements
- `sum()`: sum of all element values
- `unique()`: distinct values
- `sort()`: sort elements, omitting NAs
- `order()`: indices of sorted elements, NAs are last
- `rev()`: reverse the order
- `summary()`: simple statistics

```
a <- c(5, 5, 6, 7, 8, 4)
sum(a)
## [1] 35
length(a)
## [1] 6
unique(a)
## [1] 5 6 7 8 4
sort(a)
```

```
## [1] 4 5 5 6 7 8
order(a)
## [1] 6 1 2 3 4 5
a[order(a)]
## [1] 4 5 5 6 7 8
rev(a)
## [1] 4 8 7 6 5 5
summary(a)
##      Min. 1st Qu. Median     Mean 3rd Qu.     Max.
## 4.000   5.000   5.500   5.833   6.750   8.000
```

Handling Missing Data

- First consider the reason(s) for the missing data
 - e.g. concentrations that are below detectable levels?
- Sometimes NAs in data require special statistical methods
- Other times we can safely discard / ignore NA entries
- To remove NAs prior to a calculation:

```
y = c(1,NA,3,2,NA)
sum(y, na.rm=TRUE)
## [1] 6
```

Wrapping up day 1

The goal for today was to rapidly cover some of the essential aspects of R programming. For the remainder of the week you'll work at your own pace to get more of a hands-on deep dive into this material. If you run into trouble please don't hesitate to ask for help via Teams (QuaRantine Team), slack (QuaRantine Course), or email (Drs. Matott and Morgan) — whatever works best for you!

1.2 Day 2: Vectors and variables

Our overall goal for the next few days is to use *R* to create a daily log of quarantine activities.

Our goal for today is to become familiar with *R* vectors. Along the way we'll probably make data entry and other errors that will start to get us comfortable with *R*.

If you run into problems, reach out to the slack channel for support!

The astronaut Scott Kelly said that to survive a year on the International Space Station he found it essential to

- Follow a schedule – plan your day, and stick to the plan
- Pace yourselves – you've got a long time to accomplish tasks, so don't try to get everything done in the first week.
- Go outside – if Scott can head out to space, we should be able to make it to the back yard or around the block!
- Get a hobby – something not work related, and away from that evil little screen. Maybe it's as simple as rediscovering the joy of reading.
- Keep a journal
- Take time to connect – on a human level, with people you work with and people you don't!
- Listen to experts – Scott talked about relying on the mission controllers; for us maybe that's watching webinars or taking courses in new topics!
- Wash your hands!

I wanted to emphasize ‘follow a schedule’ and ‘keep a journal’. How can *R* help? Well, I want to create a short record of how I spend today, day 2 of my quarantine.

My first goal is to create *vectors* describing things I plan to do today. Let's start with some of these. To get up to speed, type the following into the *R* console, at the > prompt

```
1 + 2
```

Press the carriage return and remind yourself that *R* is a calculator, and knows how to work with numbers!

Now type an activity in your day, for instance I often start with

```
"check e-mail"
```

Now try assigning that to a variable, and displaying the variable, e.g.,

```
activity <- "check e-mail"
activity
## [1] "check e-mail"
```

OK, likely you have several activities scheduled. Create a *vector* of a few of these by concatenating individual values

```
c("check e-mail", "breakfast", "conference call", "webinar", "walk")
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"
```

Assign these to a variable

```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"
```

Create another vector, but this time the vector should contain the minutes spent on each activity

```
minutes <- c(20, 30, 60, 60, 60)
minutes
## [1] 20 30 60 60 60
```

So I spent 20 minutes checking email, 30 minutes having breakfast and things like that, I was in a conference call for 60 minutes, and then attended a webinar where I learned new stuff for another 60 minutes. Finally I went for a walk to clear my head and remember why I'm doing things.

Apply some basic functions to the variables, e.g., use `length()` to demonstrate that you for each `activity` you have recorded the `minutes`.

```
length(activity)
## [1] 5
length(minutes)
## [1] 5
```

Use `tail()` to select the last two activities (or `head()` to select the first two...)

```
tail(activity, 2)
## [1] "webinar" "walk"
tail(minutes, 2)
## [1] 60 60
```

R has other types of vectors. Create a logical vector that indicates whether each activity was ‘work’ activity or something you did for your own survival. We’ll say that checking email is a work-related activity!

```
is_work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
is_work
## [1] TRUE FALSE  TRUE  TRUE FALSE
```

1.3 Day 3: `factor()`, `Date()`, and `NA`

Yesterday we learned about `character`, `numeric`, and `logical` vectors in *R* (you may need to revisit previous notes and re-create these variables)

```
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"
minutes
## [1] 20 30 60 60 60
is_work
## [1] TRUE FALSE TRUE TRUE FALSE
```

Today we will learn about slightly more complicated vectors.

We created the logical vector `is_work` to classify each `activity` as either work-related or not. What if we had several different categories? For instance, we might want to classify the activities into categories inspired by astronaut Kelly's guidance. Categories might include: `connect` with others; go outside and `exercise`; `consult` experts; get a `hobby`; and (my own category, I guess) perform `essential` functions like eating and sleeping. So the values of `activity` could be classified as

```
classification <-
  c("connect", "essential", "connect", "consult", "exercise")
```

I want to emphasize a difference between the `activity` and `classification` variables. I want `activity` to be a character vector that could contain any description of an activity. But I want `classification` to be terms only from a limited set of possibilities. In *R*, I want `classification` to be a special type of vector called a `factor`, with the *values* of the vector restricted to a set of possible *levels* that I define. I create a factor by enumerating the possible *levels* that the factor can take on

```
levels <- c("connect", "exercise", "consult", "hobby", "essential")
```

And then tell *R* that the vector `classification` should be a factor with values taken from a particular set of levels

```
classification <- factor(
  c("connect", "essential", "connect", "consult", "exercise"),
  levels = levels
)
classification
## [1] connect  essential connect  consult   exercise
## Levels: connect exercise consult hobby essential
```

Notice that `activity` (a character vector) displays differently from `classification` (a factor)

```
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"
classification
## [1] connect  essential connect  consult   exercise
## Levels: connect exercise consult hobby essential
```

Also, some of the levels (e.g., `hobby`) have not been part of our schedule yet, but the factor still ‘knows’ about the level.

Notice also what happens when I try to use a value (`disconnect`) that is not a level of a factor

```
factor(c("connect", "disconnect"), levels = levels)
## [1] connect <NA>
## Levels: connect exercise consult hobby essential
```

The value with the unknown level is displayed as `NA`, for ‘not known’. `NA` values can be present in any vector, e.g.,

```
c(1, 2, NA, 4)
## [1] 1 2 NA 4
c("walk", "talk", NA)
## [1] "walk" "talk" NA
c(NA, TRUE, FALSE, TRUE, TRUE)
## [1] NA TRUE FALSE TRUE TRUE
```

This serves as an indication that the value is simply not available. Use `NA` rather than adopting some special code (e.g., `-99`) to indicate when a value is not available.

One other type of vector we will work a lot with are dates. All of my activities are for today, so I’ll start with a character vector with the same length as my activity vector, each indicating the date in a consistent month-day-year format

```
dates <- c("04-14-2020", "04-14-2020", "04-14-2020", "04-14-2020", "04-14-2020")
dates
## [1] "04-14-2020" "04-14-2020" "04-14-2020" "04-14-2020" "04-14-2020"
```

Incidentally, I could do this more efficiently using the `replicate` function

```
rep("04-14-2020", 5)
## [1] "04-14-2020" "04-14-2020" "04-14-2020" "04-14-2020" "04-14-2020"
```

And even better use `length()` to know for sure how many times I should replicate the character vector

```
rep("04-14-2020", length(activity))
## [1] "04-14-2020" "04-14-2020" "04-14-2020" "04-14-2020" "04-14-2020"
```

`dates` is a character vector, but it has specially meaning as a calendar date, *R* has a `Date class` that knows how to work with dates, for instance to calculate the number of days between two dates. We will *coerce date* to an object of class `Date` using a function `as.Date`. Here's our first attempt...

```
as.Date(dates)
```

... but this results in an error:

```
Error in charToDate(x) :
  character string is not in a standard unambiguous format
```

R doesn't know the format (month-day-year) of the dates we provide. The solution is to add a second argument to `as.Date()`. The second argument is a character vector that describes the date format. The format we use is "%m-%d-%Y", which says that we provide the %month first, then a hyphen, then the %day, another hyphen, and finally the four-digit %Year.

```
as.Date(dates, format = "%m-%d-%Y")
## [1] "2020-04-14" "2020-04-14" "2020-04-14" "2020-04-14" "2020-04-14"
```

Notice that the format has been standardized to year-month-day. Also notice that although the original value of `date` and the return from `as.Date()` look the same, they are actually of different *class*.

```
class(date)
## [1] "function"
class(as.Date(dates, format = "%m-%d-%Y"))
## [1] "Date"
```

R will use the information about class to enable specialized calculation on dates, e.g., to sort them or to determine the number of days between different dates. So here's our `date` vector as a `Date` object.

```
dates <- rep("04-14-2020", length(activity))
date <- as.Date(dates, format = "%m-%d-%Y")
date
## [1] "2020-04-14" "2020-04-14" "2020-04-14" "2020-04-14" "2020-04-14"
```

OK, time for a walk! See you tomorrow!

1.4 Day 4: Working with variables

Remember that *R* can act as a simple calculator, and that one can create new variables by assignment

```
x <- 1
x + 1
## [1] 2
y <- x + 1
y
## [1] 2
```

Let's apply these ideas to our `minutes` vector from earlier in the week.

```
minutes <- c(20, 30, 60, 60, 60)
```

We can perform basic arithmetic on vectors. Suppose we wanted to increase the time of each activity by 5 minutes

```
minutes + 5
## [1] 25 35 65 65 65
```

or to increase the time of the first two activities by 5 minutes, and the last three activities by 10 minutes

```
minutes + c(5, 5, 10, 10, 10)
## [1] 25 35 70 70 70
```

R has a very large number of *functions* that can be used on vectors. For instance, the average time spent on activities is

```
mean(minutes)
## [1] 46
```

while the total amount of time is

```
sum(minutes)
## [1] 230
```

Explore other typical mathematical transformations, e.g., `log()`, `log10()`, `sqr()` (square root), ... Check out the help pages for each, e.g., `?log`.

Explore the consequences of `NA` in a vector for functions like `mean()` and `sum()`.

```
x <- c(1, 2, NA, 3)
mean(x)
## [1] NA
```

R is saying that, since there is an unknown (`NA`) value in the vector, it cannot possibly know what the mean is! Tell *R* to remove the missing values before performing the calculation by adding the `na.rm = TRUE` argument

```
mean(x, na.rm = TRUE)
## [1] 2
```

Check out the help page `?mean` to find a description of the `na.rm` and other arguments.

It's possible to perform logical operations on vectors, e.g., to ask which activities lasted 60 minutes or more

```
minutes >= 60
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

Here's our `activity` vector

```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
```

The elements of this vector are numbered from 1 to 5. We can create a new vector that is a subset of this vector using `[` and an integer index, e.g., the second activity is

```
activity[2]
## [1] "breakfast"
```

The index can actually be a vector, so we could choose the second and fourth activity as

```
index <- c(2, 4)
activity[index]
## [1] "breakfast" "webinar"
```

In fact, we can use logical vectors for subsetting. Consider the activities that take sixty minutes or longer:

```
index <- minutes >= 60
activity[index]
## [1] "conference call" "webinar"           "walk"
```

We had previously characterized the activities as ‘work’ or otherwise.

```
is_work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
```

Use `is_work` to subset `activity` and identify the work-related activities

```
activity[is_work]
## [1] "check e-mail"    "conference call" "webinar"
```

How many minutes were work-related?

```
work_minutes <- minutes[is_work]
sum(work_minutes)
## [1] 140
```

What about not work related? `!` negates logical vectors, so

```
is_work
## [1] TRUE FALSE  TRUE  TRUE FALSE
!is_work
## [1] FALSE  TRUE FALSE FALSE  TRUE
non_work_minutes <- minutes[!is_work]
sum(non_work_minutes)
## [1] 90
```

Note that it doesn’t make sense to take the `mean()` of a character vector like `activity`, and *R* signals a warning and returns NA

```
mean(activity)
## Warning in mean.default(activity): argument is not numeric or logical: returning
## NA
## [1] NA
```

Nonetheless, there are many functions that *do* work on character vectors, e.g., the number of letters in each element `nchar()`, or transformation to upper-case

```
nchar(activity)
## [1] 12 9 15 7 4
toupper(activity)
## [1] "CHECK E-MAIL"      "BREAKFAST"        "CONFERENCE CALL" "WEBINAR"
## [5] "WALK"
```

1.5 Day 5 (Friday) Zoom check-in

1.5.1 Logistics

- Please join Microsoft Teams! Need help? Contact Adam.Kisailus at RoswellPark.org.

1.5.2 Review and trouble shoot (25 minutes; Martin)

Data representations

'Atomic' vectors

```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes <- c(20, 30, 60, 60, 60)
is_work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
```

`factor()` and `date()`

```
levels <- c("connect", "exercise", "consult", "hobby", "essential")
classification <- factor(
  c("connect", "essential", "connect", "consult", "exercise"),
  levels = levels
)

dates <- rep("04-14-2020", length(activity))
date <- as.Date(dates, format = "%m-%d-%Y")
```

Missing values

```
x <- c(1, 3, NA, 5)
sum(x)
## [1] NA
sum(x, na.rm = TRUE)
## [1] 9

factor(c("connect", "disconnect"), levels = levels)
## [1] connect <NA>
## Levels: connect exercise consult hobby essential
```

Functions and logical operators

```
x <- c(1, 3, NA, 5)
sum(x)
## [1] NA
sum(x, na.rm = TRUE)
## [1] 9

minutes >= 60
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

Subsetting vectors

- 1-based numeric indexes

```
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"

idx <- c(1, 3, 1)
activity[idx]
## [1] "check e-mail"      "conference call" "check e-mail"
```

- logical index

```
is_work
## [1] TRUE FALSE  TRUE  TRUE FALSE
activity[is_work]
## [1] "check e-mail"      "conference call" "webinar"

sum(minutes[is_work])
## [1] 140
```

- Maybe more interesting...

```
short <- minutes < 60
short
## [1] TRUE  TRUE FALSE FALSE FALSE
minutes[short]
## [1] 20 30
activity[short]
## [1] "check e-mail" "breakfast"
```

Other fun topics

`%in%`: a *binary operator*

- is each of the vector elements on the left-hand side *in* the set of elements on the right hand side

```
fruits <- c("banana", "apple", "grape", "orange", "kiwi")
c("apple", "orange", "hand sanitizer") %in% fruits
## [1] TRUE TRUE FALSE
```

named vectors (see Annual Estimates... table from census.gov)

- Define a named vector

```
state_populations <- c(
  Alabama = 4903185, Alaska = 731545, Arizona = 7278717, Arkansas = 3017804,
  California = 39512223, Colorado = 5758736, Connecticut = 3565287,
  Delaware = 973764, `District of Columbia` = 705749, Florida = 21477737,
  Georgia = 10617423, Hawaii = 1415872, Idaho = 1787065, Illinois = 12671821,
  Indiana = 6732219, Iowa = 3155070, Kansas = 2913314, Kentucky = 4467673,
  Louisiana = 4648794, Maine = 1344212, Maryland = 6045680, Massachusetts = 6892
  Michigan = 9986857, Minnesota = 5639632, Mississippi = 2976149,
  Missouri = 6137428, Montana = 1068778, Nebraska = 1934408, Nevada = 3080156,
  `New Hampshire` = 1359711, `New Jersey` = 8882190, `New Mexico` = 2096829,
  `New York` = 19453561, `North Carolina` = 10488084, `North Dakota` = 762062,
  Ohio = 11689100, Oklahoma = 3956971, Oregon = 4217737, Pennsylvania = 12801989,
  `Rhode Island` = 1059361, `South Carolina` = 5148714, `South Dakota` = 884659,
  Tennessee = 6829174, Texas = 28995881, Utah = 3205958, Vermont = 623989,
  Virginia = 8535519, Washington = 7614893, `West Virginia` = 1792147,
  Wisconsin = 5822434, Wyoming = 578759
)
```

- Computations on named vectors

```
## US population
sum(state_populations)
## [1] 328239523

## smallest states
head(sort(state_populations))
##           Wyoming          Vermont District of Columbia
##             578759            623989                705749
##           Alaska          North Dakota        South Dakota
##             731545            762062                884659

## largest states
head(sort(state_populations, decreasing = TRUE))
##      California         Texas       Florida      New York Pennsylvania      Illinois
##             328239523    28995881   21477737   19453561   12801989   1068778
```

```

##      39512223    28995881    21477737    19453561    12801989    12671821

## states with more than 10 million people
big <- state_populations[state_populations > 10000000]
big
##      California      Florida      Georgia      Illinois      New York
##      39512223    21477737    10617423    12671821    19453561
## North Carolina      Ohio      Pennsylvania      Texas
##      10488084    11689100    12801989    28995881
names(big)
## [1] "California"      "Florida"      "Georgia"      "Illinois"
## [5] "New York"        "North Carolina" "Ohio"        "Pennsylvania"
## [9] "Texas"

```

- Subset by name

```

## populations of California and New York
state_populations[c("California", "New York")]
## California      New York
##      39512223    19453561

```

1.5.3 Weekend activities (25 minutes; Shawn)

Writing *R* scripts

R scripts are convenient text files that we can use to save one or more lines of *R* syntax. Over the weekend you will get some experience working with *R* scripts. The example below will help you be a bit more prepared.

- In RStudio, click **File --> New File --> R Script** to create a new script file and open it in the editor.

If you've followed the daily coding activities throughout the week, you should have some *R* code that keeps track of your daily activities.

- If so, enter that code into your *R* script now.
- Otherwise, feel free to use the code below. Look for a **copy to clipboard** icon in the top-right of the code block. To copy the code block to your *R* script:
 - Click on the **copy to clipboard** icon
 - Place your cursor in your *R* script
 - Click **Edit --> Paste**:

```

## =====
## day 1 information
## =====
day1_activity = c("breakfast",
                  "check e-mail",
                  "projects",
                  "conference call",
                  "teams meeting",
                  "lunch",
                  "conference call",
                  "webinar")
day1_is_work = c(FALSE, TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, TRUE)
day1_minutes = c(30, 75, 120, 30, 60, 30, 60, 120)

n = length(day1_activity)
day1_total_hours = sum(day1_minutes) / 60
day1_work_hours = sum(day1_minutes[day1_is_work == TRUE]) / 60

cat("Total time recorded for day 1 : ", day1_total_hours,
    "hours, over", n, "activities\n")
cat("Total time working for day 1 : ", day1_work_hours, "hours \n\n")

```

Recall the discussion of factors and levels in Day 3; the code below leverages this but adds another level named `independent work`.

- If you've already got code to assign factors and levels to your daily activity, enter that code into your *R* script now.
- Otherwise, feel free to use the code below via the `copy to clipboard` procedure outlined above:

```

## =====
## Kelly, Morgan, and Matott's classification strategy
## =====
kmm_levels = c("connect",
               "exercise",
               "consult",
               "hobby",
               "essential",
               "independent work")
## manually map day 1 activity to appropriate kmm_levels
day1_classes = factor(
  c("essential", "connect", "independent work",
    "connect", "connect", "essential",
    "connect", "consult"),

```

```
    levels = kmm_levels
)
```

On day 3 you also got some experience working with dates. The code below stamps our day 1 activity data with an appropriately formatted date.

- If you've already got code to assign dates your daily activity, enter that code into your R script now.
- Otherwise, feel free to use the code below via the `copy to clipboard` procedure outlined above:

```
## =====
## Assign dates
## =====
day1_dates = rep("04-13-2020",length(day1_activity))
day1_dates = as.Date(day1_dates,format = "%m-%d-%Y")
```

[OPTIONAL ADVANCED MATERIAL]

Earlier today Dr. Morgan touched on named vectors. We can leverage named vectors to create a more general mapping between activities and levels. The code for this is given below. Try it and compare the result to your manual mapping!

```
## kmm_map is a named vector that maps activities to categories
kmm_map = c("breakfast"      = "essential",
           "check e-mail"   = "connect",
           "projects"       = "independent work",
           "requests"       = "independent work",
           "conference call" = "connect",
           "teams meeting"  = "connect",
           "lunch"          = "essential",
           "webinar"         = "consult",
           "walk"            = "exercise")
day1_classes = factor(kmm_map[day1_activity], levels = kmm_levels)
```

Saving *R* scripts

If you've been following along you should now have an *R* script that contains a bunch of code for keeping track of your daily activity log. Let's save this file:

- In RStudio, place your cursor anywhere in the script file

- click **File** --> **Save** (or press **CTRL+S**)
 - Name your file something like **daily_activity.R**.

Running **R** scripts

Now that we've created an *R* script you may be wondering "How do I run the code in the script?" There's actually a few ways to do this:

Option #1 (Run)

- Highlight the first block of the code (e.g the part where you recorded day 1 activity and maybe calculated amount of time worked).
- Click the --> **Run** icon in the top-right portion of the script editor window.
 - This will run the highlighted block of code. The output will appear in the RStudio console window along with an echo of the code itself.

Option #2 (Source)

- Click on the --> **Source** icon just to the right of the --> **Run** icon.
- This will run the entire script.
- Equivalent to entering into the console

```
source("daily_activity.R")
```

- Only the output generated by **print()** and **cat()** will appear in the RStudio console (i.e. the code in the script is not echoed to the console).

Option #3 (Source with Echo)

- Click on the downward pointing arrowhead next to the source button to open a dropdown menu
- In the dropdown menu, select **Source with Echo**
- This will run the entire script and the code in the script will be echoed to the RStudio console along with any output generated by **print()** and **cat()**.
- The echoed source and the normal output are not color-coded like they are when using the --> **Run** button.
- Equivalent to running

```
source("daily_activity.R", echo = TRUE, max = Inf)
```

Saving data

It can be useful to save objects created in an *R* script as a data file. These data files can be loaded or re-loaded into a new or existing *R* session.

For example, let's suppose you had an *R* script that mined a trove of Twitter feeds for sentiment data related to government responses to COVID-19. Suppose you ran the script for several weeks and collected lots of valuable data into a bunch of vectors. Even though the *R* code is saved as a script file, the data that the script is collecting would be lost once script stops running. Furthermore, due to the temporal nature of Twitter feeds, you wouldn't be able to collect the same data by simply re-running the script. Luckily, *R* provides several routines for saving and loading objects. Placing the appropriate code in your *R* script will ensure that your data is preserved even after the script stops running.

Saving individual *R* objects

R supports storing a single *R* object as an *.rds* file. For example, the code below saves the *day1_activity* vector to an *.rds* file. The *saveRDS()* function is the workhorse in this case and the *setwd()*, *getwd()*, and *file.path()* commands allow us to conveniently specify a name and location for the data file:

```
## =====
## creating .rds data files (for saving individual objects)
## =====
setwd("C:/Matott/MyQuarantine")
my_rds_file = file.path(getwd(), "day1_activity.rds")
my_rds_file # print value -- sanity check
saveRDS(day1_activity, my_rds_file)
```

Loading individual *R* objects

The complement to the *saveRDS()* function is the *readRDS()* function. It loads the *R* object stored in the specified file. In the example below a data file is loaded and stored as an object named *day1_activity_loaded*. Compare this object to the existing *day1_activity* object - they should be the same!

```
## =====
## Reading .rds data files (for loading individual objects)
## =====
setwd("C:/Matott/MyQuarantine")
```

```
my_rds_file = file.path(getwd(), "day1_activity.rds")
my_rds_file
day1_activity_loaded = readRDS(my_rds_file) # now load from disk
```

Saving multiple *R* objects

The `save()` function will save one or more objects into a `.Rdata` file (these are also known as `session` files). The example below saves various `day1` and related factor-level objects to an `.Rdata` file.

```
## =====
## creating .RData files (for saving multiple objects)
## =====
setwd("C:/Matott/MyQuarantine")
my_rdata_file = file.path(getwd(), "day1.rdata")
save(kmm_levels, kmm_map,
     day1_activity, day1_classes, day1_dates, day1_is_work,
     day1_minutes, day1_total_hours, day1_work_hours,
     file = my_rdata_file)
```

If you have many objects that you want to save, listing them all can be tedious. Fortunately, the `ls()` command provides a list of all objects in the current *R* session. The results of `ls()` can be passed along to the `save()` command and this will result in all objects being saved. An example of the required syntax is given below.

```
setwd("C:/Matott/MyQuarantine")
my_rdata_file = file.path(getwd(), "day1.rdata")
save(list = ls(), file = my_rdata_file)
```

Loading multiple *R* objects

The complement to the `save()` function is the `load()` function. This will load all objects stored in an `.Rdata` file into the current *R* session. Example syntax is given below:

```
## =====
## Reading .RData files (for loading multiple objects)
## =====
setwd("C:/Matott/MyQuarantine")
my_rdata_file = file.path(getwd(), "daily_activity.rdata")
load(my_rdata_file) # reload
```

It is also possible to load an `.Rdata` file using the RStudio interface.

- Click **Session** --> **Load Workspace** ...
- A file browser dialog will open
- Navigate to the `.rdata` file and select

Wrapping up day 5

Today we reviewed the concepts that you worked with throughout the week during your independent activity. We also troubleshooted any problems or questions that may have come up during this time. Finally, we previewed the creation and use of *R* scripts and learned about saving and loading objects. Over the weekend you will gain some more experience with these topics.

1.6 Day 6: *R* scripts

Some of you may have already started saving your R commands as script files. As the material gets more complicated (and more interesting) everyone will want to start doing this. Here is an example to get you started:

- Recall that we can create a script file in RStudio, click “File –> New File –> R Script” to create a new script file and open it in the editor



- By convention, R scripts have a `.R` extension (e.g. `my_script.R`)
 - In RStudio, click into your untitled script and click “File –> Save”
 - Name your file something fun like `my_first_script.R` and save it
- Use the `#` character for comments. Enter the following into your R Script file:

```
## This is my first R script
```

- Enter each command on a separate line. It’s also possible to enter multiple (short!) commands on a single line, separated by a semi-colon ;

```
x = "Hello world!"
y = 'Today is'; d = format(Sys.Date(), "%b %d, %Y")
cat(x, y, d)
```

- Use the “Run” button in RStudio to run the highlighted portion of an R script file. Try this on your simple R Script.



```
x = "Hello world!"; y = 'Today is'; d = format(Sys.Date(), "%b %d, %Y")
cat(x, y, d, "\n")
## Hello world! Today is Apr 28, 2020
```

- Alternatively, use “Run → Run All” to run an entire script file.

For today’s exercise, create a script file that summarizes your quarantine activities over several days. Use comments, white space (blank lines and spaces), and variable names to summarize each day. Here’s what I’ve got...

```
## 'classification' factor levels
levels <- c("connect", "exercise", "consult", "hobby", "essential")

## Quarantine log, day 1

activity_day_1 <-
  c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes_day_1 <- c(20, 30, 60, 60, 60)
is_work_day_1 <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
classification_day_1 <- factor(
  c("connect", "essential", "connect", "consult", "exercise"),
  levels = levels
)
date_day_1 <- as.Date(rep("04-14-2020", length(activity_day_1)), "%m-%d-%Y")

## Quarantine log, day 2

activity_day_2 <-
  c("check e-mail", "breakfast", "conference call", "webinar", "read a book")
minutes_day_2 <- c(20, 30, 60, 60, 60)
is_work_day_2 <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
classification_day_2 <- factor(
  c("connect", "essential", "connect", "consult", "hobby"),
  levels = levels
)
date_day_2 <- as.Date(rep("04-15-2020", length(activity_day_2)), "%m-%d-%Y")

## Quarantine log, day 3
```

```
activity_day_3 <-
  c("check e-mail", "breakfast", "webinar", "read a book")
minutes_day_3 <- c(20, 30, 60, 60)
is_work_day_3 <- c(TRUE, FALSE, TRUE, FALSE)
classification_day_3 <- factor(
  c("connect", "essential", "connect", "consult", "hobby"),
  levels = levels
)
date_day_3 <- as.Date(rep("04-16-2020", length(activity_day_3)), "%m-%d-%Y")
```

Try concatenating these values, e.g.,

```
activity <- c(activity_day_1, activity_day_2, activity_day_3)
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"              "check e-mail"     "breakfast"        "conference call"
## [9] "webinar"           "read a book"      "check e-mail"    "breakfast"
## [13] "webinar"           "read a book"
```

Save your script, quit *R* and *RStudio*, and restart *R*. Re-open and run the script to re-do your original work.

Think about how this makes your work *reproducible* from one day to the next, and how making your scientific work reproducible would be advantageous.

1.7 Day 7: Saving data

We've defined these variables

```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes <- c(20, 30, 60, 60, 60)
is_work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

levels <- c("connect", "exercise", "consult", "hobby", "essential")
classification <- factor(
  c("connect", "essential", "connect", "consult", "exercise"),
  levels = levels
)

dates <- rep("04-14-2020", length(activity))
date <- as.Date(dates, format = "%m-%d-%Y")
```

Individual variables can be saved to a file.

- Define the *path* to the file. The file extension is, by convention, ‘.rds’. We’ll use a temporary location

```
temporary_file_path <- tempfile(fileext = ".rds")
```

...but we could have chosen the destination interactively

```
interactive_file_path <- file.choose(new = TRUE)
```

...or provided path relative to the ‘current working directory’, or an absolute file path (use ‘/’ to specify paths on all operating systems, including Windows)

```
getwd()
relative_file_path <- "my_activity.rds"
absolute_file_path_on_macOS <- "/Users/ma38727/my_activity.rda"
```

- use `saveRDS()` to save a single object to a file

```
saveRDS(activity, temporary_file_path)
```

- use `readRDS()` to read the object back in

```
activity_from_disk <- readRDS(temporary_file_path)
activity_from_disk
## [1] "check e-mail"      "breakfast"          "conference call" "webinar"
## [5] "walk"
```

Use `save()` and `load()` to save and load several objects.

- Use `.RData` as the file extension. Usually we would NOT save to a temporary location, because the temporary location would be deleted when we ended our *R* session.

```
temporary_file_path <- tempfile(fileext = ".RData")
save(activity, minutes, file = temporary_file_path)
```

- Remove the objects from the *R* session, and verify that they are absent

```
rm(activity, minutes)
try(activity) # fails -- object not present
## Error in try(activity) : object 'activity' not found
```

- Load the saved objects

```
load(temporary_file_path)
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"
```

As an exercise...

- Chose a location to save your data, e.g., in the current working direcotry

```
getwd()    # Where the heck are we?
## [1] "C:/Matott/QuaRantine"
my_file_path <- "my_quaRantine.RData"
```

- Save the data

```
save(activity, minutes, is_work, classification, date, file = my_file_path)
```

- Now the moment of truth. Quit *R* without saving your workspace

```
quit(save = FALSE)
```

- Start a new session of *R*, and verify that your objects are not present

```
ls() # list objects available in the '.GlobalEnv' -- there should be none
## character(0)
try(activity) # nope, not there...
## Error in try(activity) : object 'activity' not found
```

- Create a path to the saved data file

```
my_file_path <- "my_quaRantine.RData"
```

- Load the data and verify that it is correct

```
load(my_file_path)
activity
## [1] "check e-mail"      "breakfast"        "conference call" "webinar"
## [5] "walk"
minutes
## [1] 20 30 60 60 60
is_work
## [1] TRUE FALSE  TRUE   TRUE FALSE
date
```

```
## [1] "2020-04-14" "2020-04-14" "2020-04-14" "2020-04-14" "2020-04-14"  
classification  
## [1] connect essential connect consult exercise  
## Levels: connect exercise consult hobby essential
```

See you in zoom on Monday!

Chapter 2

The data frame

2.1 Day 8 (Monday) Zoom check-in

Logistics

- Remember to use the QuaRantine Microsoft Team. Your Roswell credentials are required, and you must have been invited (by Adam.Kisailus at RoswellPark.org)
- We're thinking of having a 'networking' hour after Friday's class (so 3pm and after) where we'll break into smaller groups (if necessary) and provide an opportunity for people to turn on their video and audio so that we can increase the amount of interaction. Likely the first networking hour will be a round of introductions / what you hope to get out of the course / etc., and maybe brief discussion of topics arising.

Review and troubleshoot (15 minutes)

Saving and loading objects

Scripts

The data frame (40 minutes)

Concept

Recall from Day 1:

- Data frames are handy containers for experimental data.

- Like a spreadsheet, a data frame has rows and columns
- The columns of a data frame contain measurements describing each individual
 - Each measurement could refer to a different type of object (numeric, string, etc.)
 - Measurements could be physical observations on samples, e.g., `height`, `weight`, `age`, `minutes` an activity lasts, etc.
 - Measurements might also describe how the row is classified, e.g., `activity`, `is work?`, `classification`, `date`, etc.
- The rows of a data frame represent a ‘tuple’ of measurements corresponding to an experimental observation, e.g.,
 - Note: you must ensure units are consistent across tuples!
- Rows and columns can be assigned names.

Create a simple data frame

```
heights <- c(72, 65, 68)
weights <- c(190, 130, 150)
ages <- c(44, 35, 37)
df <- data.frame(heights, weights, ages)
df
##   heights weights ages
## 1      72     190    44
## 2      65     130    35
## 3      68     150    37
```

It’s possible to update the column names, and to provide row names...

```
named_df <- data.frame(heights, weights, ages)
colnames(named_df) <- c("hgt_inches", "wgt_lbs", "age_years")
rownames(named_df) <- c("John Doe", "Pat Jones", "Sara Grant")
named_df
##           hgt_inches wgt_lbs age_years
## John Doe          72     190       44
## Pat Jones          65     130       35
## Sara Grant         68     150       37
```

...but it’s often better practice to name columns at time of creation, and to store all information as columns (rather than designating one column as a ‘special’ row name)

- Here's our first attempt

```
data.frame(
  person = c("John Doe", "Pat Jones", "Sara Grant"),
  hgt_inches = heights, wgt_lbs = weights, age_years = ages
)
##      person hgt_inches wgt_lbs age_years
## 1  John Doe       72     190       44
## 2  Pat Jones       65     130       35
## 3 Sara Grant       68     150       37
```

- It's unsatisfactory because by default *R* treats character vectors as **factor**. We'd like them to plain-old character vectors. To accomplish this, we add the **stringsAsFactors = FALSE** argument

```
df <- data.frame(
  person = c("John Doe", "Pat Jones", "Sara Grant"),
  hgt_inches = heights, wgt_lbs = weights, age_years = ages,
  stringsAsFactors = FALSE
)
df
##      person hgt_inches wgt_lbs age_years
## 1  John Doe       72     190       44
## 2  Pat Jones       65     130       35
## 3 Sara Grant       68     150       37
```

Adding and deleting rows

Adding rows

- Add a row with **rbind()**

```
more_people <- c("Bob Kane", "Kari Patra", "Sam Groe")
more_heights <- c(61, 68, 70)
more_weights <- c(101, 134, 175)
more_ages <- c(13, 16, 24)
more_df <- data.frame(
  person = more_people,
  hgt_inches = more_heights, wgt_lbs = more_weights, age_years = more_ages,
  stringsAsFactors = FALSE
)

df_all <- rbind(df, more_df)
df_all
```

```
##      person hgt_inches wgt_lbs age_years
## 1  John Doe       72     190      44
## 2  Pat Jones      65     130      35
## 3 Sara Grant      68     150      37
## 4  Bob Kane       61     101      13
## 5 Kari Patra      68     134      16
## 6  Sam Groe        70     175      24
```

- R often has more than one way to perform an operation. We'll see `add_rows()` later in the course.

Delete rows using a logical vector...

- Create a logical or numeric index indicating the rows to be deleted

```
## suppose the study has some dropouts ....
dropouts <- c("Bob Kane", "John Doe")

## create a logical vector indicating which rows should be dropped
drop <- df_all$person %in% dropouts

## ...but we actually want to know which rows to `keep`
keep <- !drop
```

- Subset the data frame with the logical vector indicating the rows we would like to keep

```
df_all[keep,]
##      person hgt_inches wgt_lbs age_years
## 2  Pat Jones      65     130      35
## 3 Sara Grant      68     150      37
## 5 Kari Patra      68     134      16
## 6  Sam Groe        70     175      24
```

...or a numeric vector

- Create a vector containing the rows to be deleted

```
# suppose the study has some dropouts ....
dropouts = c(2, 3)
```

- Use a minus sign - to indicated that these rows should be *dropped*, rather than kept

```

df_all # refresh my memory about df contents ....
##      person hgt_inches wgt_lbs age_years
## 1  John Doe       72     190      44
## 2  Pat Jones      65     130      35
## 3 Sara Grant      68     150      37
## 4  Bob Kane       61     101      13
## 5 Kari Patra      68     134      16
## 6  Sam Groe        70     175      24
df_remaining <- df_all[!dropouts, ]

df_remaining # items 2 and 3 are dropped!
##      person hgt_inches wgt_lbs age_years
## 1  John Doe       72     190      44
## 4  Bob Kane       61     101      13
## 5 Kari Patra      68     134      16
## 6  Sam Groe        70     175      24

```

Some useful data frame operations

Try these out on your simple data frames `df` and `named_df`:

- `str(df)` # structure (NOT string!)(sorry Python programmers ;)
- `dim(df)` # dimensions
- `View(df)` # open tabular view of data frame
- `head(df)` # first few rows
- `tail(df)` # last few rows
- `names(df)` # column names
- `colnames(df)` # column names
- `rownames(df)` # row names

Writing, reading, and spreadsheets

Saving a `data.frame`

- We *could* save the `data.frame` as an *R* object, using the methods from quarantine day 7
- Often, better practice (e.g., to make it easy to share data with others in our lab) is to save data as a text file
 - A ‘csv’ file is one example
 - A plain text file

- The first line contains column names
- Each line of the text file represents a row of the data frame
- Columns within a row are separated by a comma, ,
- Example: save `df_all` to a temporary file location

```
file <- tempfile() # temporary file
## file <- file.choose()
## file <- "df_all.csv"
## file <- "/Users/ma38727/MyQuarantine/df_all.csv"
write.csv(df_all, file, row.names = FALSE)
```

- now, read the data back in from the temporary location

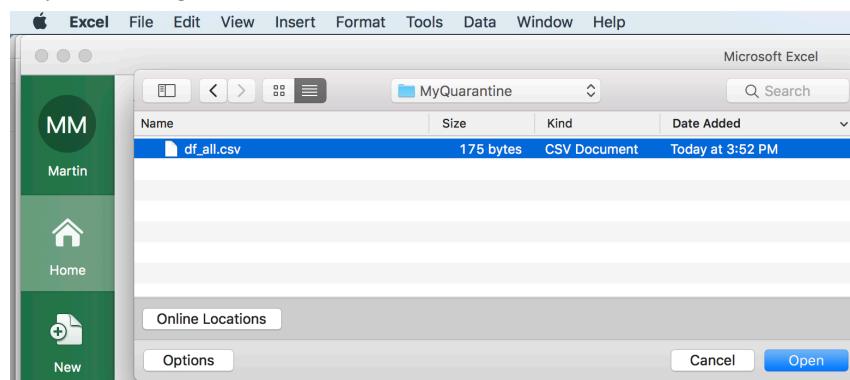
```
df_all_from_file <- read.csv(file, stringsAsFactors = FALSE)
df_all_from_file
##           person hgt_inches wgt_lbs age_years
## 1      John Doe       72     190      44
## 2    Pat Jones       65     130      35
## 3  Sara Grant       68     150      37
## 4   Bob Kane        61     101      13
## 5 Kari Patra        68     134      16
## 6   Sam Groe         70     175      24
```

R and spreadsheets

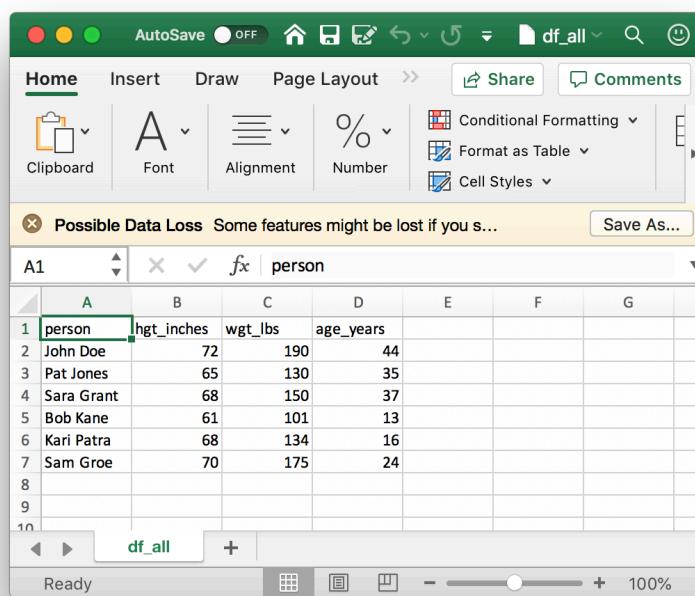
- A CSV file is a common way to move data from *R* to a spreadsheet, and vice versa. Following along with the example above, write `df_all` to a CSV file.

```
file <- "/Users/ma38727/MyQuarantine/df_all.csv" # a location on (my) disk
write.csv(df_all, file, row.names = FALSE)
```

- Now open a spreadsheet application like Excel and navigate to the directory containing the file



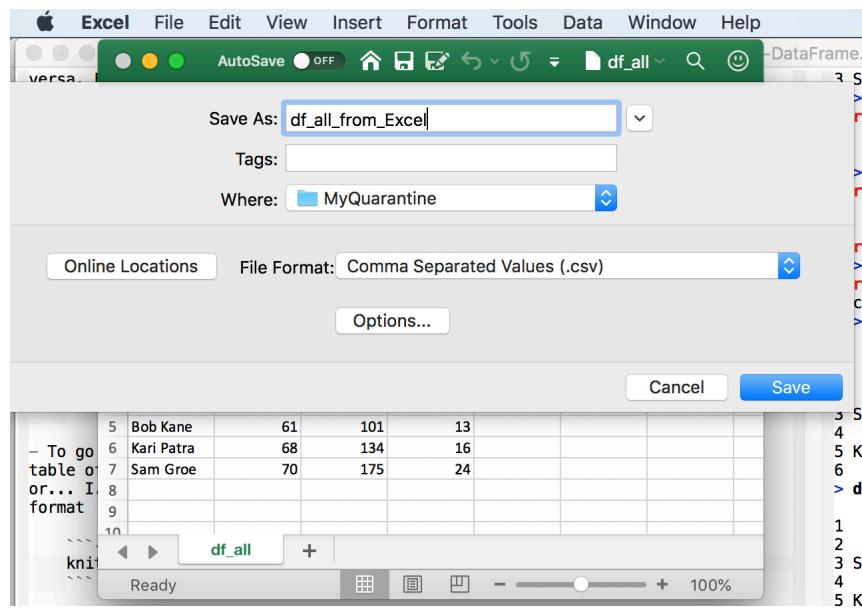
- ... and open the file



A screenshot of Microsoft Excel showing a table of data. The table has 7 columns (A-G) and 8 rows (1-8). The first row contains column headers: 'person' (A1), 'hgt_inches' (B1), 'wgt_lbs' (C1), and 'age_years' (D1). The subsequent rows contain data for seven individuals: John Doe (row 2), Pat Jones (row 3), Sara Grant (row 4), Bob Kane (row 5), Kari Patra (row 6), and Sam Groe (row 7). Row 8 is empty. The table is displayed on a sheet named 'df_all'. The Excel ribbon is visible at the top, showing the 'Home' tab is selected. A message bar at the top says 'Possible Data Loss Some features might be lost if you s...'. The status bar at the bottom right shows '100%'.

person	hgt_inches	wgt_lbs	age_years			
John Doe	72	190	44			
Pat Jones	65	130	35			
Sara Grant	68	150	37			
Bob Kane	61	101	13			
Kari Patra	68	134	16			
Sam Groe	70	175	24			

- To go from Excel to *R*, make sure your spreadsheet is a simple rectangular table of rows and columns, without ‘merged’ cells or fancy column formating, or... i.e., your spreadsheet should be as simple as the one we imported from *R*. Then save the file in CSV format



- ...and import it into R

```
df_all_from_Excel <- read.csv(
  "/Users/ma38727/MyQuarantine/df_all_from_Excel.csv",
  stringsAsFactors = FALSE
)
```

An alternative way of working with `data.frame()`

- `with()`: column selection and computation
- `within()`: update or add columns
- `subset()`: row and column subset
- Our quarantine log, day 1

```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes <- c(20, 30, 60, 60, 60)
is_work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

levels <- c("connect", "exercise", "consult", "hobby", "essential")
classification <- factor(
  c("connect", "essential", "connect", "consult", "exercise"),
  levels = levels
)
```

```

dates <- rep("04-14-2020", length(activity))
date <- as.Date(dates, format = "%m-%d-%Y")

log <- data.frame(
  activity, minutes, is_work, classification, date,
  stringsAsFactors = FALSE
)
log
##           activity   minutes   is_work classification      date
## 1      check e-mail     20    TRUE      connect 2020-04-14
## 2        breakfast     30   FALSE      essential 2020-04-14
## 3 conference call     60    TRUE      connect 2020-04-14
## 4       webinar       60    TRUE      consult 2020-04-14
## 5         walk        60   FALSE      exercise 2020-04-14

```

Summarization

- Use `with()` to simplify variable reference
- Create a new `data.frame()` containing the summary

```

with(log, {
  data.frame(
    days_in_quarantine = length(unique(date)),
    total_minutes = sum(minutes),
    work_activities = sum(is_work),
    other_activities = sum(!is_work)
  )
})
##   days_in_quarantine total_minutes work_activities other_activities
## 1                  1            230                 3                  2

```

Summarization by group

- `aggregate()`

```

## minutes per day spent on each activity, from the quarantine_log
aggregate(minutes ~ activity, log, sum)
##           activity   minutes
## 1        breakfast     30
## 2      check e-mail     20
## 3 conference call     60
## 4         walk        60

```

```

## 5      webinar      60

## minutes per day spent on each classification
aggregate(minutes ~ classification, log, sum)
##   classification minutes
## 1           connect     80
## 2       exercise      60
## 3       consult      60
## 4   essential      30

## non-work activities per day
aggregate(!is_work ~ date, log, sum)
##           date !is_work
## 1 2020-04-14          2

```

This week's activities (5 minutes)

Goal: retrieve and summarize COVID 19 cases in Erie county and nationally

2.2 Day 9: Creation and manipulation

Creation

Last week we created vectors summarizing our quarantine activities

```

activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes <- c(20, 30, 60, 60, 60)
is_work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

levels <- c("connect", "exercise", "consult", "hobby", "essential")
classification <- factor(
  c("connect", "essential", "connect", "consult", "exercise"),
  levels = levels
)

dates <- rep("04-14-2020", length(activity))
date <- as.Date(dates, format = "%m-%d-%Y")

```

Each of these vectors is the same length, and are related to one another in a specific way – the first element of `activity`, ‘check e-mail’, is related to the first element of `minutes`, ‘20’, and to `is_work`, etc.

Use `data.frame()` to construct an object containing each of these vectors

- Each argument to `data.frame()` is a vector representing a column
- The `stringsAsFactors = FALSE` argument says that character vectors should NOT be automatically coerced to factors

```
activities <- data.frame(
  activity, minutes, is_work, classification, date,
  stringsAsFactors = FALSE
)
activities
##           activity   minutes is_work classification      date
## 1     check e-mail      20    TRUE      connect 2020-04-14
## 2       breakfast      30   FALSE      essential 2020-04-14
## 3 conference call      60    TRUE      connect 2020-04-14
## 4      webinar         60    TRUE      consult 2020-04-14
## 5        walk          60   FALSE      exercise 2020-04-14
```

- We can query the object we've created for its `class()`, `dim()`ensions, take a look at the `head()` or `tail()` of the object, etc. `names()` returns the column names.

```
class(activities)
## [1] "data.frame"
dim(activities)      # number of rows and columns
## [1] 5 5
head(activities, 3) # first three rows
##           activity   minutes is_work classification      date
## 1     check e-mail      20    TRUE      connect 2020-04-14
## 2       breakfast      30   FALSE      essential 2020-04-14
## 3 conference call      60    TRUE      connect 2020-04-14
names(activities)
## [1] "activity"        "minutes"        "is_work"        "classification"
## [5] "date"
```

Column selection

Use `[` to select rows and columns

- `activities` is a two-dimensional object
- Subset the data to contain the first and third rows and the first and fourth columns

```
activities[c(1, 3), c(1, 4)]
##           activity classification
## 1    check e-mail      connect
## 3 conference call      connect
```

- Subset columns by name

```
activities[c(1, 3), c("activity", "is_work")]
##           activity is_work
## 1    check e-mail    TRUE
## 3 conference call    TRUE
```

- Subset only by row or only by column by omitting the subscript index for that dimension

```
activities[c(1, 3), ]                      # all columns for rows 1 and 3
##           activity minutes is_work classification      date
## 1    check e-mail     20    TRUE      connect 2020-04-14
## 3 conference call    60    TRUE      connect 2020-04-14
activities[, c("activity", "minutes")] # all rows for columns 1 and 2
##           activity minutes
## 1    check e-mail     20
## 2    breakfast       30
## 3 conference call    60
## 4    webinar         60
## 5    walk            60
```

- Be careful when selecting a single column!

- By default, *R* returns a *vector*

```
activities[, "classification"]
## [1] connect  essential connect  consult  exercise
## Levels: connect exercise consult hobby essential
```

- Use `drop = FALSE` to return a `data.frame`

```
activities[, "classification", drop = FALSE]
##   classification
## 1      connect
## 2    essential
## 3      connect
## 4     consult
## 5    exercise
```

Use `$` or `[[` to select a column

- Selection of individual columns as vectors is easy

```
activities$classification
## [1] connect  essential connect  consult   exercise
## Levels: connect exercise consult hobby essential
```

- An alternative, often used in scripts, is to use `[[`, which requires the name of a variable provided as a character vector

```
activities[["classification"]]
## [1] connect  essential connect  consult   exercise
## Levels: connect exercise consult hobby essential

colname <- "classification"
activities[[colname]]
## [1] connect  essential connect  consult   exercise
## Levels: connect exercise consult hobby essential
```

Column selection and subsetting are often combined, e.g., to create a `data.frame` of work-related activities, or work-related activities lasting 60 minutes or longer

```
work_related_activities <- activities[ activities$is_work == TRUE, ]
work_related_activities
##           activity minutes is_work classification      date
## 1    check e-mail      20     TRUE    connect 2020-04-14
## 3 conference call     60     TRUE    connect 2020-04-14
## 4      webinar       60     TRUE   consult 2020-04-14

row_idx <- activities$is_work & (activities$minutes >= 60)
activities[row_idx,]
##           activity minutes is_work classification      date
## 3 conference call     60     TRUE    connect 2020-04-14
## 4      webinar       60     TRUE   consult 2020-04-14
```

Adding or updating columns

Use `$` or `[` or `[[` to add a new column,

```
activities$is_long_work <- activities$is_work & (activities$minutes >= 60)
activities
##           activity minutes is_work classification      date is_long_work
## 1    check e-mail      20     TRUE    connect 2020-04-14      FALSE
```

```

## 2      breakfast    30 FALSE    essential 2020-04-14 FALSE
## 3 conference call 60 TRUE     connect 2020-04-14 TRUE
## 4      webinar     60 TRUE     consult 2020-04-14 TRUE
## 5      walk        60 FALSE    exercise 2020-04-14 FALSE

## ...another way of doing the same thing
activities[["is_long_work"]] <- activities$is_work & (activities$minutes >= 60)

## ...and another way
activities[, "is_long_work"] <- activities$is_work & (activities$minutes >= 60)

```

Columns can be updated in the same way

```

activities$activity <- toupper(activities$activity)
activities
#> #>   activity minutes is_work classification      date is_long_work
#> #> 1  CHECK E-MAIL    20  TRUE    connect 2020-04-14 FALSE
#> #> 2  BREAKFAST     30 FALSE    essential 2020-04-14 FALSE
#> #> 3 CONFERENCE CALL 60  TRUE    connect 2020-04-14 TRUE
#> #> 4  WEBINAR       60  TRUE    consult 2020-04-14 TRUE
#> #> 5      WALK        60 FALSE    exercise 2020-04-14 FALSE

```

Reading and writing

Create a file path to store a ‘csv’ file. From day 7, the path could be temporary, chosen interactively, a relative path, or an absolute path

```

## could be any of these...
##
## interactive_file_path <- file.choose(new = TRUE)
## getwd()
## relative_file_path <- "my_activity.rds"
## absolute_file_path_on_macOS <- "/Users/ma38727/my_activity.rda"
##
## ... but we'll use
temporary_file_path <- tempfile(fileext = ".csv")

```

Use `write.csv()` to save the `data.frame` to disk as a plain text file in ‘csv’ (comma-separated value) format. The `row.names = FALSE` argument means that the row indexes are not saved to the file (row names are created when data is read in using `read.csv()`).

```
write.csv(activities, temporary_file_path, row.names = FALSE)
```

If you wish, use RStudio File -> Open File to navigate to the location where you saved the file, and open it. You could also open the file in Excel or other spreadsheet. Conversely, you can take an Excel sheet and export it as a csv file for reading into *R*.

Use `read.csv()` to import a plain text file formatted as csv

```
imported_activities <- read.csv(temporary_file_path, stringsAsFactors = FALSE)
imported_activities
##           activity minutes is_work classification      date is_long_work
## 1     CHECK E-MAIL      20    TRUE   connect 2020-04-14    FALSE
## 2       BREAKFAST      30   FALSE  essential 2020-04-14    FALSE
## 3 CONFERENCE CALL      60    TRUE   connect 2020-04-14    TRUE
## 4      WEBINAR         60    TRUE  consult 2020-04-14    TRUE
## 5        WALK          60   FALSE exercise 2020-04-14    FALSE
```

Note that some information has not survived the round-trip – the `classification` and `date` columns are plain character vectors.

```
class(imported_activities$classification)
## [1] "character"
class(imported_activities$date)
## [1] "character"
```

Update these to be a `factor()` with specific levels, and a `Date`. ‘

```
levels <- c("connect", "exercise", "consult", "hobby", "essential")
imported_activities$classification <- factor(
  imported_activities$classification,
  levels = levels
)

imported_activities$date <- as.Date(imported_activities$date, format = "%Y-%m-%d")

imported_activities
##           activity minutes is_work classification      date is_long_work
## 1     CHECK E-MAIL      20    TRUE   connect 2020-04-14    FALSE
## 2       BREAKFAST      30   FALSE  essential 2020-04-14    FALSE
## 3 CONFERENCE CALL      60    TRUE   connect 2020-04-14    TRUE
## 4      WEBINAR         60    TRUE  consult 2020-04-14    TRUE
## 5        WALK          60   FALSE exercise 2020-04-14    FALSE
```

Reading from a remote file (!)

- Visit the New York Times csv file daily tally of COVID-19 cases in all US counties.
- Read the data into an *R* `data.frame`

```
url <-
  "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read.csv(url, stringsAsFactors = FALSE)
```

- Explore the data

```
class(us)
## [1] "data.frame"
dim(us)
## [1] 92592      6
head(us)
##           date   county     state fips cases deaths
## 1 2020-01-21 Snohomish Washington 53061     1      0
## 2 2020-01-22 Snohomish Washington 53061     1      0
## 3 2020-01-23 Snohomish Washington 53061     1      0
## 4 2020-01-24      Cook    Illinois 17031     1      0
## 5 2020-01-24 Snohomish Washington 53061     1      0
## 6 2020-01-25   Orange   California 6059     1      0
```

- Subset the data to only New York state or Erie county

```
ny_state <- us[us$state == "New York",]
dim(ny_state)
## [1] 2362      6

erie <- us[(us$state == "New York") & (us$county == "Erie"), ]
erie
##           date   county     state fips cases deaths
## 2569 2020-03-15    Erie New York 36029     3      0
## 3028 2020-03-16    Erie New York 36029     6      0
## 3544 2020-03-17    Erie New York 36029     7      0
## 4141 2020-03-18    Erie New York 36029     7      0
## 4870 2020-03-19    Erie New York 36029    28      0
## 5717 2020-03-20    Erie New York 36029    31      0
## 6711 2020-03-21    Erie New York 36029    38      0
## 7805 2020-03-22    Erie New York 36029    54      0
## 9003 2020-03-23    Erie New York 36029    87      0
## 10314 2020-03-24   Erie New York 36029   107      0
```

```

## 11754 2020-03-25 Erie New York 36029 122 0
## 13367 2020-03-26 Erie New York 36029 134 2
## 15111 2020-03-27 Erie New York 36029 219 6
## 16951 2020-03-28 Erie New York 36029 354 6
## 18888 2020-03-29 Erie New York 36029 380 6
## 20938 2020-03-30 Erie New York 36029 443 8
## 23078 2020-03-31 Erie New York 36029 438 8
## 25282 2020-04-01 Erie New York 36029 553 12
## 27543 2020-04-02 Erie New York 36029 734 19
## 29865 2020-04-03 Erie New York 36029 802 22
## 32253 2020-04-04 Erie New York 36029 945 26
## 34686 2020-04-05 Erie New York 36029 1059 27
## 37159 2020-04-06 Erie New York 36029 1163 30
## 39673 2020-04-07 Erie New York 36029 1163 36
## 42226 2020-04-08 Erie New York 36029 1205 38
## 44803 2020-04-09 Erie New York 36029 1362 46
## 47416 2020-04-10 Erie New York 36029 1409 58
## 50069 2020-04-11 Erie New York 36029 1472 62
## 52742 2020-04-12 Erie New York 36029 1571 75
## 55426 2020-04-13 Erie New York 36029 1624 86
## 58126 2020-04-14 Erie New York 36029 1668 99
## 60842 2020-04-15 Erie New York 36029 1751 110
## 63570 2020-04-16 Erie New York 36029 1850 115
## 66314 2020-04-17 Erie New York 36029 1929 115
## 69072 2020-04-18 Erie New York 36029 1997 115
## 71840 2020-04-19 Erie New York 36029 2070 146
## 74617 2020-04-20 Erie New York 36029 2109 153
## 77398 2020-04-21 Erie New York 36029 2147 161
## 80188 2020-04-22 Erie New York 36029 2233 174
## 82989 2020-04-23 Erie New York 36029 2450 179
## 85793 2020-04-24 Erie New York 36029 2603 184
## 88609 2020-04-25 Erie New York 36029 2773 199
## 91427 2020-04-26 Erie New York 36029 2954 205

```

2.3 Day 10: `subset()`, `with()`, and `within()`

`subset()`

`subset()`ing a `data.frame`

- Read the New York Times csv file summarizing COVID cases in the US.

```
url <-
  "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read.csv(url, stringsAsFactors = FALSE)
```

- Create subsets, e.g., to include only New York state, or only Erie county

```
ny_state <- subset(us, state == "New York")
dim(ny_state)
## [1] 2362    6
tail(ny_state)
##           date   county state fips cases deaths
## 91466 2020-04-26 Warren New York 36113 135     8
## 91467 2020-04-26 Washington New York 36115 127     0
## 91468 2020-04-26 Wayne New York 36117 65      0
## 91469 2020-04-26 Westchester New York 36119 27664 1054
## 91470 2020-04-26 Wyoming New York 36121 59      3
## 91471 2020-04-26 Yates New York 36123 17      1

erie <- subset(us, (state == "New York") & county == "Erie")
dim(erie)
## [1] 43    6
tail(erie)
##           date   county state fips cases deaths
## 77398 2020-04-21 Erie New York 36029 2147   161
## 80188 2020-04-22 Erie New York 36029 2233   174
## 82989 2020-04-23 Erie New York 36029 2450   179
## 85793 2020-04-24 Erie New York 36029 2603   184
## 88609 2020-04-25 Erie New York 36029 2773   199
## 91427 2020-04-26 Erie New York 36029 2954   205
```

`with()`

Use `with()` to simplify column references

- Goal: calculate maximum number of cases in the Erie county data subset
- First argument: a `data.frame` containing data to be manipulated – `erie`
- Second argument: an *expression* to be evaluated, usually referencing columns in the data set – `max(cases)`
 - E.g., Calculate the maximum number of cases in the `erie` subset

```
with(erie, max(cases))
## [1] 2954
```

Second argument can be more complicated, using {} to enclose several lines.

- E.g., Calculate the number of new cases, and then reports the average number of new cases per day. We will use `diff()`
 - `diff()` calculates the difference between successive values of a vector

```
x <- c(1, 1, 2, 3, 5, 8)
diff(x)
## [1] 0 1 1 2 3
```

- The length of `diff(x)` is one less than the length of `x`

```
length(x)
## [1] 6
length(diff(x))
## [1] 5
```

- `new_cases` is the `diff()` of successive values of `cases`, with an implicit initial value equal to 0.

```
with(erie, {
  new_cases <- diff(c(0, cases))
  mean(new_cases)
})
## [1] 68.69767
```

`within()`

Adding and updating columns `within()` a `data.frame`

- First argument: a `data.frame` containing data to be updated – `erie`
- Second argument: an expression of one or more variable assignments, the assignments create new columns in the `data.frame`.
- Example: add a `new_cases` column

```

erie_new_cases <- within(erie, {
  new_cases <- diff(c(0, cases))
})
head(erie_new_cases)
##           date county    state fips cases deaths new_cases
## 2569 2020-03-15 Erie New York 36029     3     0       3
## 3028 2020-03-16 Erie New York 36029     6     0       3
## 3544 2020-03-17 Erie New York 36029     7     0       1
## 4141 2020-03-18 Erie New York 36029     7     0       0
## 4870 2020-03-19 Erie New York 36029    28     0      21
## 5717 2020-03-20 Erie New York 36029    31     0       3

```

2.4 Day 11: `aggregate()` and an initial work flow

`aggregate()` for summarizing columns by group

Goal: summarize maximum number of cases by county in New York state

Setup

- Read and subset the New York Times data to contain only New York state data

```

url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties"
us <- read.csv(url, stringsAsFactors = FALSE)

```

```

ny_state <- subset(us, state == "New York")

```

`aggregate()`

- First argument: a `formula - cases ~ county`
 - Right-hand side: the variable to be used to subset (group) the data – `county`
 - Left-hand side: the variable to be used in the aggregation function – `cases`
- Second argument: source of data – `ny_state`
- Third argument: the function to be applied to each subset of data – `max`
- Maximum number of cases by county:

```
max_cases_by_county <- aggregate( cases ~ county, ny_state, max )
```

Exploring the data summary

- Subset to some interesting ‘counties’

```
head(max_cases_by_county)
##           county   cases
## 1      Albany     948
## 2    Allegany      35
## 3    Broome     257
## 4 Cattaraugus     45
## 5    Cayuga      47
## 6 Chautauqua     29
subset(
  max_cases_by_county,
  county %in% c("New York City", "Westchester", "Erie")
)
##           county   cases
## 14          Erie    2954
## 29 New York City 158268
## 57 Westchester  27664
```

Help: `?aggregate.formula`

An initial work flow

Data input

- From a remote location

```
url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read.csv(url, stringsAsFactors = FALSE)

class(us)
## [1] "data.frame"
dim(us)
## [1] 92592      6
head(us)
##       date   county      state fips cases deaths
## 1 2020-01-21 Snohomish Washington 53061      1      0
## 2 2020-01-22 Snohomish Washington 53061      1      0
```

```
## 3 2020-01-23 Snohomish Washington 53061 1 0
## 4 2020-01-24 Cook Illinois 17031 1 0
## 5 2020-01-24 Snohomish Washington 53061 1 0
## 6 2020-01-25 Orange California 6059 1 0
```

Cleaning

- `date` is a plain-old `character` vector, but should be a `Date`.

```
class(us$date) # oops, should be 'Date'
## [1] "character"
```

- Update, method 1

```
us$date <- as.Date(us$date, format = "%Y-%m-%d")
head(us)
##      date    county    state fips cases deaths
## 1 2020-01-21 Snohomish Washington 53061 1 0
## 2 2020-01-22 Snohomish Washington 53061 1 0
## 3 2020-01-23 Snohomish Washington 53061 1 0
## 4 2020-01-24 Cook Illinois 17031 1 0
## 5 2020-01-24 Snohomish Washington 53061 1 0
## 6 2020-01-25 Orange California 6059 1 0
```

- Update, method 2

```
us <- within(us, {
  date = as.Date(date, format = "%Y-%m-%d")
})
head(us)
##      date    county    state fips cases deaths
## 1 2020-01-21 Snohomish Washington 53061 1 0
## 2 2020-01-22 Snohomish Washington 53061 1 0
## 3 2020-01-23 Snohomish Washington 53061 1 0
## 4 2020-01-24 Cook Illinois 17031 1 0
## 5 2020-01-24 Snohomish Washington 53061 1 0
## 6 2020-01-25 Orange California 6059 1 0
```

Subset to only Erie county, New York state

- Subset, method 1

```
row_idx <- (us$county == "Erie") & (us$state == "New York")
erie <- us[row_idx,]
dim(erie)
## [1] 43 6
```

- Subset, method 2

```
erie <- subset(us, (county == "Erie") & (state == "New York"))
dim(erie)
## [1] 43 6
```

Manipulation

- Goal: calculate `new_cases` as the difference between successive days, using `diff()`
- Remember use of `diff()`

```
## example: `diff()` between successive numbers in a vector
x <- c(1, 1, 2, 3, 5, 8, 13)
diff(x)
## [1] 0 1 1 2 3 5
```

- Update, methods 1 & 2 (prepend a 0 when using `diff()`, to get the initial number of new cases)

```
## one way...
erie$new_cases <- diff(c(0, erie$cases))

## ...or another
erie <- within(erie, {
  new_cases <- diff(c(0, cases))
})
```

Summary: calculate maximum (total) number of cases per county in New York state

- For Erie county, let's see how to calculate the maximum (total) number of cases

```
max(erie$cases)      # one way...
## [1] 2954
with(erie, max(cases)) # ... another
## [1] 2954
```

- Subset US data to New York state

```
ny_state <- subset(us, state == "New York")
```

- Summarize each county in the state using `aggregate()`.

- First argument: summarize `cases` grouped by `county ~ cases ~ county`
- Second argument: data source – `ny_state`
- Third argument: function to apply to each subset – `max`

```
max_cases_by_county <- aggregate( cases ~ county, ny_state, max)
head(max_cases_by_county)
##           county cases
## 1      Albany    948
## 2   Allegany     35
## 3     Broome   257
## 4 Cattaraugus    45
## 5     Cayuga    47
## 6 Chautauqua    29
```

- `subset()` to select counties

```
subset(
  max_cases_by_county,
  county %in% c("New York City", "Westchester", "Erie")
)
##           county cases
## 14          Erie  2954
## 29 New York City 158268
## 57 Westchester 27664
```

Summary: calculate maximum (total) number of cases per state

- Use entire data set, `us`
- `aggregate()` cases by county *and* state – `cases ~ county + state`

```
max_cases_by_county_state <-
  aggregate( cases ~ county + state, us, max )
dim(max_cases_by_county_state)
## [1] 2859    3
head(max_cases_by_county_state)
##   county state cases
```

```
## 1 Autauga Alabama    37
## 2 Baldwin Alabama   161
## 3 Barbour Alabama   33
## 4 Bibb Alabama      38
## 5 Blount Alabama    34
## 6 Bullock Alabama   12
```

- aggregate() a second time, using max_cases_by_county_state and aggregating by state

```
max_cases_by_state <-
  aggregate( cases ~ state, max_cases_by_county_state, max )
```

- Explore the data

```
head(max_cases_by_state)
##           state cases
## 1     Alabama    917
## 2     Alaska    168
## 3   Arizona  3359
## 4 Arkansas    727
## 5 California 19528
## 6 Colorado  2583
subset(
  max_cases_by_state,
  state %in% c("California", "Illinois", "New York", "Washington")
)
##           state cases
## 5 California 19528
## 15 Illinois 30574
## 34 New York 158268
## 52 Washington 5914
```

2.5 Day 12 (Friday) Zoom check-in

Review and troubleshoot (20 minutes)

```
## retrieve and clean the current data set
url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read.csv(url, stringsAsFactors = FALSE)
us <- within(us, {
  date = as.Date(date, format = "%Y-%m-%d")
```

```

})

## subset
erie <- subset(us, (county == "Erie") & (state == "New York"))

## manipulate
erie <- within(erie, {
  new_cases <- diff( c(0, cases) )
})

## record of cases to date
erie
##      date county state fips cases deaths new_cases
## 1 2020-03-15   Erie New York 36029     3      0       3
## 2 2020-03-16   Erie New York 36029     6      0       3
## 3 2020-03-17   Erie New York 36029     7      0       1
## 4 2020-03-18   Erie New York 36029     7      0       0
## 5 2020-03-19   Erie New York 36029    28      0      21
## 6 2020-03-20   Erie New York 36029    31      0       3
## 7 2020-03-21   Erie New York 36029    38      0       7
## 8 2020-03-22   Erie New York 36029    54      0      16
## 9 2020-03-23   Erie New York 36029    87      0      33
## 10 2020-03-24   Erie New York 36029   107      0      20
## 11 2020-03-25   Erie New York 36029   122      0      15
## 12 2020-03-26   Erie New York 36029   134      2      12
## 13 2020-03-27   Erie New York 36029   219      6      85
## 14 2020-03-28   Erie New York 36029   354      6     135
## 15 2020-03-29   Erie New York 36029   380      6      26
## 16 2020-03-30   Erie New York 36029   443      8      63
## 17 2020-03-31   Erie New York 36029   438      8      -5
## 18 2020-04-01   Erie New York 36029   553     12     115
## 19 2020-04-02   Erie New York 36029   734     19     181
## 20 2020-04-03   Erie New York 36029   802     22      68
## 21 2020-04-04   Erie New York 36029   945     26     143
## 22 2020-04-05   Erie New York 36029  1059     27     114
## 23 2020-04-06   Erie New York 36029  1163     30     104
## 24 2020-04-07   Erie New York 36029  1163     36      0
## 25 2020-04-08   Erie New York 36029  1205     38      42
## 26 2020-04-09   Erie New York 36029  1362     46     157
## 27 2020-04-10   Erie New York 36029  1409     58      47
## 28 2020-04-11   Erie New York 36029  1472     62      63
## 29 2020-04-12   Erie New York 36029  1571     75      99
## 30 2020-04-13   Erie New York 36029  1624     86      53
## 31 2020-04-14   Erie New York 36029  1668     99      44
## 32 2020-04-15   Erie New York 36029  1751    110      83

```

```

## 63570 2020-04-16 Erie New York 36029 1850 115 99
## 66314 2020-04-17 Erie New York 36029 1929 115 79
## 69072 2020-04-18 Erie New York 36029 1997 115 68
## 71840 2020-04-19 Erie New York 36029 2070 146 73
## 74617 2020-04-20 Erie New York 36029 2109 153 39
## 77398 2020-04-21 Erie New York 36029 2147 161 38
## 80188 2020-04-22 Erie New York 36029 2233 174 86
## 82989 2020-04-23 Erie New York 36029 2450 179 217
## 85793 2020-04-24 Erie New York 36029 2603 184 153
## 88609 2020-04-25 Erie New York 36029 2773 199 170
## 91427 2020-04-26 Erie New York 36029 2954 205 181

## aggregate() cases in each county to find total (max) number
ny_state <- subset(us, state == "New York")
head( aggregate(cases ~ county, ny_state, max) )
##          county cases
## 1      Albany    948
## 2    Allegany     35
## 3    Broome    257
## 4 Cattaraugus     45
## 5    Cayuga     47
## 6 Chautauqua     29

```

User-defined functions

Basic structure:

```

my_function_name <- function(arg1, arg2, ...)
{
  statements

  return(object)
}

```

A concrete example:

```

# declare a function to convert temperatures
toFahrenheit <- function(celsius)
{
  f <- (9/5) * celsius + 32
  return(f)
}

```

```
# invoke the function
temp <- c(20:25)
toFahrenheit(temp)
## [1] 68.0 69.8 71.6 73.4 75.2 77.0
```

Functions can be loaded from a separate file using the `source` command. Enter the temperature conversion function into an *R* script and save as `myFunctions.R`.

```
my_R_funcs <-
  file.path("C:\\Matott\\MyQuarantine", "myFunctions.R")
source(my_R_funcs)

toFahrenheit(c(40, 45, 78, 92, 12, 34))
## [1] 104.0 113.0 172.4 197.6 53.6 93.2
```

Statistical functions in *R*

R has many built-in statistical functions. Some of the more commonly used are listed below:

- `mean()` # average
- `median()` # median (middle value of sorted data)
- `range()` # max - min
- `var()` # variance
- `sd()` # standard deviation
- `summary()` # prints a combination of useful measures

Plotting data

Review of Plot Types

- Pie chart
 - Display proportions of different values for some variable
- Bar plot
 - Display counts of values for categorical variables
- Histogram, density plot
 - Display counts of values for a binned, numeric variable
- Scatter plot

- Display y vs. x
- Box plot
 - Display distributions over different values of a variable

Plotting packages

3 Main Plotting Packages

- Base graphics, lattice, and ggplot2

ggplot2

- The “Cadillac” of plotting packages.
- Part of the “tidyverse”
- Beautiful plots
- To install: `install.packages('ggplot2')`

Words of wisdom on using plotting packages

- A good approach is to learn by doing but don’t start from scratch
- Find an example that is similar in appearance to what you are trying to achieve - many *R* galleries are available on the net.
- When you find something you like, grab the code and modify it to use your own data.
- Fine tune things like labels and fonts at the end, after you are sure you like the way the data is being displayed.

Some example plots

Many of these are based on the Diamonds dataset. Others are based on the `mtcars` dataset and this requires a bit of cleaning in preparation for the corresponding plots:

```
data("mtcars")

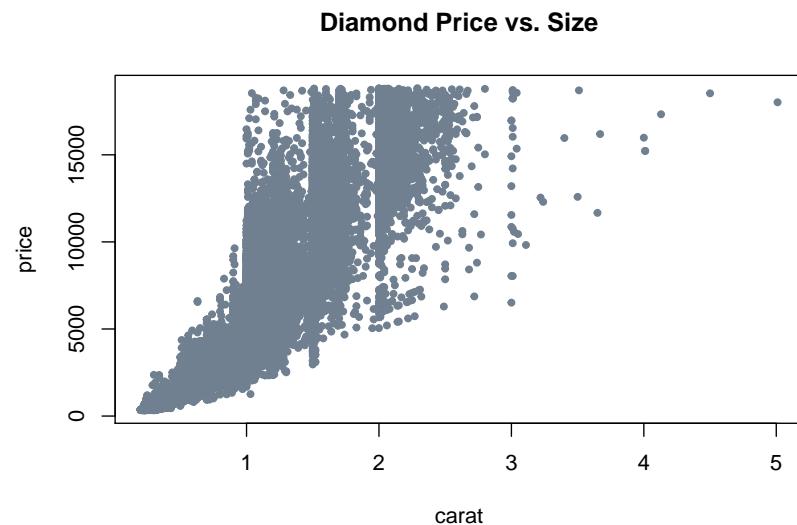
# convert gear to a factor-level object
mtcars$gear = factor(
  mtcars$gear,
  levels = c(3, 4, 5),
  labels = c("3 gears", "4 gears", "5 gears")
)
```

```
# convert cyl to a factor-level object
mtcars$cyl = factor(
  mtcars$cyl,
  levels = c(4, 6, 8),
  labels = c("4 cyl", "6 cyl", "8 cyl")
)
```

- Scatterplot using base graphics

```
library(ggplot2) # for diamonds dataset
data("diamonds")

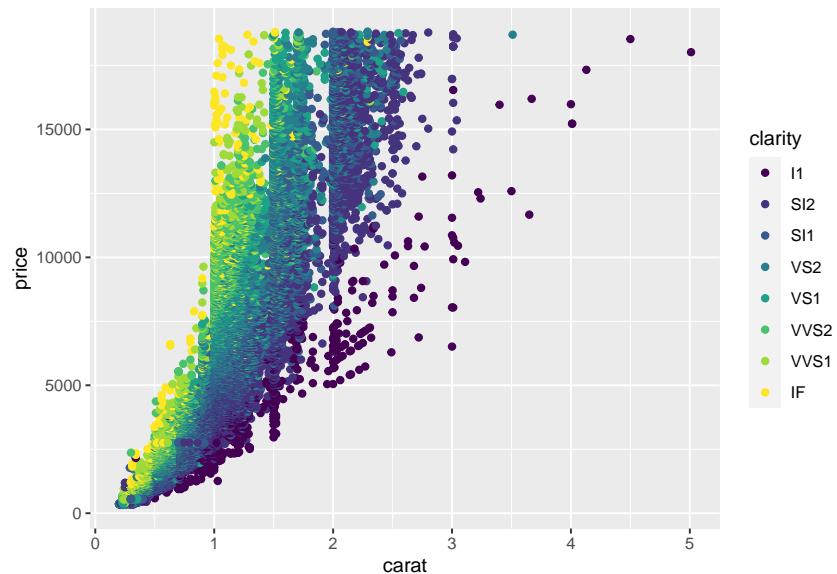
# using base graphics plot() function
plot(formula = price ~ carat, # price vs. carat
      data = diamonds,
      col = "slategray",
      pch = 20,
      main = "Diamond Price vs. Size"
)
```



- Scatterplot using ggplot

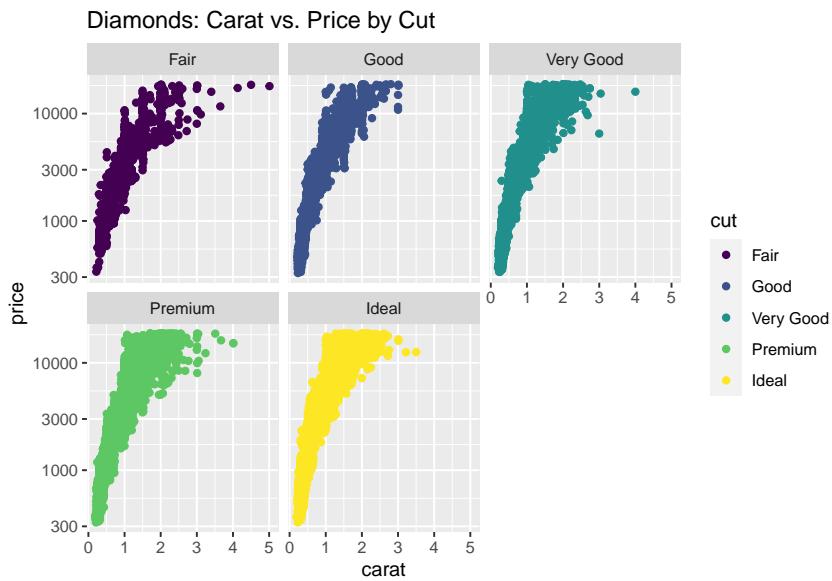
```
library(ggplot2)
data("diamonds")
ggplot(diamonds,
```

```
aes(x=carat, y=price, colour=clarity)) +  
geom_point()
```



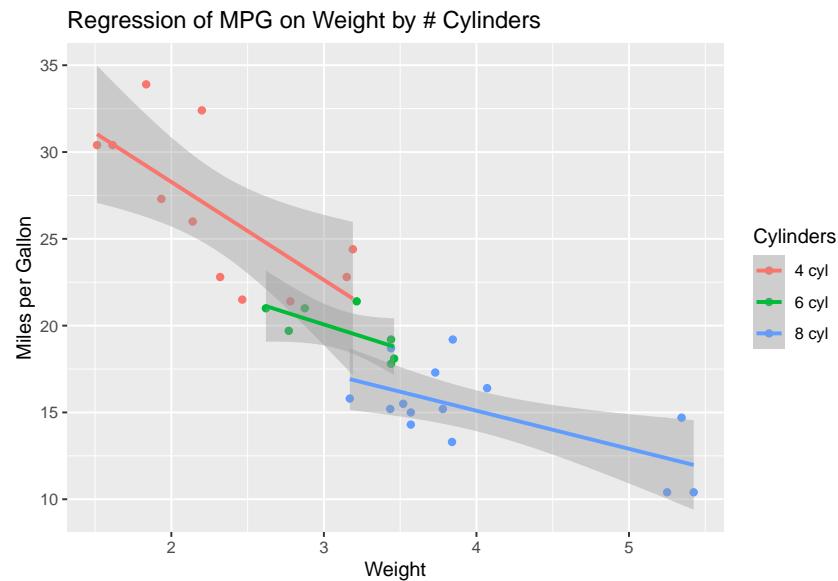
- Scatterplot by category (using ggplot)

```
library(ggplot2)  
data("diamonds")  
ggplot(diamonds) +  
  geom_point(aes(x=carat, y=price, colour=cut)) +  
  scale_y_log10() +  
  facet_wrap(~cut) +  
  ggtitle("Diamonds: Carat vs. Price by Cut")
```



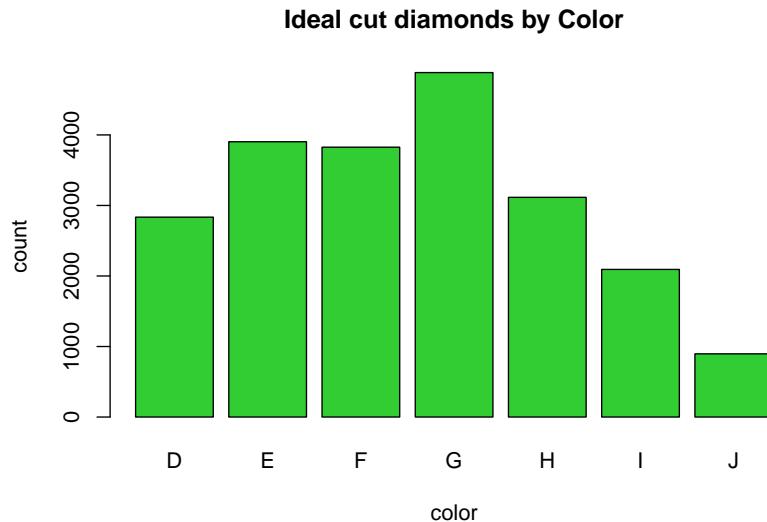
- Scatterplot by category with regression lines (using ggplot2)

```
library(ggplot2)
ggplot(mtcars, aes(wt, mpg, color=cyl))+
  geom_point()+
  geom_smooth(method="lm")+
  labs(title="Regression of MPG on Weight by # Cylinders",
       x = "Weight",
       y = "Miles per Gallon",
       color = "Cylinders")
## `geom_smooth()` using formula 'y ~ x'
```



- Bar Plot using Base Graphics

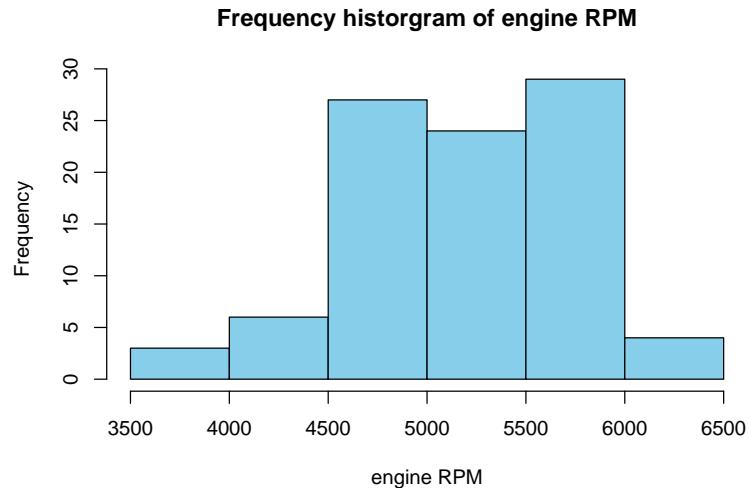
```
library(ggplot2) # for diamonds dataset
data("diamonds")
ideal_cut_colors = diamonds[diamonds$cut == "Ideal", "color"]
# using base graphics barplot() function
barplot(table(ideal_cut_colors),
        xlab = "color",
        ylab = "count",
        main = "Ideal cut diamonds by Color",
        col = "limegreen")
```



- `limegreen` is one of many cool color choices supported by R. See <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf> for a nice 8-page compendium!

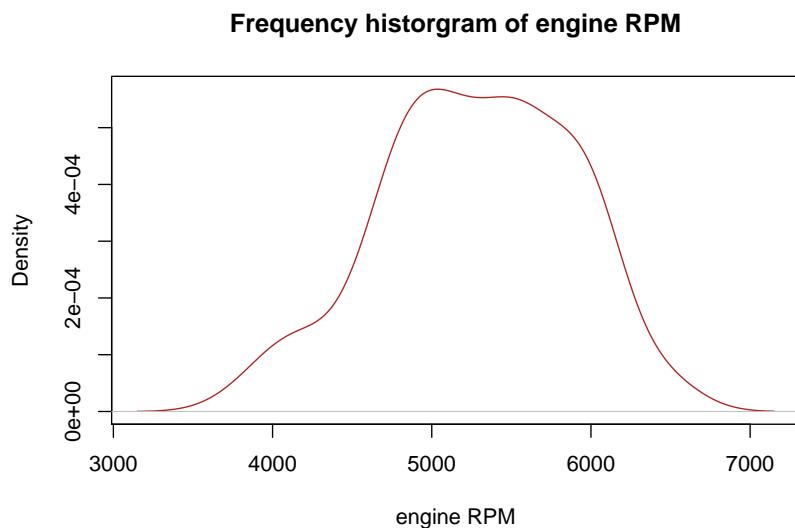
- Frequency histogram using base graphics
- The MASS library provides datasets for Venables and Ripley's MASS textbook

```
library(MASS)
# using the base graphics hist() function
hist(Cars93$RPM,
     xlab = "engine RPM",
     main = "Frequency histogram of engine RPM",
     col = "skyblue")
```



- Density plot using base graphics

```
library(MASS)
plot(density(Cars93$RPM),
      xlab = "engine RPM",
      main = "Frequency histogram of engine RPM",
      col = "firebrick")
```

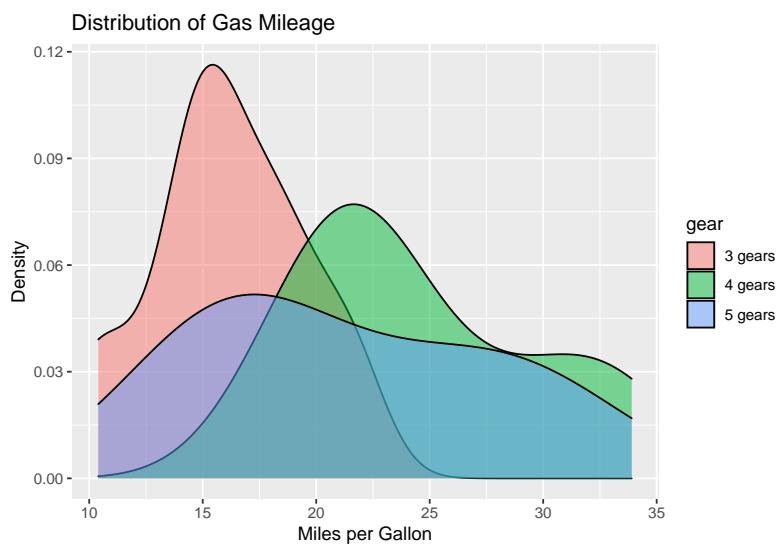


- Density plot using ggplot2

- The `qplot()` function is a “quick plot” wrapper for `ggplot()`.
- It uses an interface that is similar to the `plot()` function of the base graphics package.

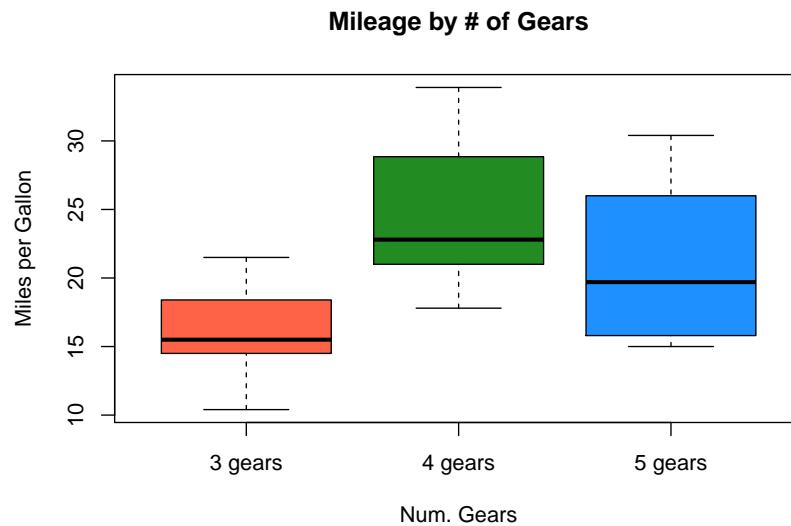
```
library(ggplot2)

qplot(mpg,
      data = mtcars,
      geom = "density",
      fill = gear, alpha=I(0.5),
      main = "Distribution of Gas Mileage",
      xlab = "Miles per Gallon",
      ylab = "Density")
```



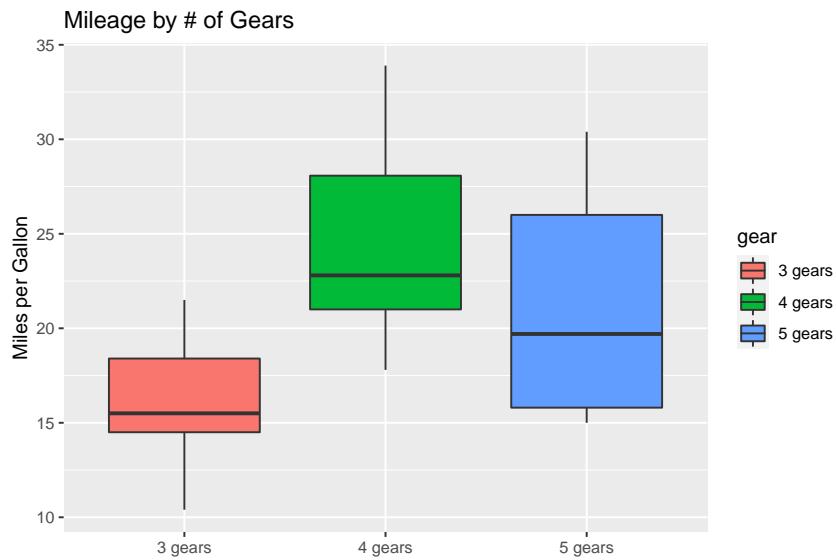
- Box-and-whisker plot using base graphics

```
boxplot(formula = mpg ~ gear,
        data = mtcars,
        main = "Mileage by # of Gears",
        xlab = "Num. Gears",
        ylab = "Miles per Gallon",
        col = c("tomato", "forestgreen", "dodgerblue"))
```



- Box-and-whisker plot using ggplot

```
library(ggplot2)
ggplot(data = mtcars,
       aes(gear, mpg, fill=gear))+
  geom_boxplot()+
  labs(title="Mileage by # of Gears",
       x = "",
       y = "Miles per Gallon")
```



2.6 Day 13: Basic visualization

Let's get the current Erie county data, and create the `new_cases` column

```
## retrieve and clean the current data set
url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read.csv(url, stringsAsFactors = FALSE)
us <- within(us, {
  date = as.Date(date, format = "%Y-%m-%d")
})

## get the Erie county subset
erie <- subset(us, (county == "Erie") & (state == "New York"))

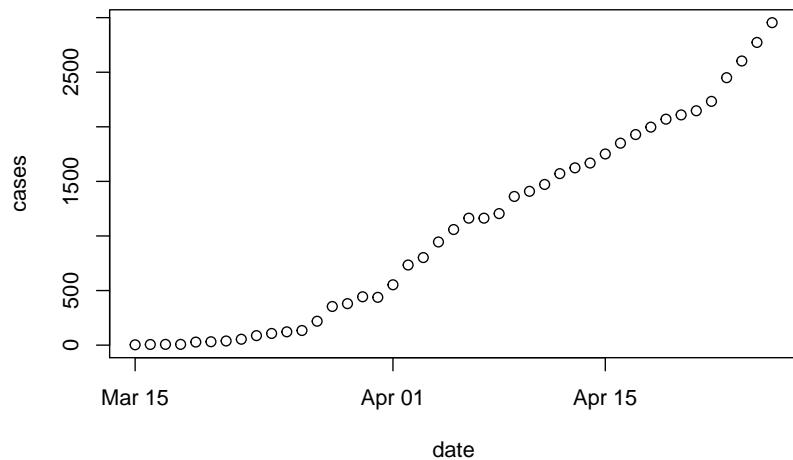
## add the `new_cases` column
erie <- within(erie, {
  new_cases <- diff( c(0, cases) )
})
```

Simple visualization

- We'll use the `plot()` function to create a visualization of the progression of COVID cases in Erie county.
- `plot()` can be used with a `formula`, similar to how we used `aggregate()`.

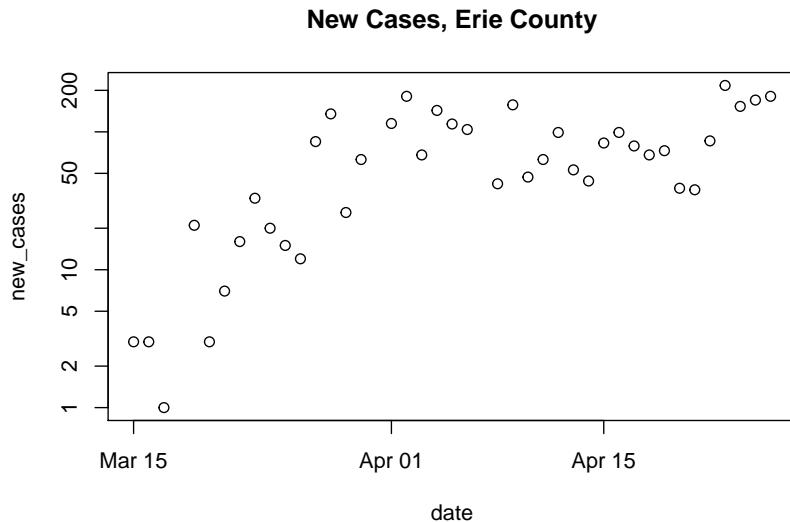
- The `formula` describes the independent (y-axis) variable as a function of the dependent (x-axis) variable
- For our case, the formula will be `cases ~ date`, i.e., plot the number of cases on the y-axis, and date on the x-axis.
- As with `aggregate()`, we need to provide, in the second argument, the `data.frame` where the variables to be plotted can be found.
- Ok, here we go...

```
plot( cases ~ date, erie)
```



- It might be maybe more informative to plot new cases (so that we can see more easily whether social distancing and other measures are having an effect on the spread of COVID cases. Using log-transformed new cases helps to convey the proportional increase

```
plot( new_cases ~ date, erie, log = "y", main = "New Cases, Erie County" )
## Warning in xy.coords(x, y, xlabel, ylabel, log): 3 y values <= 0 omitted from
## logarithmic plot
```



- See `?plot.formula` for some options available when using the formula interface to plot. Additional arguments are described on the help page `?help.default`.

2.7 Day 14: Functions

Yesterday we created a plot for Erie county. The steps to create this plot can be separated into two parts

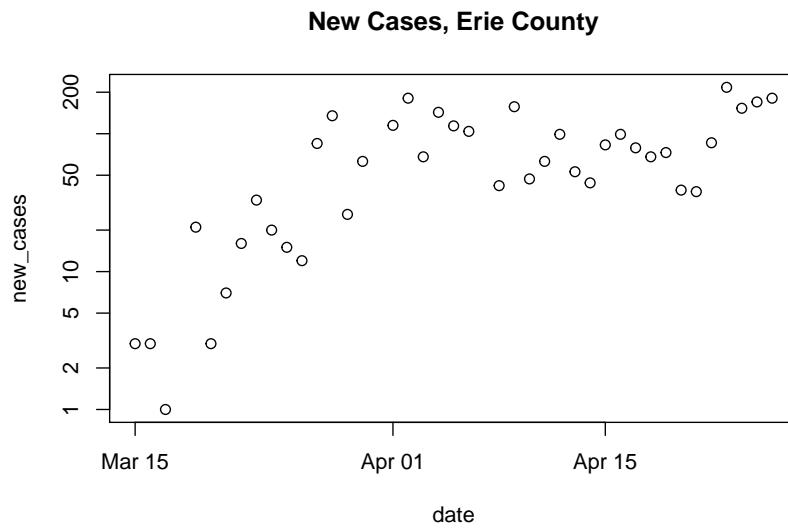
1. Get the full data

```
url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read.csv(url, stringsAsFactors = FALSE)
us <- within(us, {
  date = as.Date(date, format = "%Y-%m-%d")
})
```

2. Subset, update, and plot the data for county of interest

```
erie <- subset(us, (county == "Erie") & (state == "New York"))
erie <- within(erie, {
  new_cases <- diff( c(0, cases) )
})
plot( new_cases ~ date, erie, log = "y", main = "New Cases, Erie County" )
```

```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 3 y values <= 0 omitted from
## logarithmic plot
```



What if we were interested in a different county? We could repeat (cut-and-paste) step 2, updating and generalizing a little

- Define a new variable to indicate the county we are interested in plotting

```
county_of_interest <- "Westchester"
```

- `paste()` concatenates its arguments together into a single character vector. We use this to construct the title of the plot

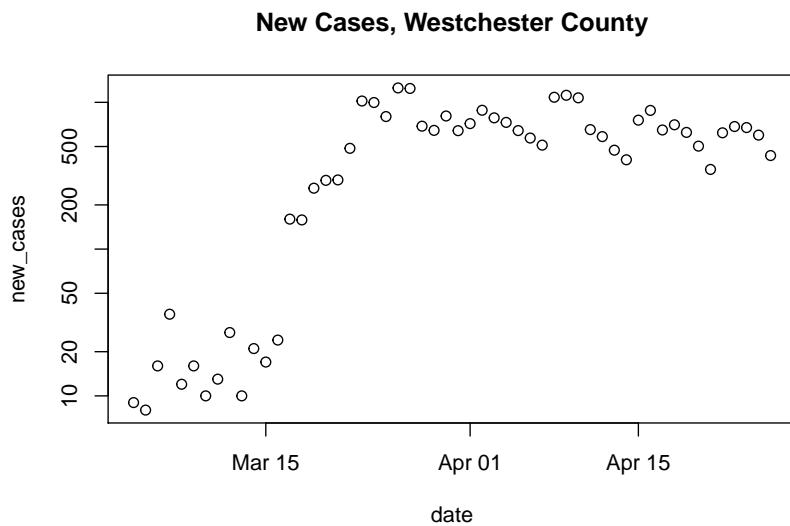
```
main_title <- paste("New Cases,", county_of_interest, "County")
```

- Now create and update a subset of the data for the county that we are interested in

```
county_data <-
subset(us, (county == county_of_interest) & (state == "New York"))
county_data <- within(county_data, {
  new_cases <- diff( c(0, cases) )
})
```

- ... and finally plot the county data

```
plot( new_cases ~ date, county_data, log = "y", main = main_title)
```



- Here is the generalization

```
county_of_interest <- "Westchester"

main_title <- paste("New Cases, ", county_of_interest, "County")
county_data <-
  subset(us_data, (county == county_of_interest) & (state == "New York"))
county_data <- within(county_data, {
  new_cases <- diff( c(0, cases) )
})
plot( new_cases ~ date, county_data, log = "y", main = main_title)
```

It would be tedious and error-prone to copy and paste this code for each county we were interested in.

A better approach is to write a **function** that takes as inputs the **us** data.frame, and the name of the county that we want to plot. Functions are easy to write

- Create a variable to contain the function, use the keyword **function** and then the arguments you want to pass in.

```
plot_county <-
  function(us_data, county_of_interest)
```

- ... then provide the ‘body’ of the function between curly braces

```
{
  main_title <- paste("New Cases,", county_of_interest, "County")
  county_data <-
    subset(us_data, (county == county_of_interest) & (state == "New York"))
  county_data <- within(county_data, {
    new_cases <- diff( c(0, cases) )
  })

  plot( new_cases ~ date, county_data, log = "y", main = main_title)
}
```

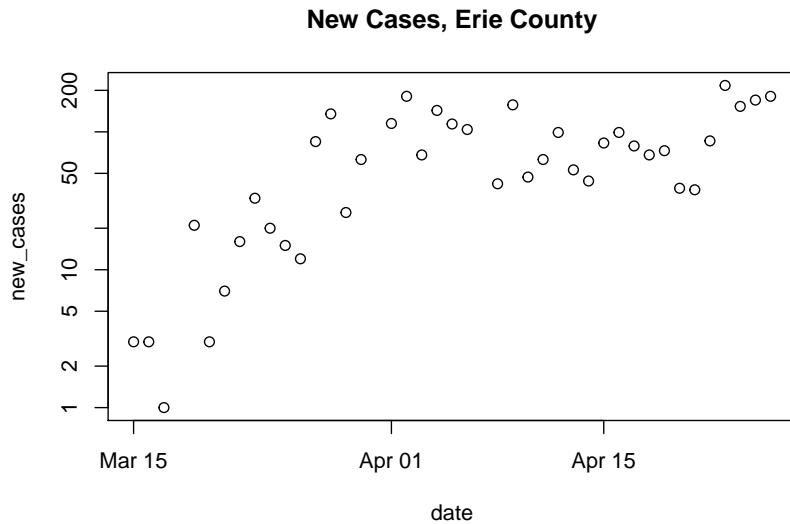
- Normally, the last evaluated line of the code (the `plot()` statement in our example) is returned from the function and can be captured by a variable. In our specific case, `plot()` creates the plot as a *side effect*, and the return value is actually the special symbol `NULL`.
- Here’s the full definition

```
plot_county <-
  function(us_data, county_of_interest)
{
  main_title <- paste("New Cases,", county_of_interest, "County")
  county_data <-
    subset(us_data, (county == county_of_interest) & (state == "New York"))
  county_data <- within(county_data, {
    new_cases <- diff( c(0, cases) )
  })

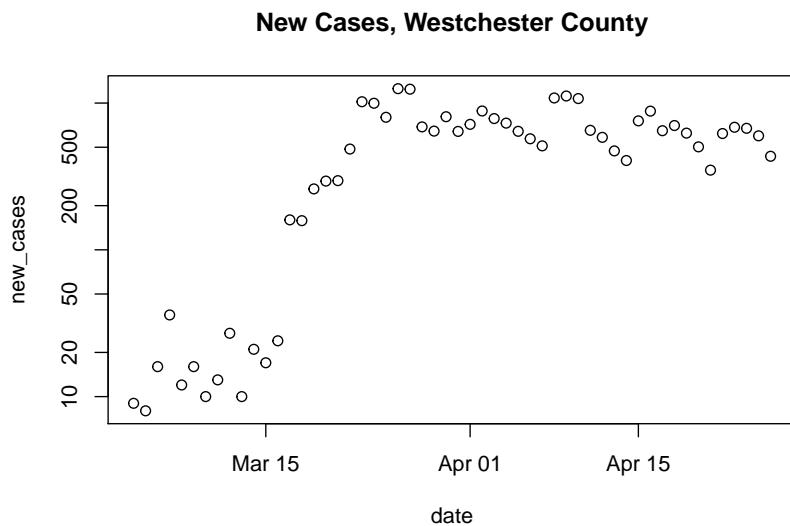
  plot( new_cases ~ date, county_data, log = "y", main = main_title)
}
```

- Run the code defining the function in the *R* console, then use it to plot different counties:

```
plot_county(us, "Erie")
## Warning in xy.coords(x, y, xlabel, ylabel, log): 3 y values <= 0 omitted from
## logarithmic plot
```



```
plot_county(us, "Westchester")
```



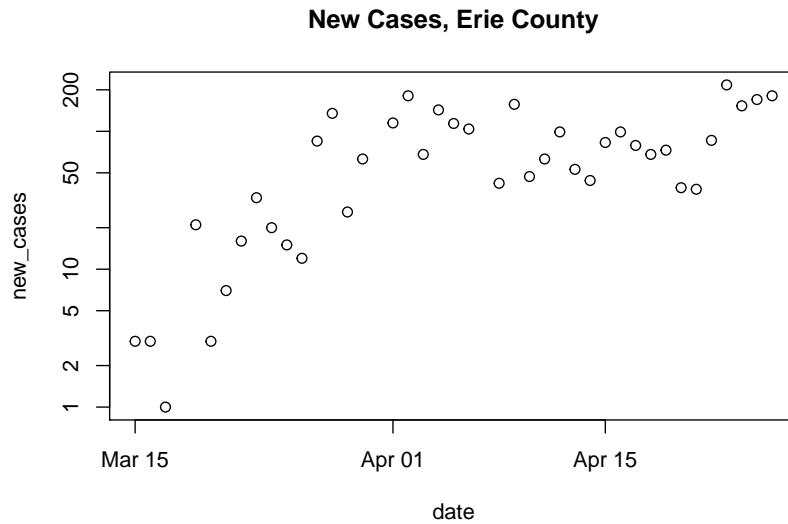
Hmm, come to think of it, we might want to write a simple function to get and clean the US data.

- Get and clean the US data; we don't need any arguments, and the return value (the last line of code evaluated) is the cleaned data

```
get_US_data <-
  function()
{
  url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
  us <- read.csv(url, stringsAsFactors = FALSE)
  within(us, {
    date = as.Date(date, format = "%Y-%m-%d")
  })
}
```

- Verify that it is now just two lines to plot county-level data

```
us <- get_US_data()
plot_county(us, "Erie")
## Warning in xy.coords(x, y, xlabel, ylabel, log): 3 y values <= 0 omitted from
## logarithmic plot
```



- How could you generalize `plot_county()` to plot county-level data for a county in any state? Hint: add a `state` = argument, perhaps using default values

```
plot_county <-
  function(us_data, county = "Erie", state = "New York")
{
  ## your code here!
}
```


Chapter 3

Packages and the ‘tidyverse’

3.1 Day 15 (Monday) Zoom check-in

Review and troubleshoot (15 minutes)

Over the weekend, I wrote two functions. The first retrieves and ‘cleans’ the US data set.

```
get_US_data <-
  function()
{
  ## retrieve data from the internet
  url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
  us <- read.csv(url, stringsAsFactors = FALSE)

  ## update 'date' from character vector to 'Date'. this is the
  ## last line of executed code in the function, so the return
  ## value (the updated 'us' object) is returned by the function
  within(us, {
    date = as.Date(date, format = "%Y-%m-%d")
  })
}
```

The second plots data for a particular county and state

```

plot_county <-
  function(us_data, county_of_interest = "Erie", state_of_interest = "New York")
{
  ## create the title for the plot
  main_title <- paste(
    "New Cases,", county_of_interest, "County", state_of_interest
  )

  ## subset the us data to just the county and state of interest
  county_data <- subset(
    us_data,
    (county == county_of_interest) & (state == state_of_interest)
  )

  ## calculate new cases for particular county and state
  county_data <- within(county_data, {
    new_cases <- diff( c(0, cases) )
  })

  ## plot
  plot( new_cases ~ date, county_data, log = "y", main = main_title)
}

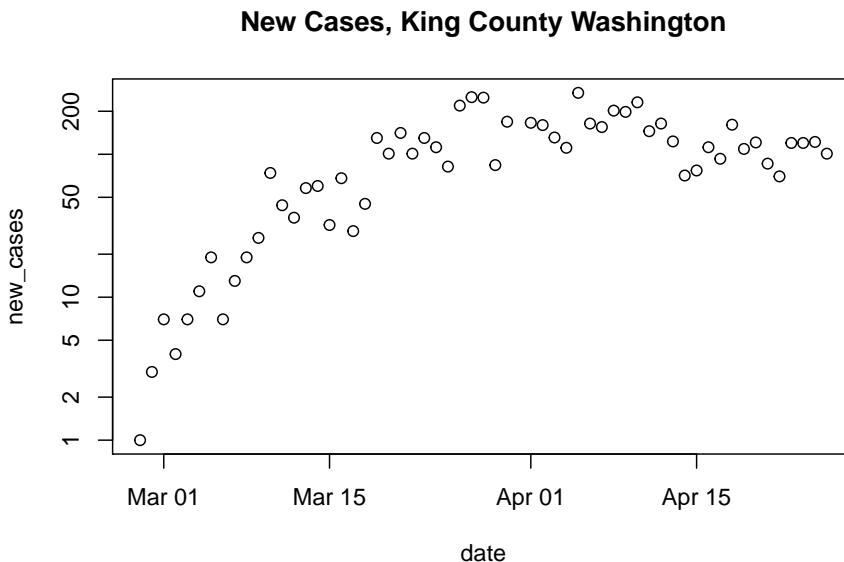
```

I lived in Seattle (King County, Washington), for a while, and this is where the first serious outbreak occurred. Here’s the relevant data:

```

us <- get_US_data()
plot_county(us, "King", "Washington")
## Warning in xy.coords(x, y, xlabel, ylabel, log): 1 y value <= 0 omitted from
## logarithmic plot

```



Packages (20 minutes)

Base *R*

- *R* consists of ‘packages’ that implement different functionality. Each package contains *functions* that we can use, and perhaps data sets (like the *mtcars*) data set from Friday’s presentation) and other resources.
- *R* comes with several ‘base’ packages installed, and these are available in a new *R* session.
- Discover packages that are currently available using the `search()` function. This shows that the ‘stats’, ‘graphics’, ‘grDevices’, ‘utils’, ‘datasets’, ‘methods’, and ‘base’ packages, among others, are available in our current *R* session.

```
> search()
## [1] ".GlobalEnv"      "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"      "package:datasets"
## [7] "package:methods"   "Autoloads"        "package:base"
```

- When we create a variable like

```
x <- c(1, 2, 3)
```

R creates a new *symbol* in the `.GlobalEnv` location on the search path.

- When we evaluate a function like `length(x)`...
 - *R* searches for the function `length()` along the `search()` path. It doesn’t find `length()` in the `.GlobalEnv` (because we didn’t define it there), or in the ‘stats’, ‘graphics’, ... packages. Eventually, *R* finds the definition of `length` in the ‘base’ package.
 - *R* then looks for the definition of `x`, finds it in the `.GlobalEnv`.
 - Finally, *R* applies the definition of `length` found in the base package to the value of `x` found in the `.GlobalEnv`.

Contributed packages

- *R* would be pretty limited if it could only do things that are defined in the base packages.
- It is ‘easy’ to write a package, and to make the package available for others to use.
- A major repository of contributed packages is CRAN – the Comprehensive *R* Archive Network. There are more than 15,000 packages in CRAN.
- Many CRAN packages are arranged in task views that highlight the most useful packages.

Installing and attaching packages

- There are too many packages for all to be distributed with *R*, so it is necessary to *install* contributed packages that you might find interesting.
- once a package is installed (you only need to install a package once), it can be ‘loaded’ and ‘attached’ to the search path using `library()`.
- As an exercise, try to attach the ‘readr’, ‘dplyr’, and ‘ggplot2’ packages

```
library(readr)
library(dplyr)
library(ggplot2)
```

- If any of these fails with a message like

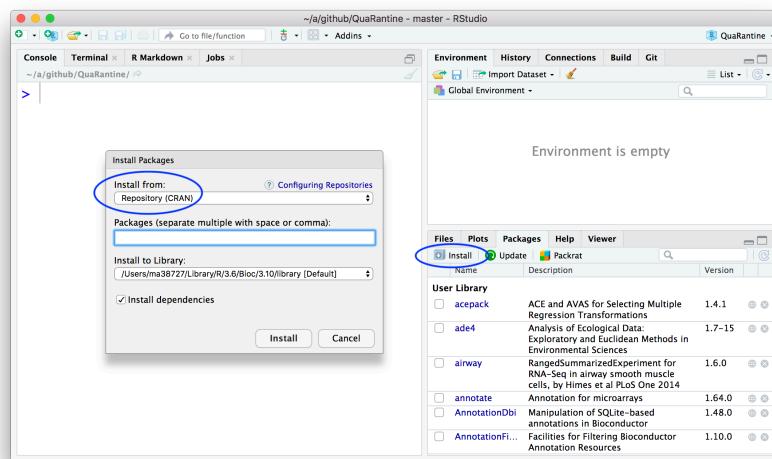
```
library("dplyr")
## Error in library("dplyr") : there is no package called 'dplyr'
```

it means that the package has not been installed (or that you have a typo in the name of the library!)

- Install any package that failed when `library()` was called with

```
install.packages(c("readr", "dplyr"), repos = "https://cran.r-project.org")
```

Alternatively, use the *RStudio* interface to select (in the lower right panel, by default) the ‘Packages’ tab, ‘Install’ button.



- One package may use functions from one or more other packages, so when you install, for instance ‘`dplyr`’, you may actually install *several* packages.

The ‘tidyverse’ of packages (20 minutes)

The ‘tidyverse’ of packages provides a very powerful paradigm for working with data.

- Based on the idea that a first step in data analysis is to transform the data into a standard format. Subsequent steps can then be accomplished in a much more straight-forward way, using a small set of functions.
- Hadley Wickham’s ‘Tidy Data’ paper provides a kind of manifesto for what constitutes tidy data:

1. Each variable forms a column.
 2. Each observation forms a row.
 3. Each type of observational unit forms a table
- We’ll look at the `readr` package for data input, and the `dplyr` package for essential data manipulation.

`readr` for fast data input

- Load (install if necessary!) and attach the `readr` package

```
library(readr)

## if it fails to load, try
##   install.packages("readr", repos = "https://cran.r-project.org")
```

- Example: US COVID data. N.B., `readr::read_csv()` rather than `read.csv()`

```
url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read_csv(url)
## Parsed with column specification:
## cols(
##   date = col_date(format = ""),
##   county = col_character(),
##   state = col_character(),
##   fips = col_character(),
##   cases = col_double(),
##   deaths = col_double()
## )
us
## # A tibble: 92,592 x 6
##   date      county     state    fips  cases  deaths
##   <date>    <chr>     <chr>    <chr> <dbl>  <dbl>
## 1 2020-01-21 Snohomish Washington 53061     1     0
## 2 2020-01-22 Snohomish Washington 53061     1     0
## 3 2020-01-23 Snohomish Washington 53061     1     0
## 4 2020-01-24 Cook       Illinois 17031     1     0
## 5 2020-01-24 Snohomish Washington 53061     1     0
## 6 2020-01-25 Orange    California 06059     1     0
## 7 2020-01-25 Cook       Illinois 17031     1     0
## 8 2020-01-25 Snohomish Washington 53061     1     0
## 9 2020-01-26 Maricopa   Arizona   04013     1     0
## 10 2020-01-26 Los Angeles California 06037    1     0
## # ... with 92,582 more rows
```

- The `us` data is now represented as a `tibble`: a nicer `data.frame`
- Note that
 - `date` has been deduced correctly
 - `read_csv()` does not coerce inputs to `factor` (no need to use `stringsAsFactors = FALSE`)
 - The tibble displays nicely (first ten lines, with an indication of total lines)

dplyr for data manipulation

- Load and attach the `dplyr` package.

```
library(dplyr)
```

- `dplyr` implements a small number of *verbs* for data transformation
 - A small set of functions that allow very rich data transformation
 - All have the same first argument – the `tibble` to be transformed
 - All allow ‘non-standard’ evaluation – use the variable name without quotes “.”.
- `filter()` rows that meet specific criteria

```
filter(us, state == "New York", county == "Erie")
## # A tibble: 43 x 6
##   date     county state    fips cases deaths
##   <date>   <chr>  <chr>   <chr> <dbl>  <dbl>
## 1 2020-03-15 Erie   New York 36029     3     0
## 2 2020-03-16 Erie   New York 36029     6     0
## 3 2020-03-17 Erie   New York 36029     7     0
## 4 2020-03-18 Erie   New York 36029     7     0
## 5 2020-03-19 Erie   New York 36029    28     0
## 6 2020-03-20 Erie   New York 36029    31     0
## 7 2020-03-21 Erie   New York 36029    38     0
## 8 2020-03-22 Erie   New York 36029    54     0
## 9 2020-03-23 Erie   New York 36029    87     0
## 10 2020-03-24 Erie   New York 36029   107     0
## # ... with 33 more rows
```

- `dplyr` uses the ‘pipe’ `%>%` as a way to chain data and functions together

```
us %>%
  filter(state == "New York", county == "Erie")
## # A tibble: 43 x 6
##   date      county state    fips cases deaths
##   <date>    <chr>  <chr>    <chr> <dbl>  <dbl>
## 1 2020-03-15 Erie   New York 36029     3     0
## 2 2020-03-16 Erie   New York 36029     6     0
## 3 2020-03-17 Erie   New York 36029     7     0
## 4 2020-03-18 Erie   New York 36029     7     0
## 5 2020-03-19 Erie   New York 36029    28     0
## 6 2020-03-20 Erie   New York 36029    31     0
## 7 2020-03-21 Erie   New York 36029    38     0
## 8 2020-03-22 Erie   New York 36029    54     0
## 9 2020-03-23 Erie   New York 36029    87     0
## 10 2020-03-24 Erie   New York 36029   107     0
## # ... with 33 more rows
```

- The pipe works by transforming whatever is on the left-hand side of the `%>%` to the first argument of the function on the right-hand side.
- Like `filter()`, most dplyr functions take as their first argument a tibble, and return a tibble. So the functions can be chained together, as in the following example.

- `select()` specific columns

```
us %>%
  filter(state == "New York", county == "Erie") %>%
  select(state, county, date, cases)
## # A tibble: 43 x 4
##   state      county date      cases
##   <chr>    <chr>  <date>    <dbl>
## 1 New York Erie   2020-03-15     3
## 2 New York Erie   2020-03-16     6
## 3 New York Erie   2020-03-17     7
## 4 New York Erie   2020-03-18     7
## 5 New York Erie   2020-03-19    28
## 6 New York Erie   2020-03-20    31
## 7 New York Erie   2020-03-21    38
## 8 New York Erie   2020-03-22    54
## 9 New York Erie   2020-03-23    87
## 10 New York Erie  2020-03-24   107
## # ... with 33 more rows
```

Other common verbs (see tomorrow’s quarantine)

- `mutate()` (add or update) columns
- `summarize()` one or more columns
- `group_by()` one or more variables when performing computations. `ungroup()` removes the grouping.
- `arrange()` rows based on values in particular column(s); `desc()` in descending order.
- `count()` the number of times values occur

Other ‘tidyverse’ packages

- Packages adopting the ‘tidy’ approach to data representation and management are sometimes referred to as the tidyverse.
- `ggplot2` implements high-quality data visualization in a way consistent with tidy data representations.
- The `tidyverse` package implements functions that help to transform data to ‘tidy’ format; we’ll use `pivot_longer()` later in the week.

3.2 Day 16 Key tidyverse packages: `readr` and `dplyr`

Start a script for today. In the script

- Load the libraries that we will use

```
library(readr)
library(dplyr)
```

- If *R* responds with (similarly for `dplyr`)

```
Error in library(readr) : there is no package called 'readr'
```

then you’ll need to install (just once per *R* installation) the `readr` pacakge

```
install.packages("readr", repos = "https://cran.r-project.org")
```

Work through the following commands, adding appropriate lines to your script

- Read US COVID data. N.B., `readr::read_csv()` rather than `read.csv()`

```

url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read_csv(url)
## Parsed with column specification:
## cols(
##   date = col_date(format = ""),
##   county = col_character(),
##   state = col_character(),
##   fips = col_character(),
##   cases = col_double(),
##   deaths = col_double()
## )
us
## # A tibble: 92,592 x 6
##   date       county     state     fips   cases   deaths
##   <date>     <chr>     <chr>     <chr>   <dbl>   <dbl>
## 1 2020-01-21 Snohomish Washington 53061      1      0
## 2 2020-01-22 Snohomish Washington 53061      1      0
## 3 2020-01-23 Snohomish Washington 53061      1      0
## 4 2020-01-24 Cook        Illinois 17031      1      0
## 5 2020-01-24 Snohomish Washington 53061      1      0
## 6 2020-01-25 Orange      California 06059      1      0
## 7 2020-01-25 Cook        Illinois 17031      1      0
## 8 2020-01-25 Snohomish Washington 53061      1      0
## 9 2020-01-26 Maricopa    Arizona  04013      1      0
## 10 2020-01-26 Los Angeles California 06037     1      0
## # ... with 92,582 more rows

```

- `filter()` rows that meet specific criteria

```

us %>%
  filter(state == "New York", county == "Erie")
## # A tibble: 43 x 6
##   date       county state     fips   cases   deaths
##   <date>     <chr> <chr>     <chr>   <dbl>   <dbl>
## 1 2020-03-15 Erie    New York 36029      3      0
## 2 2020-03-16 Erie    New York 36029      6      0
## 3 2020-03-17 Erie    New York 36029      7      0
## 4 2020-03-18 Erie    New York 36029      7      0
## 5 2020-03-19 Erie    New York 36029     28      0
## 6 2020-03-20 Erie    New York 36029     31      0
## 7 2020-03-21 Erie    New York 36029     38      0
## 8 2020-03-22 Erie    New York 36029     54      0
## 9 2020-03-23 Erie    New York 36029     87      0
## 10 2020-03-24 Erie   New York 36029    107      0
## # ... with 33 more rows

```

- `select()` specific columns

```
us %>%
  filter(state == "New York", county == "Erie") %>%
  select(state, county, date, cases)
## # A tibble: 43 x 4
##   state    county date      cases
##   <chr>    <chr> <date>    <dbl>
## 1 New York Erie  2020-03-15     3
## 2 New York Erie  2020-03-16     6
## 3 New York Erie  2020-03-17     7
## 4 New York Erie  2020-03-18     7
## 5 New York Erie  2020-03-19    28
## 6 New York Erie  2020-03-20    31
## 7 New York Erie  2020-03-21    38
## 8 New York Erie  2020-03-22    54
## 9 New York Erie  2020-03-23   87
## 10 New York Erie 2020-03-24  107
## # ... with 33 more rows
```

- `mutate()` (add or update) columns

```
erie <-
  us %>%
  filter(state == "New York", county == "Erie")
erie %>%
  mutate(new_cases = diff(c(0, cases)))
## # A tibble: 43 x 7
##   date      county state    fips  cases deaths new_cases
##   <date>    <chr> <chr>  <chr> <dbl> <dbl>    <dbl>
## 1 2020-03-15 Erie  New York 36029     3     0       3
## 2 2020-03-16 Erie  New York 36029     6     0       3
## 3 2020-03-17 Erie  New York 36029     7     0       1
## 4 2020-03-18 Erie  New York 36029     7     0       0
## 5 2020-03-19 Erie  New York 36029    28     0      21
## 6 2020-03-20 Erie  New York 36029    31     0       3
## 7 2020-03-21 Erie  New York 36029    38     0       7
## 8 2020-03-22 Erie  New York 36029    54     0      16
## 9 2020-03-23 Erie  New York 36029    87     0      33
## 10 2020-03-24 Erie  New York 36029   107     0      20
## # ... with 33 more rows
```

- `summarize()` one or more columns

```

erie %>%
  mutate(new_cases = diff(c(0, cases))) %>%
  summarize(
    duration = n(),
    total_cases = max(cases),
    max_new_cases_per_day = max(new_cases),
    mean_new_cases_per_day = mean(new_cases),
    median_new_cases_per_day = median(new_cases)
  )
## # A tibble: 1 x 5
##   duration total_cases max_new_cases_per~ mean_new_cases_per~ median_new_cases_
##       <int>        <dbl>             <dbl>             <dbl>
## 1        43        2954            217             68.7

```

- `group_by()` one or more variables when performing computations

```

us_county_cases <-
  us %>%
  group_by(county, state) %>%
  summarize(total_cases = max(cases))

us_state_cases <-
  us_county_cases %>%
  group_by(state) %>%
  summarize(total_cases = sum(total_cases))

```

- `arrange()` based on a particular column; `desc()` in descending order.

```

us_county_cases %>%
  arrange(desc(total_cases))
## # A tibble: 2,859 x 3
## # Groups:   county [1,676]
##   county           state     total_cases
##   <chr>            <chr>      <dbl>
## 1 New York City  New York    158268
## 2 Nassau          New York    34522
## 3 Suffolk          New York    32059
## 4 Cook             Illinois   30574
## 5 Westchester      New York    27664
## 6 Los Angeles      California  19528
## 7 Wayne            Michigan    15748
## 8 Bergen           New Jersey  14965
## 9 Hudson           New Jersey  13708
## 10 Essex            New Jersey  12863
## # ... with 2,849 more rows

```

```
us_state_cases %>%
  arrange(desc(total_cases))
## # A tibble: 55 x 2
##   state      total_cases
##   <chr>        <dbl>
## 1 New York     290167
## 2 New Jersey    113223
## 3 Massachusetts  55060
## 4 Illinois      43916
## 5 California     43696
## 6 Pennsylvania   42717
## 7 Michigan       37761
## 8 Florida        31535
## 9 Louisiana      26782
## 10 Connecticut   25284
## # ... with 45 more rows
```

- count() the number of times values occur (duration of the pandemic?)

```
us %>%
  count(county, state) %>%
  arrange(desc(n))
## # A tibble: 2,859 x 3
##   county      state     n
##   <chr>        <chr>  <int>
## 1 Snohomish    Washington 97
## 2 Cook         Illinois  94
## 3 Orange        California 93
## 4 Los Angeles   California 92
## 5 Maricopa      Arizona  92
## 6 Santa Clara   California 87
## 7 Suffolk        Massachusetts 86
## 8 San Francisco California 85
## 9 Dane          Wisconsin 82
## 10 San Diego     California 77
## # ... with 2,849 more rows
```

3.3 Day 17 Visualization with ggplot2

Setup

Load packages we'll use today

```
library(readr)
library(dplyr)
library(ggplot2)
library(tidyr)
```

Remember that packages need to be installed before loading; if you see...

```
> library(ggplot2)
## Error in library(ggplot2) : there is no package called 'ggplot2'
```

...then you’ll need to install the package and try again

```
install.packages("ggplot2", repos = "https://cran.r-project.org")
library(ggplot2)
```

Input data using `readr::read_csv()`

```
url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"
us <- read_csv(url)
## Parsed with column specification:
## cols(
##   date = col_date(format = ""),
##   county = col_character(),
##   state = col_character(),
##   fips = col_character(),
##   cases = col_double(),
##   deaths = col_double()
## )
```

Create the Erie county subset, with columns `new_cases` and `new_deaths`

```
erie <-
  us %>%
  filter(county == "Erie", state == "New York") %>%
  mutate(
    new_cases = diff(c(0, cases)),
    new_deaths = diff(c(0, deaths))
  )
```

ggplot2 essentials

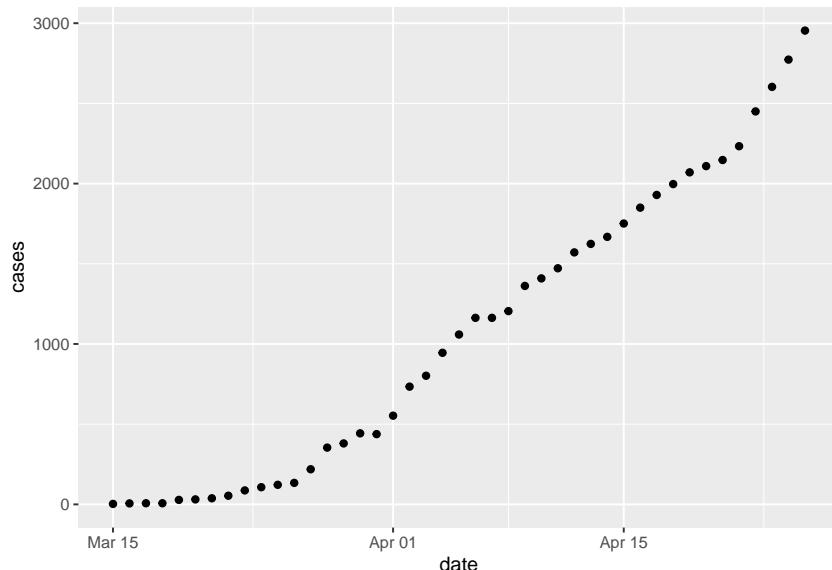
The ‘gg’ in ggplot2

- ‘Grammar of Graphics’ – a formal, scholarly system for describing and creating graphics.
- See the usage guide, and the data visualization chapter of R for Data Science.
- The reference section of the usage guide provides a good entry point

A first plot

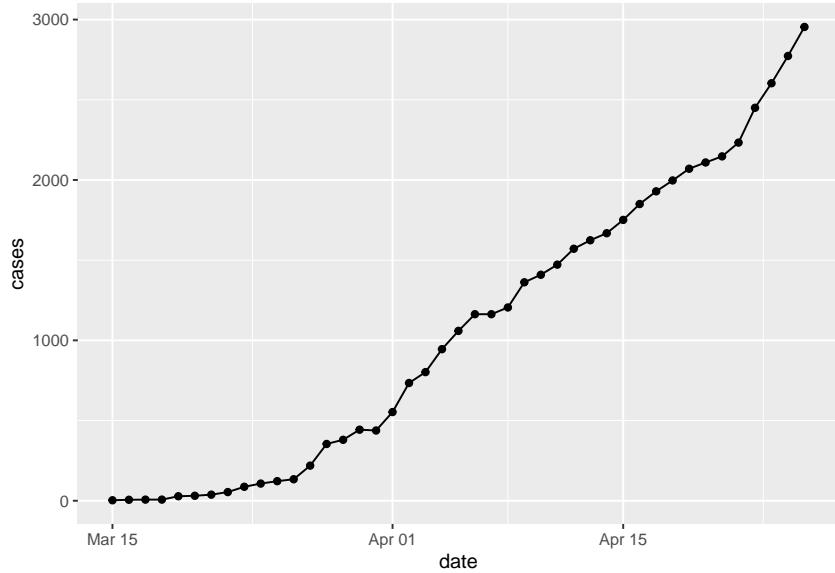
- Specify the data to use. Do this by (a) providing the tibble containing the data (`erie`) and (b) communicating the ‘aesthetics’ of the overall graph by specifying the `x` and `y` data columns – `ggplot(erie, aes(x = date, y = new_cases))`
- Add a `geom_` describing the geometric object used to represent the data, e.g., use `geom_point()` to represent the data as points.

```
ggplot(erie, aes(date, cases)) +
  geom_point()
```



- Note that the plot is assembled by adding elements using a simple `+`. Connect the points with `geom_line()`.

```
ggplot(erie, aes(date, cases)) +
  geom_point() +
  geom_line()
```

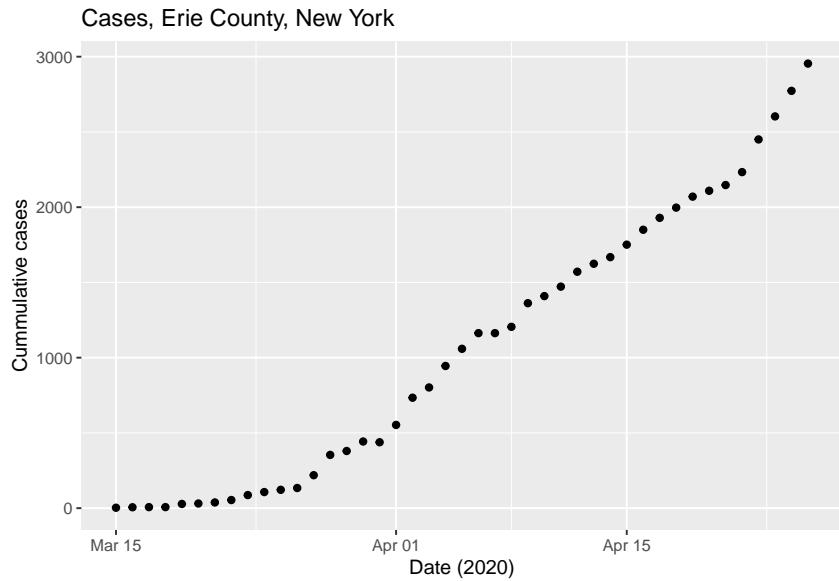


- Plots can actually be captured in a variable, e.g., p

```
p <- ggplot(erie, aes(date, cases)) +
  geom_point()
```

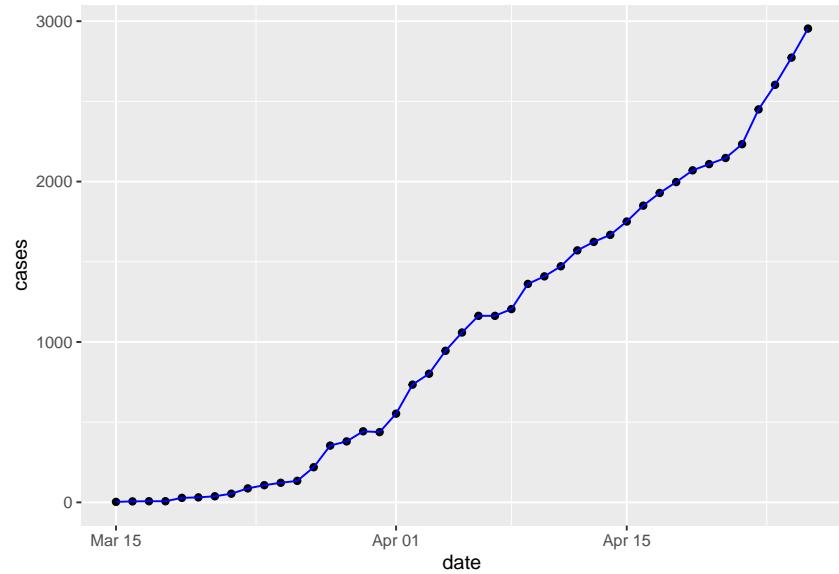
... and then updated and displayed

```
p +
  xlab("Date (2020)") +
  ylab("Cumulative cases") +
  ggtitle("Cases, Erie County, New York")
```



- Arguments to each `geom` influence how the geometry is displayed, e.g.,

```
p +  
  geom_line(color = "blue")
```



COVID-19 in Erie county

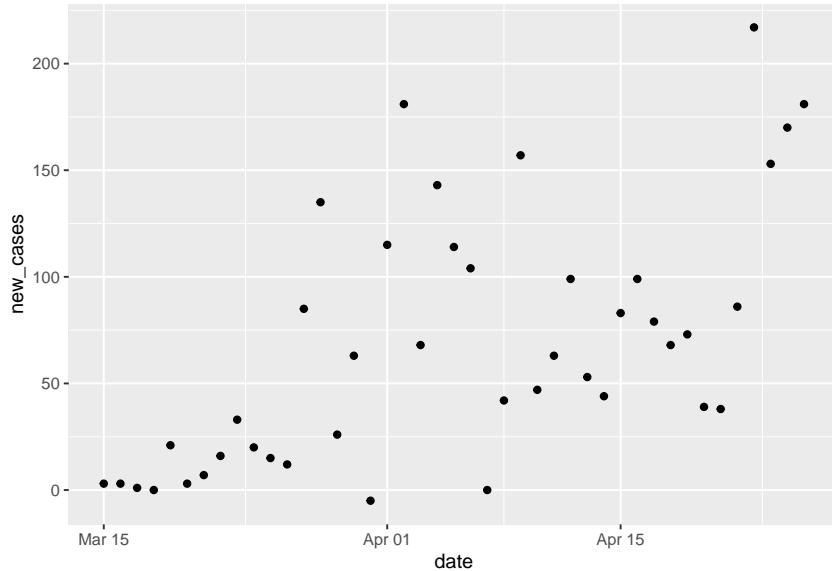
New cases

- Create a base plot using `new_cases`)

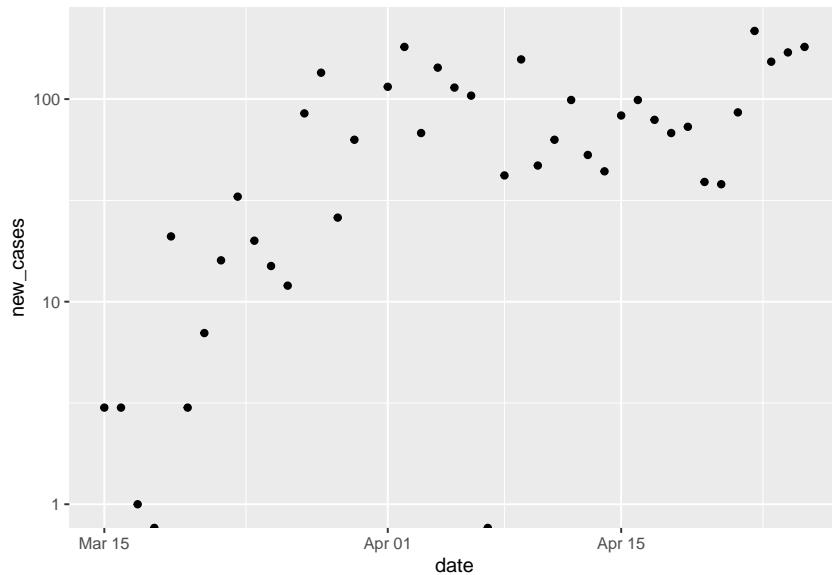
```
p <- ggplot(erie, aes(date, new_cases)) +
  geom_point()
```

- Visualize on a linear and a log-transformed y-axis

```
p # linear
```

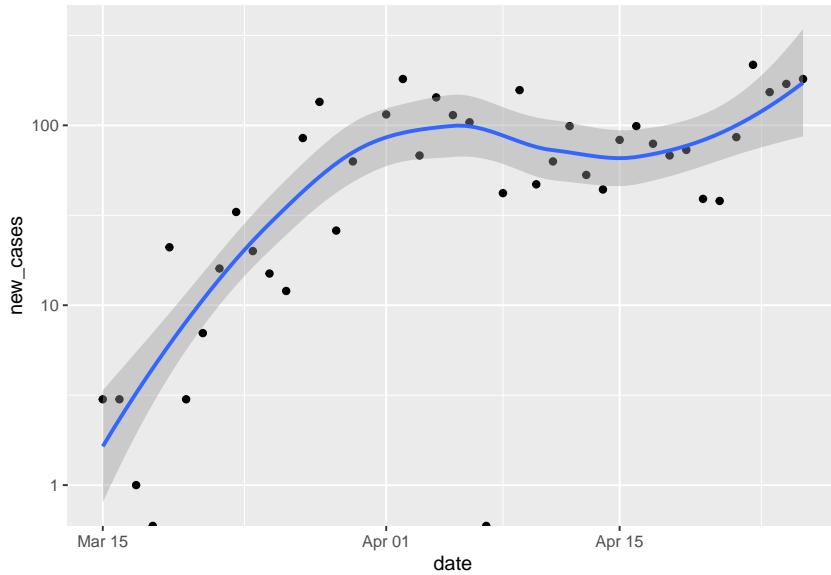


```
p + scale_y_log10()
## Warning in self$trans$transform(x): NaNs produced
## Warning: Transformation introduced infinite values in continuous y-axis
## Warning: Removed 1 rows containing missing values (geom_point).
```



Add a smoothed line to the plot. By default the smoothed line is a local regression appropriate for exploratory data analysis. Note the confidence bands displayed in the plot, and how they convey a measure of certainty about the fit.

```
p +
  scale_y_log10() +
  geom_smooth()
## Warning in self$trans$transform(x): NaNs produced
## Warning: Transformation introduced infinite values in continuous y-axis
## Warning in self$trans$transform(x): NaNs produced
## Warning: Transformation introduced infinite values in continuous y-axis
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
## Warning: Removed 3 rows containing non-finite values (stat_smooth).
## Warning: Removed 1 rows containing missing values (geom_point).
```

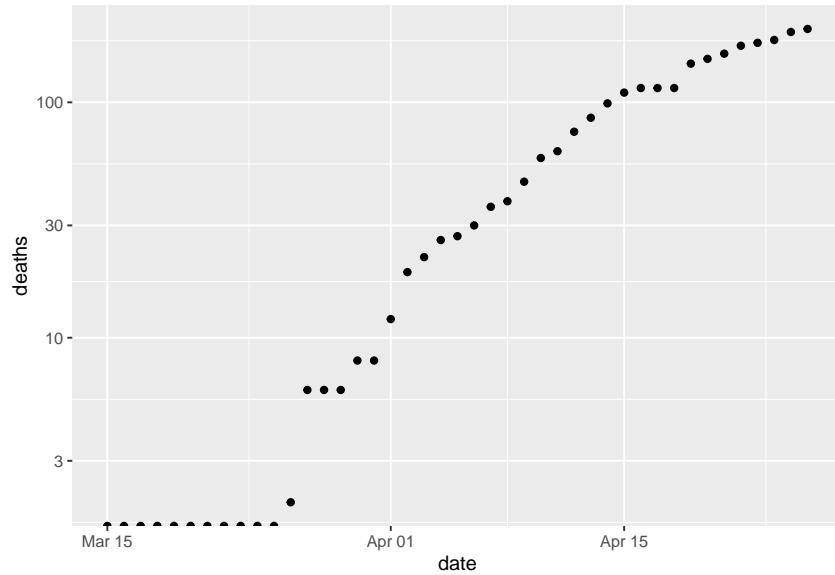


- Reflect on the presentation of data, especially how log-transformation and a clarifies our impression of the local progress of the pandemic.
- The local regression used by `geom_smooth()` can be replaced by a linear regressin with `geom_smooth(method = "lm")`. Create this plot and reflect on the assumptions and suitability of a linear model for this data.

New cases and mortality

- It’s easy to separately plot `deaths` by updating the aesthetic in `ggplot()`

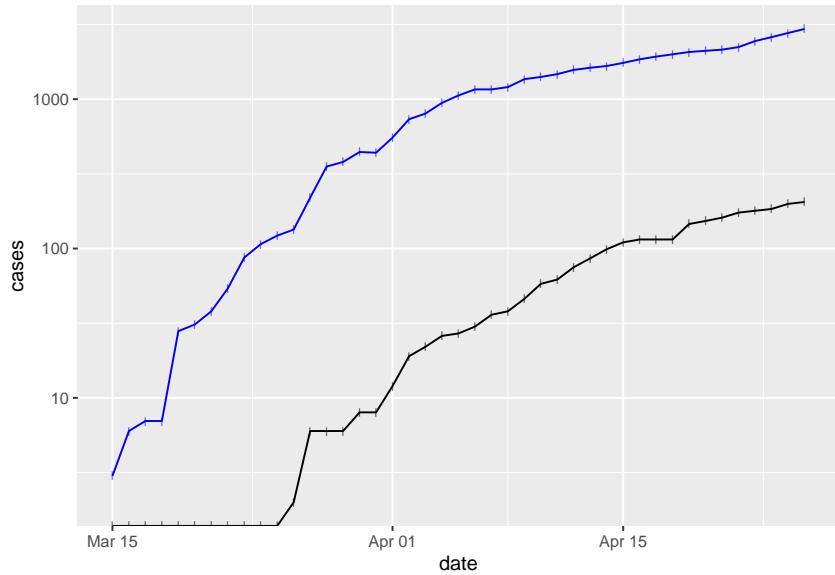
```
ggplot(erie, aes(date, deaths)) +
  scale_y_log10() +
  geom_point()
## Warning: Transformation introduced infinite values in continuous y-axis
```



- What about plotting cases and deaths? Move the `aes()` argument to the individual geometries. Use different colors for each geometry

```
ggplot(erie) +
  scale_y_log10() +
  geom_point(aes(date, cases), shape = "|", color = "blue") +
  geom_line(aes(date, cases), color = "blue") +
  geom_point(aes(date, deaths), shape = "|") +
  geom_line(aes(date, deaths))
## Warning: Transformation introduced infinite values in continuous y-axis

## Warning: Transformation introduced infinite values in continuous y-axis
```



Deaths lag behind cases by a week or so.

‘Long’ data and an alternative approach to plotting multiple curves.

- Let’s simplify the data to just the columns of interest for this exercise

```
simple <-
  erie %>%
  select(date, cases, deaths)
simple
## # A tibble: 43 x 3
##   date      cases  deaths
##   <date>    <dbl>   <dbl>
## 1 2020-03-15     3     0
## 2 2020-03-16     6     0
## 3 2020-03-17     7     0
## 4 2020-03-18     7     0
## 5 2020-03-19    28     0
## 6 2020-03-20    31     0
## 7 2020-03-21    38     0
## 8 2020-03-22    54     0
## 9 2020-03-23   87     0
## 10 2020-03-24  107     0
## # ... with 33 more rows
```

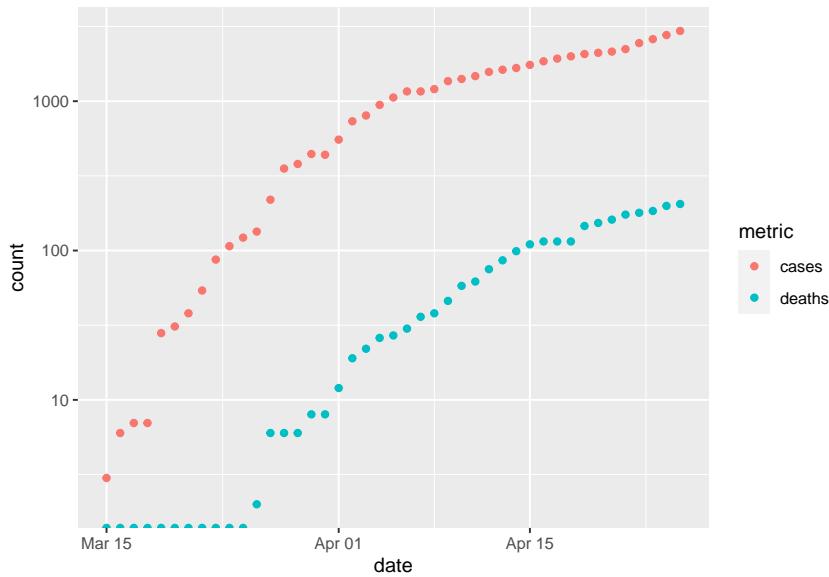
- Use `tidyverse::pivot_longer()` to transform the two columns ‘cases’ and ‘deaths’ into a column that indicates ‘name’ and ‘value’; ‘name’ is ‘cases’

when the corresponding ‘value’ came from the ‘cases’ column, and similarly for ‘deaths’. See the help page `?tidy::pivot_longer` and tomorrow’s exercises for more on `pivot_longer()`.

```
longer <-
  simple %>%
  pivot_longer(
    c("cases", "deaths"),
    names_to = "metric",
    values_to = "count"
  )
longer
## # A tibble: 86 x 3
##   date      metric count
##   <date>    <chr>  <dbl>
## 1 2020-03-15 cases     3
## 2 2020-03-15 deaths    0
## 3 2020-03-16 cases     6
## 4 2020-03-16 deaths    0
## 5 2020-03-17 cases     7
## 6 2020-03-17 deaths    0
## 7 2020-03-18 cases     7
## 8 2020-03-18 deaths    0
## 9 2020-03-19 cases    28
## 10 2020-03-19 deaths    0
## # ... with 76 more rows
```

- Plot date and value, coloring points by name^c

```
ggplot(longer, aes(date, count, color = metric)) +
  scale_y_log10() +
  geom_point()
## Warning: Transformation introduced infinite values in continuous y-axis
```



COVID-19 in New York State

We’ll explore ‘facet’ visualizations, which create a panel of related plots

Setup

- From the US data, extract Erie and Westchester counties and New York City. Use `coi` (‘counties of interest’) as a variable to hold this data

```
coi <-
  us %>%
  filter(
    county %in% c("Erie", "Westchester", "New York City"),
    state == "New York"
  ) %>%
  select(date, county, cases, deaths)
coi
## # A tibble: 154 x 4
##   date       county     cases  deaths
##   <date>     <chr>     <dbl>   <dbl>
## 1 2020-03-01 New York City     1      0
## 2 2020-03-02 New York City     1      0
## 3 2020-03-03 New York City     2      0
## 4 2020-03-04 New York City     2      0
## 5 2020-03-04 Westchester      9      0
```

```

## 6 2020-03-05 New York City      4      0
## 7 2020-03-05 Westchester      17      0
## 8 2020-03-06 New York City      5      0
## 9 2020-03-06 Westchester      33      0
## 10 2020-03-07 New York City     12      0
## # ... with 144 more rows

```

- Pivot cases and deaths into long form

```

coi_longer <-
  coi %>%
  pivot_longer(
    c("cases", "deaths"),
    names_to = "metric",
    values_to = "count"
  )
coi_longer
## # A tibble: 308 x 4
##   date       county     metric count
##   <date>     <chr>      <chr>  <dbl>
## 1 2020-03-01 New York City cases     1
## 2 2020-03-01 New York City deaths    0
## 3 2020-03-02 New York City cases     1
## 4 2020-03-02 New York City deaths    0
## 5 2020-03-03 New York City cases     2
## 6 2020-03-03 New York City deaths    0
## 7 2020-03-04 New York City cases     2
## 8 2020-03-04 New York City deaths    0
## 9 2020-03-04 Westchester cases      9
## 10 2020-03-04 Westchester deaths     0
## # ... with 298 more rows

```

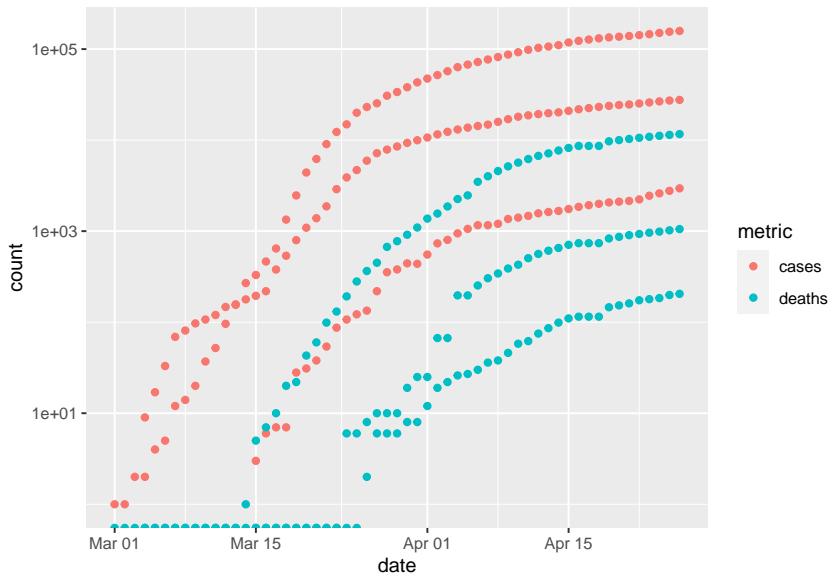
Visualization

- We can plot cases and deaths of each county...

```

p <-
  ggplot(coi_longer, aes(date, count, color = metric)) +
  scale_y_log10() +
  geom_point()
p
## Warning: Transformation introduced infinite values in continuous y-axis

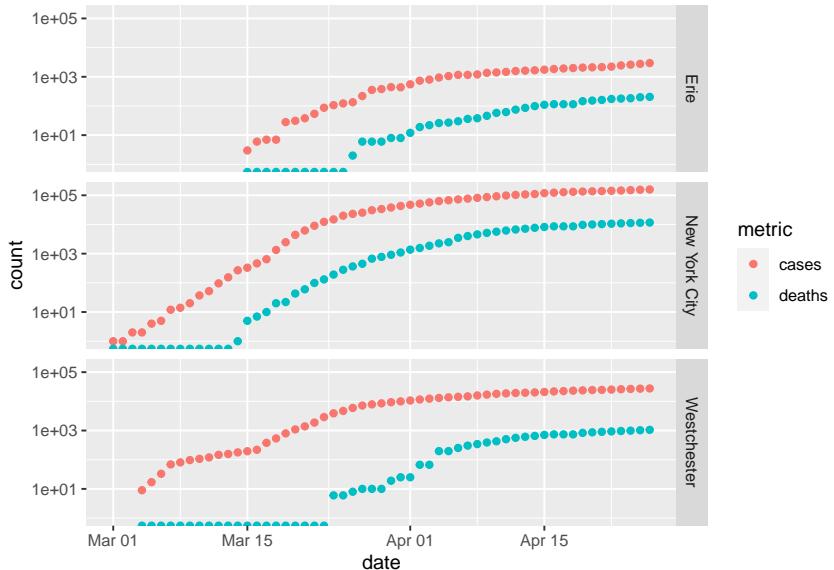
```



... but this is too confusing.

- Separate each county into a facet

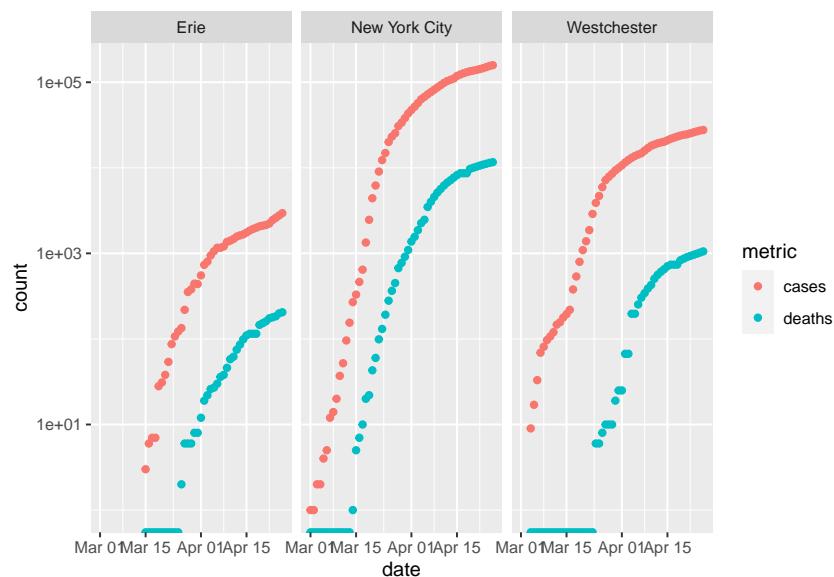
```
p + facet_grid(rows=vars(county))
## Warning: Transformation introduced infinite values in continuous y-axis
```



Note the common scales on the x and y axes.

- Plotting counties as ‘rows’ of the graph emphasize temporal comparisons – e.g., the earlier onset of the pandemic in Westchester and New York City compared to Erie, and perhaps longer lag between new cases and deaths in Westchester.
- Plotting counties as ‘columns’ emphasizes comparison between number of cases and deaths – there are many more cases in New York City than in Erie County.

```
p + facet_grid(cols=vars(county))
## Warning: Transformation introduced infinite values in continuous y-axis
```



COVID-19 nationally

Setup

- Summarize the total (maximum) number of cases in each county and state

```
county_summary <-
  us %>%
  group_by(county, state) %>%
  summarize(
    cases = max(cases),
    deaths = max(deaths)
  )
```

```

county_summary
## # A tibble: 2,859 x 4
## # Groups:   county [1,676]
##   county     state       cases  deaths
##   <chr>      <chr>     <dbl>   <dbl>
## 1 Abbeville South Carolina    27      0
## 2 Acadia      Louisiana      130      7
## 3 Accomack   Virginia      187      3
## 4 Ada         Idaho        615     15
## 5 Adair       Iowa          1        0
## 6 Adair       Kentucky      83       9
## 7 Adair       Missouri      12       0
## 8 Adair       Oklahoma      55       3
## 9 Adams       Colorado      1348     58
## 10 Adams      Idaho         3        0
## # ... with 2,849 more rows

```

- Now summarize the number of cases per state

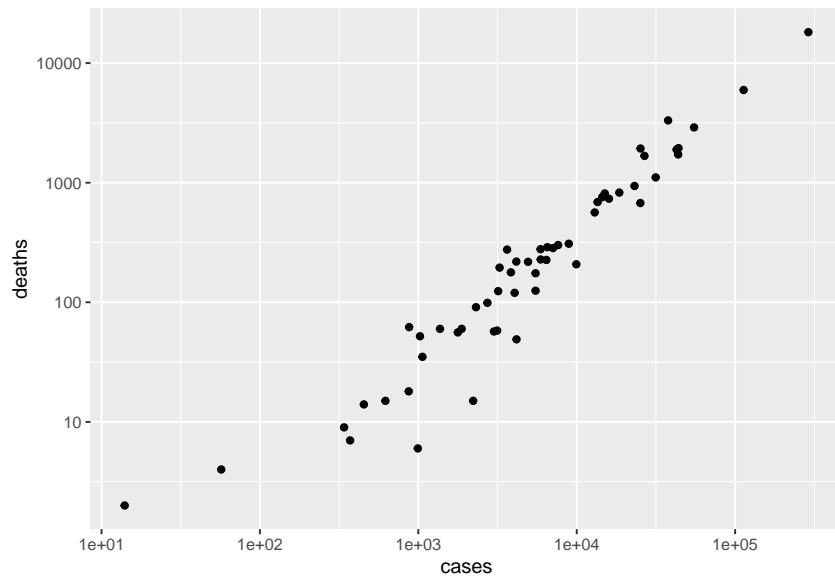
```

state_summary <-
  county_summary %>%
  group_by(state) %>%
  summarize(
    cases = sum(cases),
    deaths = sum(deaths)
  ) %>%
  arrange(desc(cases))
state_summary
## # A tibble: 55 x 3
##   state       cases  deaths
##   <chr>     <dbl>   <dbl>
## 1 New York  290167  18145
## 2 New Jersey 113223  5960
## 3 Massachusetts 55060  2899
## 4 Illinois   43916  1946
## 5 California 43696  1723
## 6 Pennsylvania 42717  1892
## 7 Michigan    37761  3318
## 8 Florida     31535  1108
## 9 Louisiana   26782  1676
## 10 Connecticut 25284  1932
## # ... with 45 more rows

```

- Plot the relationship between cases and deaths as a scatter plot

```
ggplot(state_summary, aes(cases, deaths)) +
  scale_x_log10() +
  scale_y_log10() +
  geom_point()
```



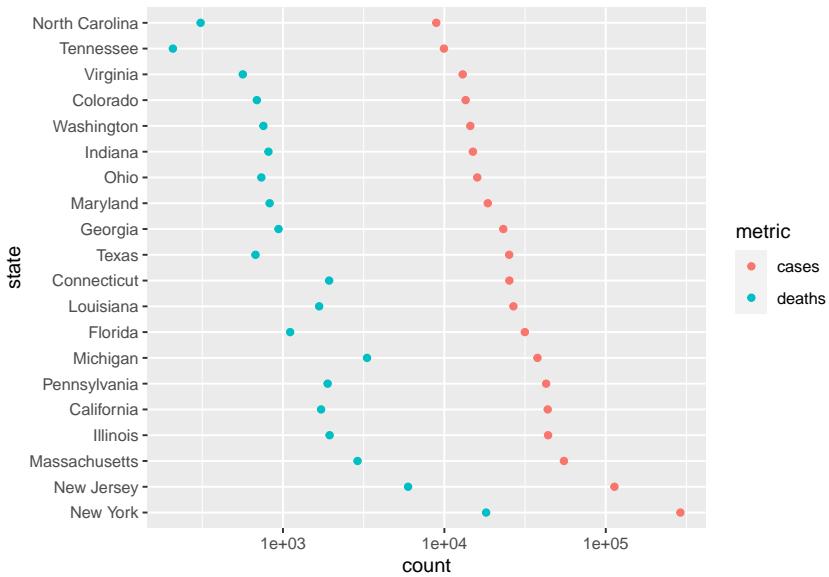
- Create a ‘long’ version of the state summary. The transformations include making ‘state’ a factor with the ‘levels’ ordered from most- to least-affected state. This is a ‘trick’ so that states are ordered, when displayed, from most to least affected. The transformations also choose only the 20 most-affected states using `head(20)`.

```
state_longer <-
  state_summary %>%
  mutate(
    ## this 'trick' causes 'state' to be ordered from most to
    ## least cases, rather than alphabetically
    state = factor(state, levels = state)
  ) %>%
  head(20) %>% # look at the 20 states with the most cases
  pivot_longer(
    c("cases", "deaths"),
    names_to = "metric",
    values_to = "count"
  )
state_longer
## # A tibble: 40 x 3
```

```
##   state      metric  count
##   <fct>     <chr>   <dbl>
## 1 New York    cases  290167
## 2 New York    deaths  18145
## 3 New Jersey   cases  113223
## 4 New Jersey   deaths  5960
## 5 Massachusetts cases  55060
## 6 Massachusetts deaths  2899
## 7 Illinois     cases  43916
## 8 Illinois     deaths  1946
## 9 California   cases  43696
## 10 California  deaths  1723
## # ... with 30 more rows
```

- Use a dot plot to provide an alternative representation that is more easy to associate statistics with individual states

```
ggplot(state_longer, aes(x = count, y = state, color = metric)) +
  scale_x_log10() +
  geom_point()
```



3.4 Day 18 Worldwide COVID data

Setup

- Start a new script and load the packages we'll use

```
library(readr)
library(dplyr)
library(ggplot2)
library(tidyr)      # specialized functions for transforming tibbles
```

These packages should have been installed during previous quarantines.

Source

- CSSE at Johns Hopkins University, available on github

```
hopkins = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data.csv" <- read_csv(hopkins)
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   `Province/State` = col_character(),
##   `Country/Region` = col_character()
## )
## See spec(...) for full column specifications.
```

‘Tidy’ data

- The data has initial columns describing region, and then a column for each date of the pandemic. There are 264 rows, corresponding to the different regions covered by the database.
- We want instead to ‘pivot’ the data, so that each row represents cases in a particular region on a particular date, analogous to the way the US data we have been investigating earlier has been arranged.
- `tidy` provides functions for manipulating a `tibble` into ‘tidy’ format.
- `tidy::pivot_longer()` takes a ‘wide’ data frame like `csv`, and allows us to transform it to the ‘long’ format we are interested in.
 - I discovered how to work with `pivot_longer()` using its help page `?tidy::pivot_longer`
 - The first argument represents columns to pivot or, as a convenience when these are negative values, columns we *do not* want to pivot. We *do not* want to pivot columns 1 through 4, so this argument will be `-(1:4)`.

- The `names_to` argument is the column name we want to use to refer to the names of the columns that we *do* pivot. We’ll pivot the columns that have a date in them, so it makes sense to use `names_to = "date"`.
- The `values_to` argument is the column name we want to use for the pivoted values. Since the values in the main part of `csv` are the number of cases observed, we’ll use `values_to = "cases"`
- Here’s what we have after pivoting

```
csv %>%
  pivot_longer(-(1:4), names_to = "date", values_to = "cases")
## # A tibble: 25,608 x 6
##   `Province/State` `Country/Region`   Lat  Long date      cases
##   <chr>            <chr>          <dbl> <dbl> <chr>     <dbl>
## 1 <NA>              Afghanistan      33    65  1/22/20     0
## 2 <NA>              Afghanistan      33    65  1/23/20     0
## 3 <NA>              Afghanistan      33    65  1/24/20     0
## 4 <NA>              Afghanistan      33    65  1/25/20     0
## 5 <NA>              Afghanistan      33    65  1/26/20     0
## 6 <NA>              Afghanistan      33    65  1/27/20     0
## 7 <NA>              Afghanistan      33    65  1/28/20     0
## 8 <NA>              Afghanistan      33    65  1/29/20     0
## 9 <NA>              Afghanistan      33    65  1/30/20     0
## 10 <NA>             Afghanistan      33    65  1/31/20     0
## # ... with 25,598 more rows
```

- We’d like to further clean this up data
 - Format our newly created ‘date’ column (using `as.Date()`, but with a `format=` argument appropriate for the format of the dates in this data set)
 - Re-name, for convenience, the `Country/Region` column as just `country`.
 - Select only columns of interest – `country`, `date`, `cases`
 - Some countries have multiple rows, because the data is a provincial or state levels, so we would like to sum all cases, grouped by `country` and `date`

```
world <-
  csv %>%
  pivot_longer(-(1:4), names_to = "date", values_to = "cases") %>%
  mutate(
    country = `Country/Region`,
```

```

    date = as.Date(date, format = "%m/%d/%y")
) %>%
group_by(country, date) %>%
summarize(cases = sum(cases))
world
## # A tibble: 17,945 x 3
## # Groups:   country [185]
##       country     date   cases
##       <chr>     <date>   <dbl>
## 1 Afghanistan 2020-01-22     0
## 2 Afghanistan 2020-01-23     0
## 3 Afghanistan 2020-01-24     0
## 4 Afghanistan 2020-01-25     0
## 5 Afghanistan 2020-01-26     0
## 6 Afghanistan 2020-01-27     0
## 7 Afghanistan 2020-01-28     0
## 8 Afghanistan 2020-01-29     0
## 9 Afghanistan 2020-01-30     0
## 10 Afghanistan 2020-01-31    0
## # ... with 17,935 more rows

```

- Let's also calculate `new_cases` by country
 - Use `group_by()` to perform the `new_cases` computation for each country
 - Use `mutate()` to calculate the new variable
 - Use `ungroup()` to remove the grouping variable, so it doesn't unexpectedly influence other calculations
 - re-assign the updated `tibble` to the variable `world`

```

world <-
  world %>%
  group_by(country) %>%
  mutate(new_cases = diff(c(0, cases))) %>%
  ungroup()

```

Exploration

- Use `group_by()` and `summarize()` to find the maximum (total) number of cases, and `arrange()` these `desc()`'ending order

```

world %>%
  group_by(country) %>%
  summarize(n = max(cases)) %>%

```

```

arrange(desc(n))
## # A tibble: 185 x 2
##   country             n
##   <chr>              <dbl>
## 1 US                 988197
## 2 Spain               229422
## 3 Italy                199414
## 4 France               165963
## 5 Germany              158758
## 6 United Kingdom       158348
## 7 Turkey                112261
## 8 Iran                  91472
## 9 Russia                87147
## 10 China                 83918
## # ... with 175 more rows

```

Visualization

- Start by creating a subset, e.g., the US

```

country <- "US"
us <-
  world %>%
  filter(country == "US")

```

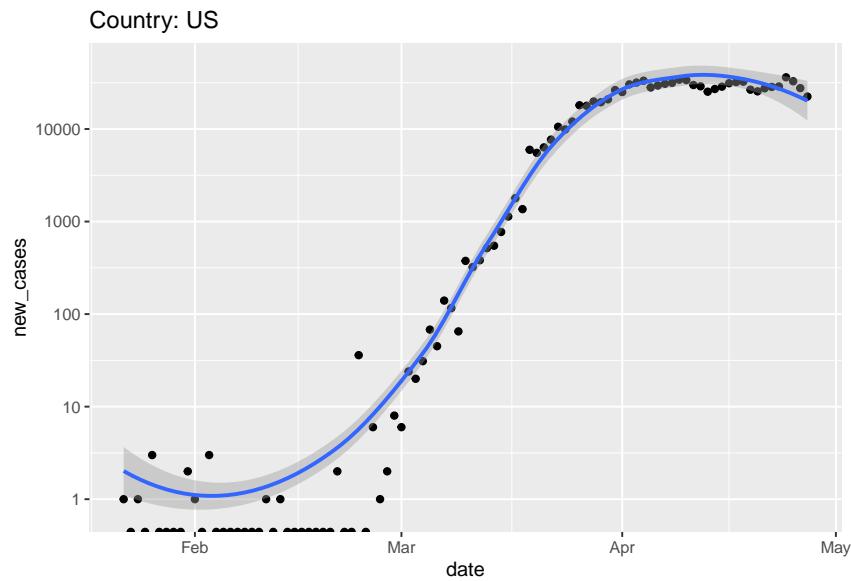
- Use ggplot2 to visualize the progression of the pandemic

```

ggplot(us, aes(date, new_cases)) +
  scale_y_log10() +
  geom_point() +
  geom_smooth() +
  ggtitle(paste("Country:", country))
## Warning: Transformation introduced infinite values in continuous y-axis

## Warning: Transformation introduced infinite values in continuous y-axis
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
## Warning: Removed 25 rows containing non-finite values (stat_smooth).

```



It seems like it would be convenient to capture our data cleaning and visualization steps into separate functions that can be re-used, e.g., on different days or for different visualizations.

- write a function for data retrieval and cleaning

```
get_world_data <-
  function()
{
  ## read data from Hopkins' github repository
  hopkins = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_time_series/time_series_covid_19_new.csv"
  csv <- read_csv(hopkins)

  ## 'tidy' the data
  world <-
    csv %>%
    pivot_longer(-(1:4), names_to = "date", values_to = "cases") %>%
    mutate(
      country = `Country/Region`,
      date = as.Date(date, format = "%m/%d/%y")
    )

  ## sum cases across regions within aa country
  world <-
    world %>%
    group_by(country, date) %>%
```

```

    summarize(cases = sum(cases))

    ## add `new_cases`, and return the result
    world %>%
      group_by(country) %>%
      mutate(new_cases = diff(c(0, cases))) %>%
      ungroup()
}

```

- ...and for plotting by country

```

plot_country <-
  function(tbl, view_country = "US")
{
  country_title <- paste("Country:", view_country)

  ## subset to just this country
  country_data <-
    tbl %>%
    filter(country == view_country)

  ## plot
  country_data %>%
    ggplot(aes(date, 1 + new_cases)) +
    scale_y_log10() +
    geom_point() +
    ## add method and formula to quieten message
    geom_smooth(method = "loess", formula = y ~ x) +
    ggtitle(country_title)
}

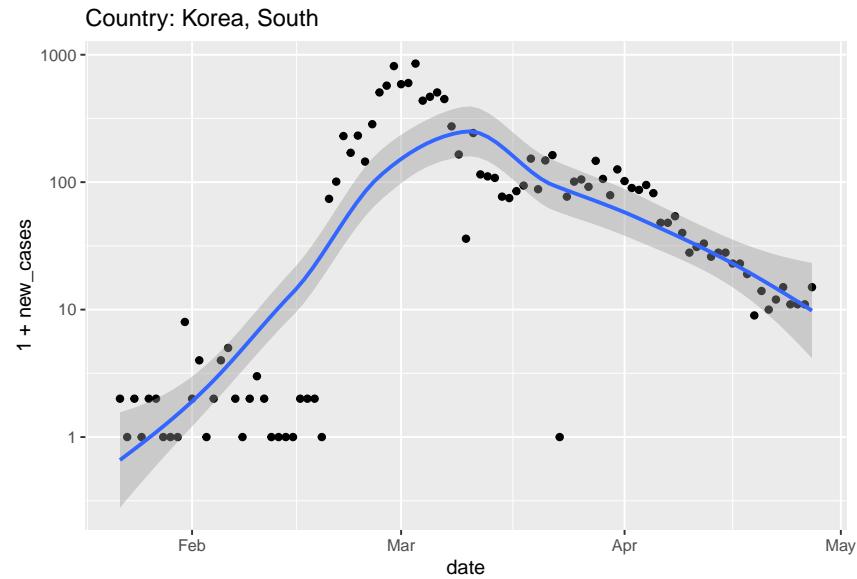
```

- Note that, because the first argument of `plot_country()` is a tibble, the output of `get_world_data()` can be used as the input of `plot_country()`, and can be piped together, e.g.,

```

world <- get_world_data()
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   `Province/State` = col_character(),
##   `Country/Region` = col_character()
## )
## See spec(...) for full column specifications.
world %>% plot_country("Korea, South")

```



3.5 Day 19 (Friday) Zoom check-in

3.5.1 Review and trouble shoot (25 minutes)

3.5.2 Next week (25 minutes)

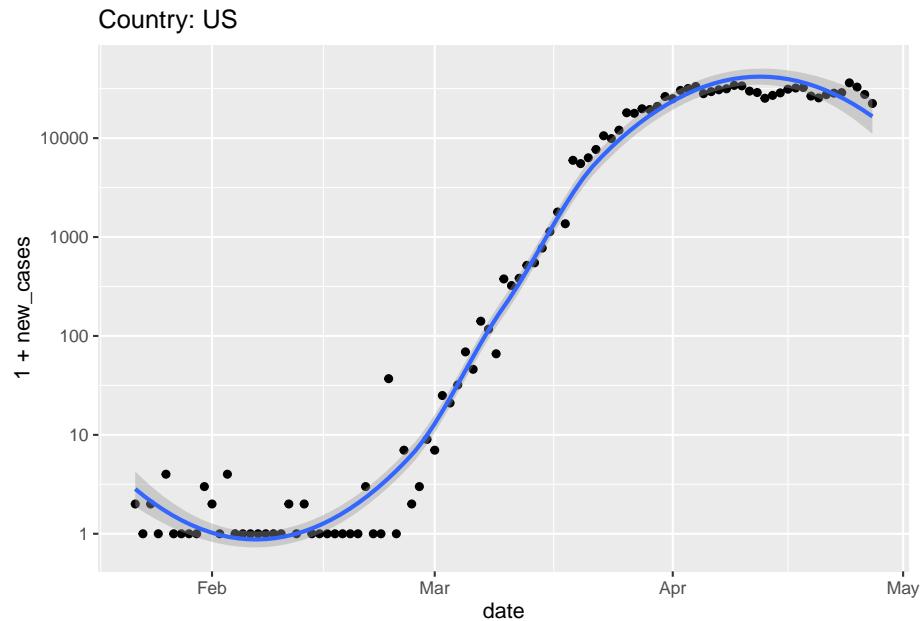
3.6 Day 20 Exploring the course of pandemic in different regions

Use the data and functions from quarantine day 18 to place the pandemic into quantitative perspective. Start by retrieving the current data

```
world <- get_world_data()
```

Start with the United States

```
world %>% plot_country("US")
```

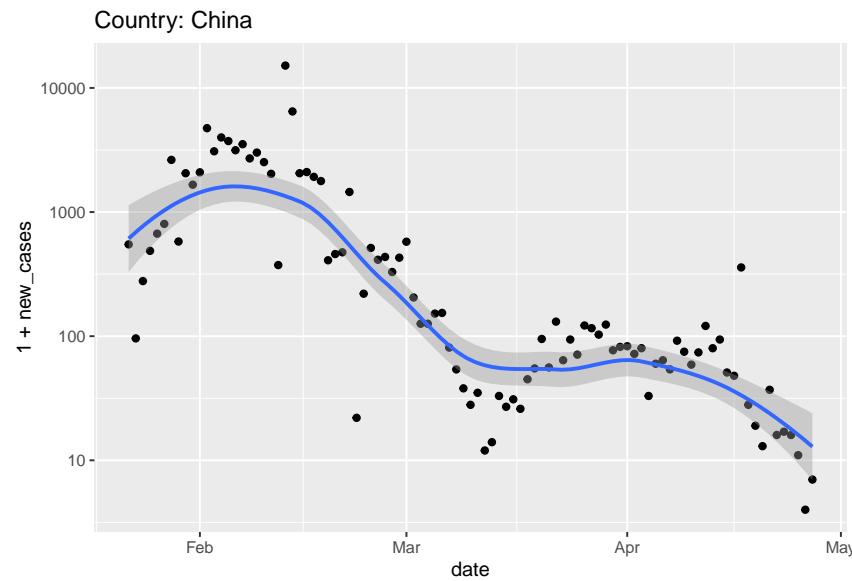


- When did ‘stay at home’ orders come into effect? Did they appear to be effective?
- When would the data suggest that the pandemic might be considered ‘under control’, and country-wide stay-at-home orders might be relaxed?

Explore other countries.

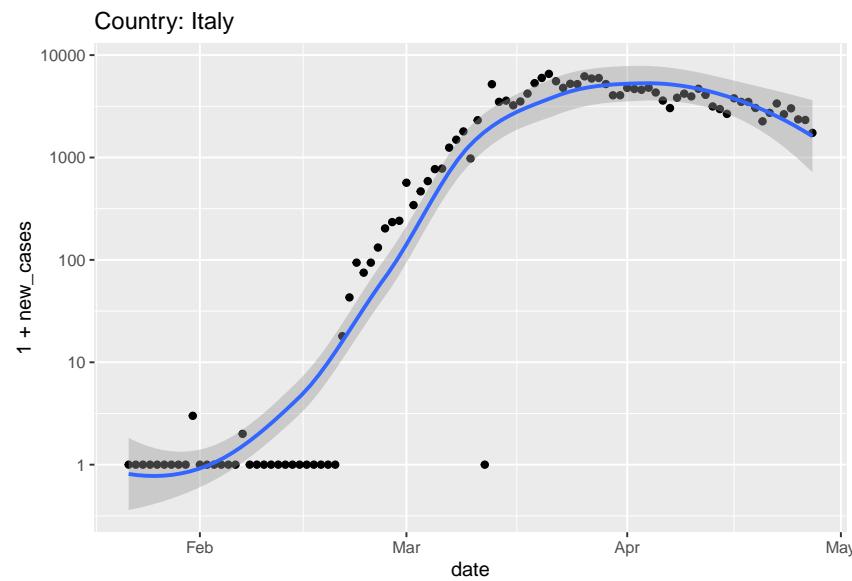
- The longest trajectory is probably displayed by China

```
world %>% plot_country("China")
```

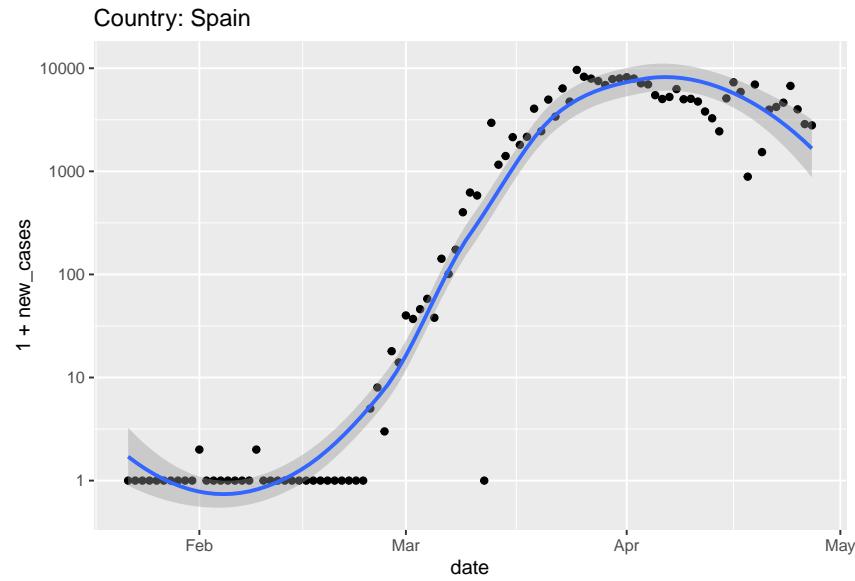


- Italy and Spain were hit very hard, and relatively early, by the pandemic

```
world %>% plot_country("Italy")
```

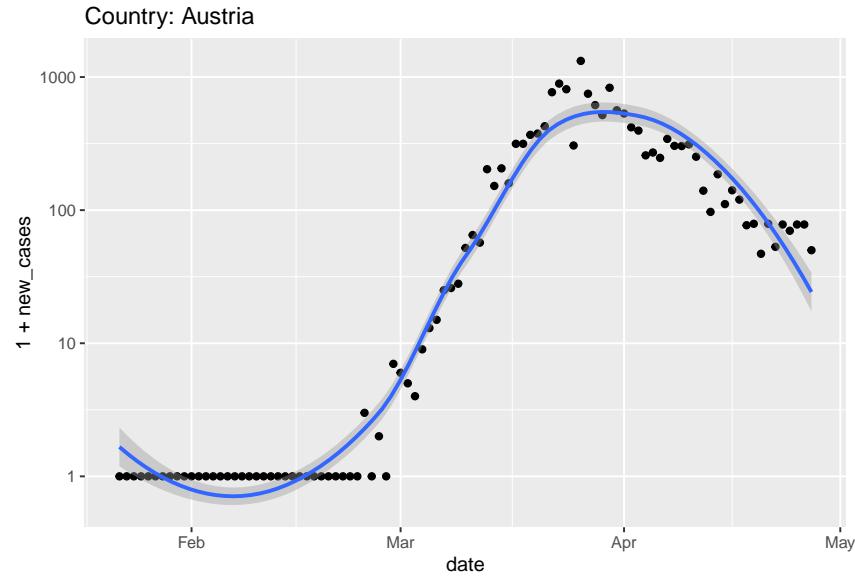


```
world %>% plot_country("Spain")
```



- Austria relaxed quarantine very early, in the middle of April; does that seem like a good idea?

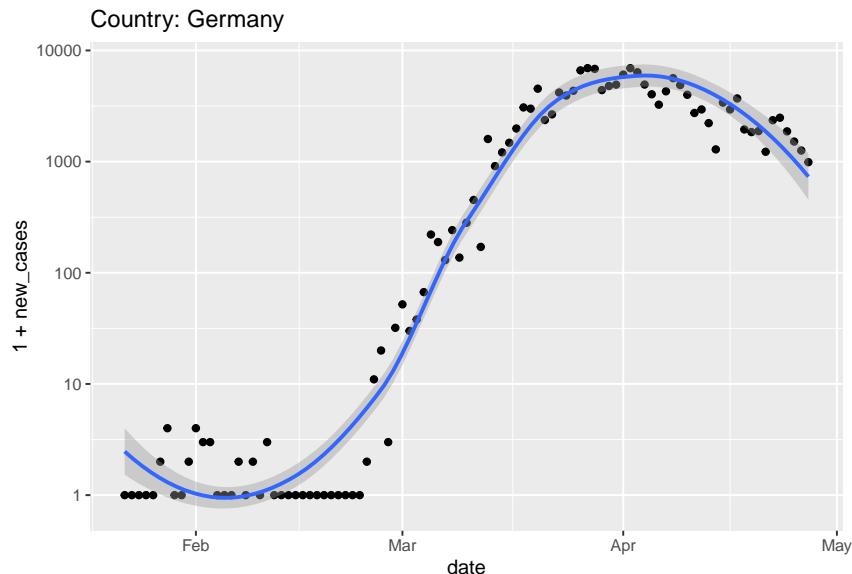
```
world %>% plot_country("Austria")
```



- Germany also had strong leadership (e.g., chancellor Angela Merkel provided clear and unambiguous rules for Germans to follow, and then self-isolated when her doctor, whom she had recently visited, tested positive)

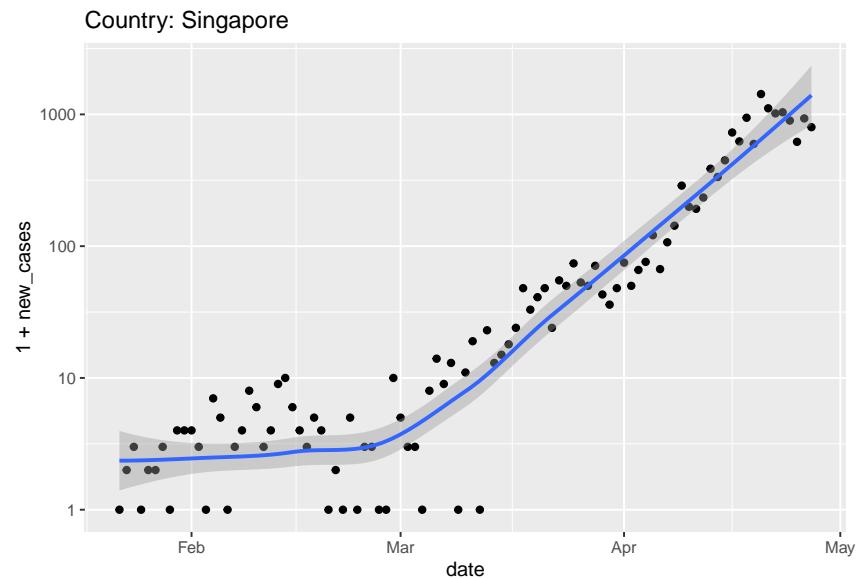
and an effective screening campaign (e.g., to make effective use of limited testing resources, in some instances pools of samples were screened, and only if the pool indicated infection were the individuals in the pool screened).

```
world %>% plot_country("Germany")
```



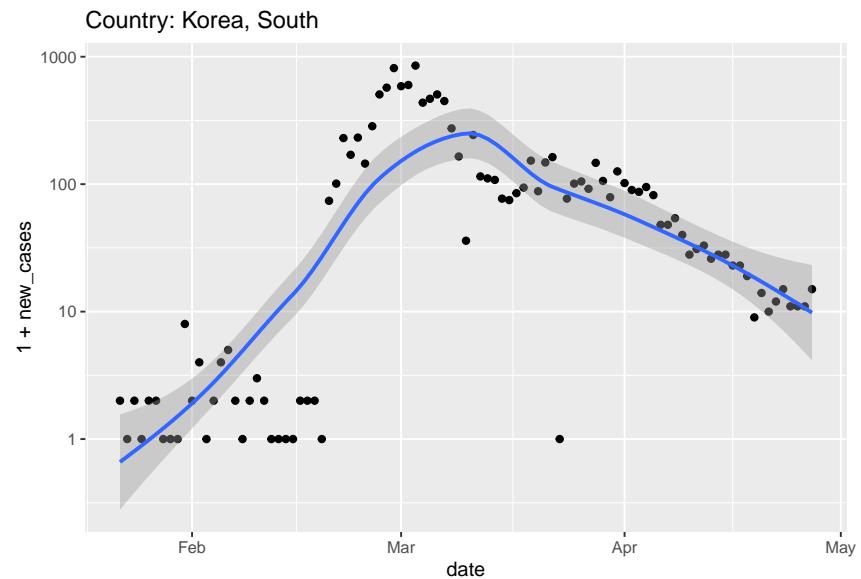
- At the start of the pandemic, Singapore had excellent surveillance (detecting individuals with symptoms) and contact tracing (identifying and placing in quarantine those individuals coming in contact with the infected individuals). New cases were initially very low, despite proximity to China, and Singapore managed the pandemic through only moderate social distancing (e.g., workers were encouraged to operate in shifts; stores and restaurants remained open). Unfortunately, Singaporeans returning from Europe (after travel restrictions were in place there) introduced new cases that appear to have overwhelmed the surveillance network. Later, the virus spread to large, densely populated migrant work housing. Singapore's initial success at containing the virus seems to have fallen apart in the face of this wider spread, and more severe restrictions on economic and social life were imposed.

```
world %>% plot_country("Singapore")
```



- South Korea had a very ‘acute’ spike in cases associated with a large church. The response was to deploy very extensive testing and use modern approaches to tracking (e.g., cell phone apps) coupled with transparent accounting. South Korea imposed relatively modest social and economic restrictions. It seems like this has effectively ‘flattened the curve’ without pausing the economy.

```
world %>% plot_country("Korea, South")
```



Where does your own exploration of the data take you?

3.7 Day 21

Self-directed activities.

Chapter 4

Machine learning

This week you will learn about machine learning for classification using *R*. Objectives are:

- To provide an overview of the underlying concepts of machine learning for classification.
- To provide an introduction to some popular classification algorithms.
- To explore these classification algorithms using various packages in *R*.
- To choose a classification algorithm and apply it to the statewide COVID-19 dataset.

4.1 Day 22 (Monday) Zoom check-in

Here is an overview of what we'll cover in today's Zoom session:

- Overview of Machine Learning for Classification (30 minutes)
- Introduction to the Random Forest algorithm (5 minutes)
- A Random Forest Example in *R* using COVID-19 Data (25 minutes)

4.1.1 A Machine Learning Primer

Machine Learning (ML) may be defined as using computers to make inferences about data.

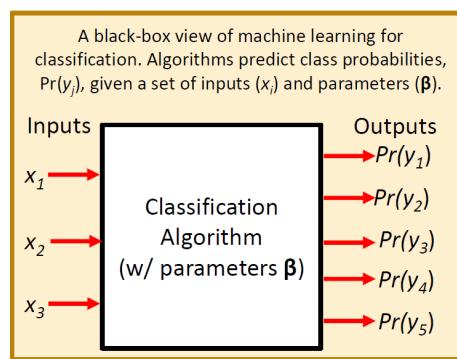
- A mapping of inputs to outputs, $Y = f(X, \beta)$
- ML for Classification refers to algorithms that map inputs to a discrete set of outputs (i.e. classes or categories)
 - For example, predicting health risk (mild, moderate, severe) based on patient data (height, weight, age, smoker?)
 - Or predicting pandemic severity index (PSI) of COVID-19 in a state based on statewide population data.

```
##  PSI Death.Rate    Example
##  1    < 0.1%      seasonal flu
##  2    0.1% - 0.5%  Asian flu
##  3    0.5% - 1.0%  n/a
##  4    1.0% - 2.0%  n/a
##  5    > 2.0%       Spanish flu
```

- Predictions are typically expressed as a vector of probabilities.
 - e.g. $\Pr(\text{cancerous})$ vs. $\Pr(\text{benign})$
 - e.g. $\Pr(\text{PSI}=1), \Pr(\text{PSI}=2), \dots, \Pr(\text{PSI}=5)$

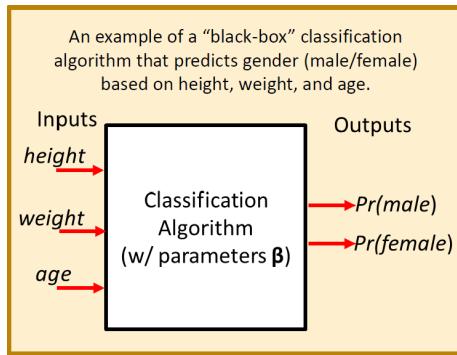
A “Black Box” View of Machine Learning

The diagram below illustrates the concept in terms of a “black box” model that converts inputs into predictions.



A “Black Box” Example

The diagram below illustrates a “black box” model that converts height, weight, and age into a prediction of whether or not the individual is male or female.



Some Important Machine Learning Terms

The above example highlights some important machine learning terminology:

- Features (X): the inputs (also known as descriptive attributes or explanatory variables) to the algorithm.
- Parameters (β): these are internal variables of the selected machine learning algorithm and are also known as coefficients or training weights. Algorithm parameters need to be adjusted to minimize the deviation between predictions and observations.
- Labels (Y): these are the outputs of the algorithm (i.e. the categories you are attempting to predict).
- Training Data: this is a data sets containing paired observations of inputs (i.e features) and outputs (i.e. labels). Training data is also known as measurement data, observation data or calibration data.
- Training: this is the process of adjusting algorithm parameters (β) to obtain the best possible match between training data and corresponding model predictions. Training is also known as model calibration or parameter estimations.

An example set of training data that could be used in the gender prediction example is given below:

```
##  height_m weight_kg age_y gender
##  1.69      62     30   male
##  1.74      76     27 female
##  1.92      82     25   male
##  1.80     100     41   male
##  1.59      47     24 female
```

```

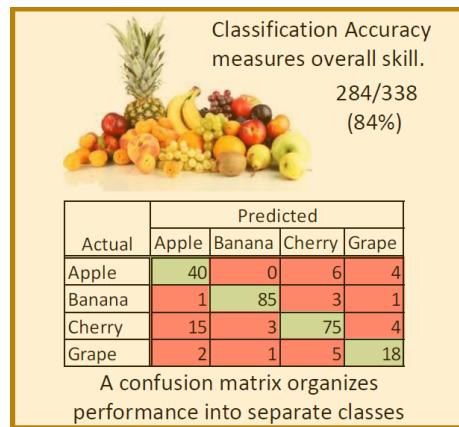
##  1.85    75    26   male
##  1.75    63    33 female
##  1.96    83    33   male
##  1.85    39    32   male
##  1.78    58    28 female
##  1.74    70    30 female
##  1.81    57    26 female
##  1.73    78    32   male

```

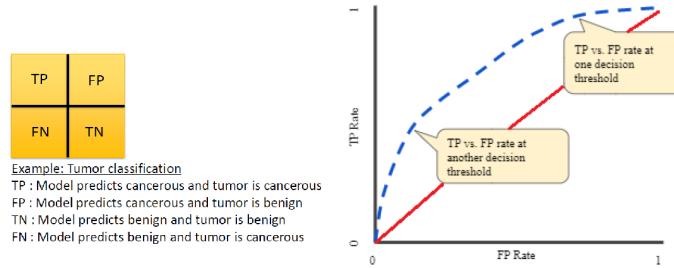
Upon completion of training, it is important to evaluate the quality of the model and its skill or ability at making correct predictions. Some terms related to this evaluation process are defined below:

- Classification Accuracy: this is the ratio of correct predictions to the total predictions when validating a trained classification algorithm.
- Confusion Matrix: this is a more detailed summary (relative to classification accuracy) of the performance of a classification algorithm. The diagonals of the matrix count how often the algorithm yielded the correct classification for each class. The off-diagonal entries count how often the algorithm confused one class with another.

The figure below illustrates the classification accuracy and confusion matrix for an example that attempts to classify images of fruits.



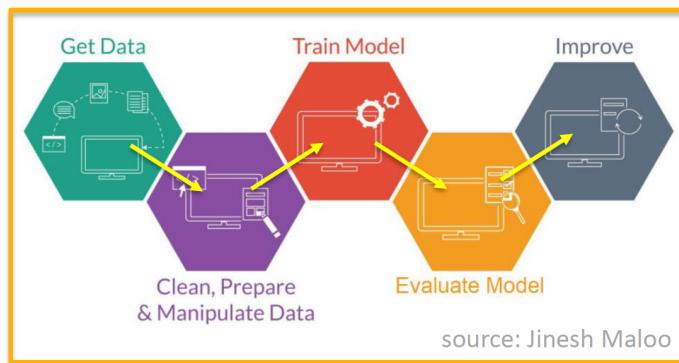
- ROC curve: As illustrated in the figure below, the ROC (Receiver Operating Characteristic) curve plots the true positive (TP) vs. false positive (FP) rate at various probability thresholds. In the figure, the dashed blue line represents a hypothetical ROC curve for some machine learning model and the solid red line is the curve for a “non-informative” model (i.e. a model that makes a uniform random guess). As such, we’d like the blue curve to be as far above the red curve as possible.



- AUC: this stands for “area under curve” and is the area under the ROC curve. In the previous figure, the AUC would be the area under the dashed blue curve. Values of AUC quantify the degree to which an ROC curve lies above (or below) the red “non-informative” curve. Some interesting AUC values:
 - AUC = 0.0: the model is always wrong (with respect to TP vs. FP)
 - AUC = 0.5: the model is no better than guessing (i.e. the model matches the red “non-informative” curve in the figure)
 - AUC = 1.0: the model is always right (with respect to TP vs. FP)

The Machine Learning Process

Now that we’ve defined some of the important machine learning terminology let’s take a 30,000 foot view at the overall machine learning process. This is a general description of the process that you’ll want to follow each time you build and use a machine learning model. The process is illustrated in the figure below (with credit to Jinesh Maloo):



- Step 1: Prepare labeled data for training and validation.
- Step 2: Select a machine learning algorithm (i.e. model).

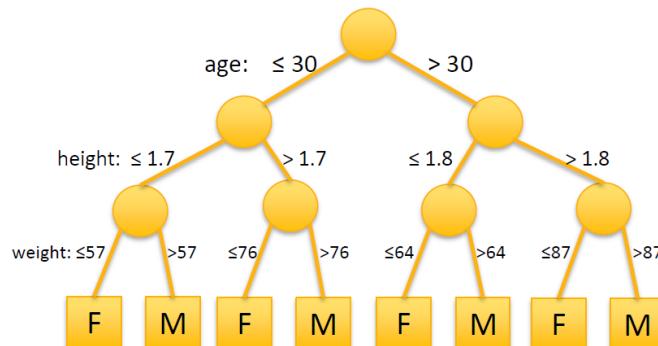
- Step 3: Train the model.
- Step 4: Evaluate model performance.
- If model is useful:
 - Step 5: Apply to unlabeled data.
- Else (needs improvement):
 - Collect more data (go back to Step 1).
 - Revise model (go back to Step 2)

4.1.2 The Random Forest Algorithm

The random forest algorithm is a popular choice for machine learning.

- Over 20 *R* packages have an implementation of some form of the algorithm.
- We'll be using the `randomForest` package.
- The algorithm is like `bagging` (bootstrap aggregating) regression trees, but the regression trees are de-correlated.

The figure below illustrates one possible **tree** in a **random forest** for a gender prediction model. In computer science terminology, each split in the figure is a **branch** of a graph **tree**. In simple terms, the split points are randomly generated and the resulting **trees** combine to form a **random forest**.

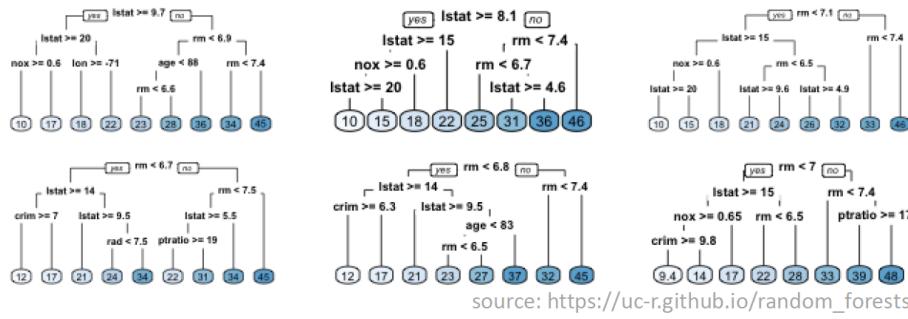


One possible classification tree for predicting gender.

The bagging process develops multiple trees and makes a prediction by combining the results.

The figure below illustrates a set of 6 **trees** that make up a **random forest** for predicting housing prices. The image is courtesy Bradley Boehmke at the University of Cincinnati. Examine the figure closely and notice that:

- The split variables can differ across trees (not all variables are included in all trees).
- The split variables can differ within trees (not all paths consider the same set of variables).
- The order of splits can differ.
- The split values can differ.



The job of the random forest algorithm is to determine the optimal set of trees for your data set, including the splitting configuration (i.e. order, values, etc.).

Preparing a COVID-19 Dataset for Machine Learning

We'd like to predict the severity of COVID-19 in a given state using statewide feature data like population, urban density, number of hospital beds, date of stay at home order, etc. We've already seen that we can get the information about cases and deaths from the New York Times github page. However, gathering corresponding statewide feature data requires quite a bit of hunting through various public websites. Consequently, we're going to skip over the painstaking process of marshalling the feature data and just provide you with a dataset that is already nice and prepped for machine learning.

You'll work with two **.csv** files - a data file that contains a variety of statewide data, and a metadata file that describes the various columns of the data file. This combination of data and metadata files is a common way of sharing publicly available datasets.

To give you an idea of what was involved in assembling the data and metadata file, a summary of the data collection and processing steps is given below:

- First, a snapshot of the New York Times COVID-19 data from April 27th was downloaded from the nytimes github repo and stored on local disk.

```
$ wget https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv
$ mv us-counties.csv us-counties_04_27_2020.csv
```

- The data was processed using *R* :

- the cases and deaths in `us-counties_04_27_2020.csv` were aggregated into statewide values.
- the death rate was calculated and categorized according to an 8-point severity index
- finally, the statewide data was exported as a `.csv` file

```
# get data from file
covid_data_file = file.path("C:\\\\Matott\\\\MyQuarantine", "us-counties_04_27_2020.csv")

# read in as data frame
us_data = read.csv(covid_data_file, stringsAsFactors = FALSE)

# aggregate by county and state
cases_county_state <- aggregate( cases ~ county + state, us_data, max )
deaths_county_state <- aggregate( deaths ~ county + state, us_data, max )

# aggregate by state
cases_state <- aggregate(cases ~ state, cases_county_state, sum)
deaths_state <- aggregate(deaths ~ state, deaths_county_state, sum)

# calculate death rate
death_rate_state <- 100.00 * (deaths_state$deaths / cases_state$cases)

# Assign the following severity index using cut:
## PSI Death.Rate
## 1 < 0.1%
## 2 0.1% - 0.5%
## 3 0.5% - 1.0%
## 4 1.0% - 2.0%
## 5 2.0% - 4.0%
## 6 4.0% - 6.0%
## 7 6.0% - 8.0%
## 8 >8.0%
psi_state <- cut(death_rate_state,
                  breaks=c(0.0,0.1,0.5,1.0,2.0,4.0,6.0,8.0,100.0),
                  labels=c(1,2,3,4,5,6,7,8))
```

```
# combine into new data frame
out_df = data.frame(
  state=cases_state$state,
  cases=cases_state$cases,
  deaths=deaths_state$deaths,
  death_rate=death_rate_state,
  severity_index=psi_state)

# write out as csv
my_out_file <- file.path("C:\\\\Matott\\\\MyQuaRantine", "covid_data.csv")
write.csv(out_df, file = my_out_file, row.names = FALSE)
```

- The resulting statewide COVID-19 label data (i.e. what we would like to predict) was augmented with 32 statewide features, including population, percent urban, number of hospital beds, etc. Feature data was collected from a variety of sources, including the Center for Disease Control, the American Heart Association, the U.S. Census Bureau, etc. In some cases the feature data was available for direct download (e.g. as a .csv file) and in other cases the feature data was manually harvested (e.g. cut-and-paste from websites).
- The augmented (i.e. features + labels) .csv file was split into two .csv files that you will need to download:
 - statewide_covid_19_data_04_27_2020.csv: This file contains the final COVID-19 machine learning data set, but features and labels are coded so that feature columns are named X01, X02, X03, etc. and label columns are named Y01, Y02, Y03, etc.
 - statewide_covid_19_metadata_04_27_2020.csv: This file maps the column names in the data file to more meaningful names and descriptions (including units) of the associated variables. For example X01 is Pct_Sun and has a description of Percent sunny days. This is known as metadata - data that describes other data.

Click on the links above to download the data and metadata files that you'll need for the machine learning examples presented throughout the week.

4.1.3 A Random Forest Example Using COVID-19 Data

Let's apply the random forest algorithm to the COVID-19 dataset. We'll build out the required *R* code in sections. To get started, open a new *R* script in *RStudio* and name it `covid_19_rf.R`. Enter the code below:

```

library(randomForest)
## randomForest 4.6-14
## Type rfNews() to see new features/changes/bug fixes.
library(readr)

data_file = file.path("C:/Matott/MyQuaRantine",
                      "statewide_covid_19_data_04_27_2020.csv")
df = read_csv(data_file)
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   State = col_character(),
##   X31 = col_character(),
##   X32 = col_character()
## )
## See spec(...) for full column specifications.
# coerce severity to a factor (so RF algorithm uses classification)
df$Y04 = as.factor(df$Y04)
df
## # A tibble: 50 x 37
##   State    X01    X02    X03    X04    X05    X06    X07    X08    X09    X10    X11
##   <chr> <dbl> <dbl>
## 1 Alab~    58  2641     99  71.6    54  4.90e6   827  16.9  42240    3.9  3.5
## 2 Alas~    41  2061     61  77.1   26.5  7.32e5    87  11.8  59605    6.6  9.44
## 3 Arizo~    85  3806    193  38.5   32.5  7.28e6   1258  17.5  44414    4.8  8.38
## 4 Arkans~   61  2771    123  70.9   50.7  3.02e6    512  17    43292    3.7  4.51
## 5 Califo~   68  3055    146  61    44.2  3.95e7   5669  14.3  63711    4.2  3.14
## 6 Colono~   71  3204    136  54.1   28.8  5.76e6    808  14.2  58500    3.3  4.63
## 7 Connec~   56  2585     82  69.2   40.7  3.57e6    615  17.2  76481    4.1  2.58
## 8 Delawar~   55  2500     97  70.9   45.8  9.74e5   181  18.7  52599    3.8  3.64
## 9 Florid~   66  2927    101  74.5   62.7  2.15e7   4358  20.5  50199    3.6  7.55
## 10 Georig~  66  2986    112  71.1   53.4  1.06e7   1460  13.9  46519    3.9  8.14
## # ... with 40 more rows, and 25 more variables: X12 <dbl>, X13 <dbl>,
## #   X14 <dbl>, X15 <dbl>, X16 <dbl>, X17 <dbl>, X18 <dbl>, X19 <dbl>,
## #   X20 <dbl>, X21 <dbl>, X22 <dbl>, X23 <dbl>, X24 <dbl>, X25 <dbl>,
## #   X26 <dbl>, X27 <dbl>, X28 <dbl>, X29 <dbl>, X30 <dbl>, X31 <chr>,
## #   X32 <chr>, Y01 <dbl>, Y02 <dbl>, Y03 <dbl>, Y04 <fct>

metadata_file = file.path("C:/Matott/MyQuaRantine",
                          "statewide_covid_19_metadata_04_27_2020.csv")
mdf = read_csv(metadata_file)
## Parsed with column specification:
## cols(
##   ID = col_double(),
##   Code = col_character(),

```

```

##   Variable = col_character(),
##   Description = col_character()
## )
mdf
## # A tibble: 36 x 4
##       ID Code  Variable          Description
##   <dbl> <chr> <chr>
## 1     1 X01  Pct_Sun    Percent sunny days
## 2     2 X02  Total_Hours_Sun  Total hours of sun
## 3     3 X03  Num_Clear_Days Number of clear days
## 4     4 X04  Avg_RH      Average relative humidity
## 5     5 X05  Avg_Dew_Point Average dew point
## 6     6 X06  Total_Population Total population
## 7     7 X07  Senior_Pop_Thousands Population 65+ years in thousands
## 8     8 X08  Senior_Pop_Pct  Percentage of population 65+ year
## 9     9 X09  per_capita_income per capita income
## 10   10 X10  Unemployment_Rate Percent unemployment
## # ... with 26 more rows

```

Enter the code above and try to run it. You may get an errors about missing the `randomForest` package. You can install it from the RStudio console (see below) or using the installer in the RStudio packages pane.

```
install.packages("randomForest")
```

Now we've loaded the data and metadata file. Let's pick a subset of 5 of the features and use them to try and predict the pandemic severity index (i.e. Y04):

```

library(dplyr)
##
## Attaching package: 'dplyr'
## The following object is masked from 'package:randomForest':
##
##     combine
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

# describe all possible features and labels
print(mdf, n=nrow(mdf))
## # A tibble: 36 x 4

```

<code>##</code>	<code>ID</code>	<code>Code</code>	<code>Variable</code>	<code>Description</code>
				<code><chr></code>
<code>##</code>	<code>1</code>	<code>X01</code>	<code>Pct_Sun</code>	<code>Percent sunny days</code>
<code>##</code>	<code>2</code>	<code>X02</code>	<code>Total_Hours_Sun</code>	<code>Total hours of sun</code>
<code>##</code>	<code>3</code>	<code>X03</code>	<code>Num_Clear_Days</code>	<code>Number of clear days</code>
<code>##</code>	<code>4</code>	<code>X04</code>	<code>Avg_RH</code>	<code>Average relative humidity</code>
<code>##</code>	<code>5</code>	<code>X05</code>	<code>Avg_Dew_Point</code>	<code>Average dew point</code>
<code>##</code>	<code>6</code>	<code>X06</code>	<code>Total_Population</code>	<code>Total population</code>
<code>##</code>	<code>7</code>	<code>X07</code>	<code>Senior_Pop_Thousa~</code>	<code>Population 65+ years in thousands</code>
<code>##</code>	<code>8</code>	<code>X08</code>	<code>Senior_Pop_Pct</code>	<code>Percentage of population 65+ year</code>
<code>##</code>	<code>9</code>	<code>X09</code>	<code>per_capita_income</code>	<code>per capita income</code>
<code>##</code>	<code>10</code>	<code>X10</code>	<code>Unemployment_Rate</code>	<code>Percent unemployment</code>
<code>##</code>	<code>11</code>	<code>X11</code>	<code>Uninsured_Rate_Ch~</code>	<code>Percent uninsured children</code>
<code>##</code>	<code>12</code>	<code>X12</code>	<code>Uninsured_Rate_Ad~</code>	<code>Percent uninsured adults</code>
<code>##</code>	<code>13</code>	<code>X13</code>	<code>Heart_Disease_Rate</code>	<code>Deaths per 100000 due to heart disease</code>
<code>##</code>	<code>14</code>	<code>X14</code>	<code>Heart_Disease_Dea~</code>	<code>Total deaths due to heart disease</code>
<code>##</code>	<code>15</code>	<code>X15</code>	<code>Tobacco_Use_Rate</code>	<code>Percentage of tobacco users</code>
<code>##</code>	<code>16</code>	<code>X16</code>	<code>Obesity_Prevalenc~</code>	<code>Percentage of population this is considered over weight</code>
<code>##</code>	<code>17</code>	<code>X17</code>	<code>Num_Hospitals</code>	<code>Number of hospitals</code>
<code>##</code>	<code>18</code>	<code>X18</code>	<code>Num_Hosp_Staffed_~</code>	<code>Number of hospital beds</code>
<code>##</code>	<code>19</code>	<code>X19</code>	<code>Total_Hosp_Discha~</code>	<code>Total number of hospital discharges</code>
<code>##</code>	<code>20</code>	<code>X20</code>	<code>Hosp_Patient_Days</code>	<code>Total number of patient days in hospital</code>
<code>##</code>	<code>21</code>	<code>X21</code>	<code>Hosp_Gross_Patien~</code>	<code>Total hospital gross patient revenue</code>
<code>##</code>	<code>22</code>	<code>X22</code>	<code>Number_of_Farms</code>	<code>Total number of farms</code>
<code>##</code>	<code>23</code>	<code>X23</code>	<code>Urban_Population_~</code>	<code>Percentage of population living in urban areas</code>
<code>##</code>	<code>24</code>	<code>X24</code>	<code>Urban_Population</code>	<code>Total urban population</code>
<code>##</code>	<code>25</code>	<code>X25</code>	<code>Urban_Land_Area_S~</code>	<code>Amount of urban land area in square miles</code>
<code>##</code>	<code>26</code>	<code>X26</code>	<code>Urban_Density_Sq~</code>	<code>Urban density in persons per square mile</code>
<code>##</code>	<code>27</code>	<code>X27</code>	<code>Urban_Land_Pct</code>	<code>Percentage of land use classified as urban</code>
<code>##</code>	<code>28</code>	<code>X28</code>	<code>Pct_Republican</code>	<code>Percentage of population registered republican</code>
<code>##</code>	<code>29</code>	<code>X29</code>	<code>Pct_Independent</code>	<code>Percentage of population not affiliated with any party</code>
<code>##</code>	<code>30</code>	<code>X30</code>	<code>Pct_Democrat</code>	<code>Percentage of population registered democrat</code>
<code>##</code>	<code>31</code>	<code>X31</code>	<code>Stay_at_Home_Star~</code>	<code>Date stay at home order issued</code>
<code>##</code>	<code>32</code>	<code>X32</code>	<code>Stay_at_Home_End_~</code>	<code>Date stay at home order scheduled to be lifted</code>
<code>##</code>	<code>33</code>	<code>1 Y01</code>	<code>cases</code>	<code>total number of covid-19 cases</code>
<code>##</code>	<code>34</code>	<code>2 Y02</code>	<code>deaths</code>	<code>total number of covid-19 deaths</code>
<code>##</code>	<code>35</code>	<code>3 Y03</code>	<code>death_rate</code>	<code>death rate as a percentage</code>
<code>##</code>	<code>36</code>	<code>4 Y04</code>	<code>severity_index</code>	<code>severity index based on death_rate: ## 1 = < ~</code>
<code># select some features and the PSI label</code>				
<code>my_x = c("X01", "X10", "X12", "X13", "X23")</code>				
<code>my_y = c("Y04")</code>				
<code>my_xy = c(my_x, my_y)</code>				
<code># get descriptions of the selected features and label</code>				

```

filter(mdf, Code %in% my_xy)
## # A tibble: 6 x 4
##   ID Code Variable      Description
##   <dbl> <chr> <chr>        <chr>
## 1 1 X01 Pct_Sun    Percent sunny days
## 2 10 X10 Unemployment_Rate ~ Percent unemployment
## 3 12 X12 Uninsured_Rate ~ Percent uninsured adults
## 4 13 X13 Heart_Disease_R ~ Deaths per 100000 due to heart disease
## 5 23 X23 Urban_Population ~ Percentage of population living in urban areas
## 6 4 Y04 severity_index severity index based on death_rate: ## 1 = < 0.1 ~

# subset the dataframe
rf_df = select(df, all_of(my_xy))

```

Now we'll add code to train a basic Random Forest model:

```

# split into train (75%) and test (25%) datasets
train = rf_df[seq(1,nrow(rf_df), by = 4),]
train = rbind(train, rf_df[seq(2,nrow(rf_df), by = 4),])
train = rbind(train, rf_df[seq(3,nrow(rf_df), by = 4),])
test = rf_df[seq(4,nrow(rf_df), by = 4),]

# create and train the RF model
model = randomForest(Y04 ~ X01 + X10 + X12 + X13 + X23,
                      data = train)
print(model) # show results, includes confusion matrix
##
## Call:
##   randomForest(formula = Y04 ~ X01 + X10 + X12 + X13 + X23, data = train)
##   Type of random forest: classification
##   Number of trees: 500
##   No. of variables tried at each split: 2
##
##   OOB estimate of error rate: 65.79%
## Confusion matrix:
##   3 4 5 6 7 8 class.error
## 3 0 0 1 0 0 0 1.0000000
## 4 0 0 1 1 0 0 1.0000000
## 5 0 0 7 8 0 0 0.5333333
## 6 0 1 8 6 0 0 0.6000000
## 7 0 0 2 2 0 0 1.0000000
## 8 0 0 0 1 0 0 1.0000000
importance(model) # measure of parameter importance
##   MeanDecreaseGini
## X01           3.932726

```

```
## X10      5.363756
## X12      5.609658
## X13      4.729598
## X23      5.170473
```

Taking a look at the results we see an error rate of >50%. The classification accuracy for the training dataset is 100% - `error rate` - so not a very good model. Further we can see some systematic failures in the confusion matrix. The most likely culprit is that the set of features is inadequate for making the desired prediction. We should re-run the model using different or additional features.

It's also good practice to examine the ROC and AUC metrics. For a problem with multiple classes (as opposed to a binary True/False or Yes/No problem) we can compute the ROC curve curve and AUC measures using a “one vs all” approach:

- First, extract predicted probabilities from the RF model (the scores).
- Next, extract actual classification for each category.
- Finally, leverage two commands of the `ROCR` module:
 - `prediction()`: retrieve scores
 - `performance()`: generates TPR, FPR, and AUC measure through two separate calls

Unfortunately, the calculation is fairly involved and uses a looping structure (i.e. a `for` loop) to assemble the plot and auc values:

```
library(ROCR)
## Loading required package: gplots
##
## Attaching package: 'gplots'
## The following object is masked from 'package:stats':
## 
##     lowess

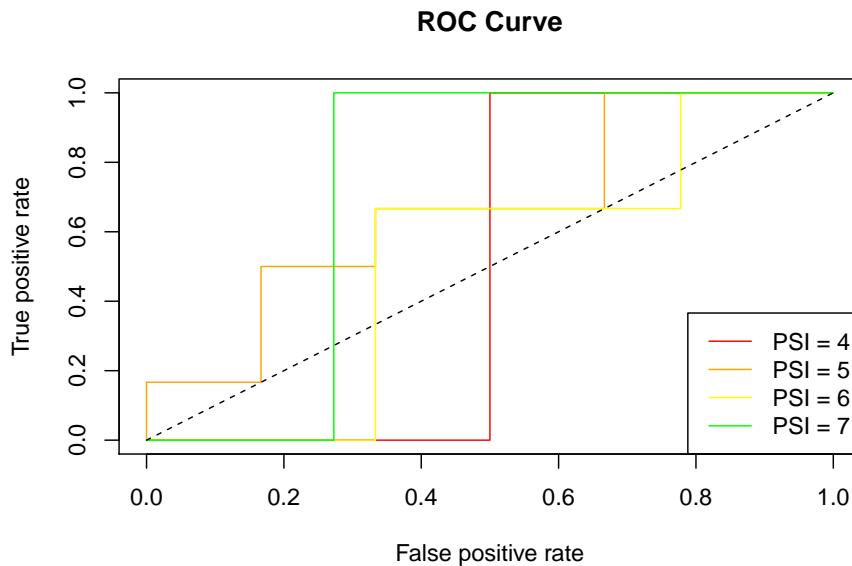
# compute ROC/AUC using test dataset
x_test = select(test, all_of(my_x))
y_test = select(test, all_of(my_y))
probs = predict(model, x_test, type='prob')
lvls = unique(as.character(y_test[[1]])) # category names
# up to 8 colors needed
cols = c("red","orange","yellow","green",
```

```
"blue","indigo","violet","black")
for(i in 1:length(lvls)) # for each category
{
  actual = as.numeric(y_test[[1]] == lvls[i])
  score = probs[,i]

  pred = prediction(score, actual)
  perf = performance(pred, "tpr", "fpr")
  if(i == 1){
    plot(perf, main="ROC Curve", col=cols[i])
  }
  else {
    plot(perf, main="ROC Curve", col=cols[i], add = TRUE)
  }

  # calculate the AUC and print
  auc = performance(pred, measure="auc")
  auc = as.numeric(auc@y.values)
  cat('AUC for PSI of ',lvls[i], ' = ', auc, '\n')
}

## AUC for PSI of 4 = 0.5
## AUC for PSI of 5 = 0.6666667
## AUC for PSI of 6 = 0.5185185
## AUC for PSI of 7 = 0.7272727
legend("bottomright",legend=paste("PSI =",lvls), col=cols, lty=1,
       cex=1.0) # adjust cex to scale the legend
lines(x=c(0,1),y=c(0,1),lty=2) # add threshold for non-informative
```



4.2 Day 23

For today's independent work you will learn about the Support Support Vector Machine (SVM) algorithm and apply it to the radiomics data that you worked with on Monday.

4.3 Day 24

For today's independent work you will learn about the K-nearest neighbors (KNN) algorithm and apply it to the radiomics data.

4.4 Day 25

For today's independent work you will learn about artificial neural networks (ANN). You will attempt to add Keras and TensorFlow support to your R/RStudio installation. If this turns out to be too dofficult, you will learn some “remote computing” alternatives that you can use in lieu of a local *R/Kera/TensorFlow* installation. Finally, you'll apply *R-Keras-TensorFlow* to the radiomics example that you've been working with during the week.

4.5 Day 26 (Friday) Zoom check-in

4.5.1 Review and trouble shoot (25 minutes)

4.5.2 Next week (25 minutes)

4.6 Day 27

4.7 Day 28

Self-directed activities.

Chapter 5

Bioinformatics with Bioconductor

5.1 Day 29 (Monday) Zoom check-in

5.2 Day 30

5.3 Day 31

5.4 Day 32

5.5 Day 33 (Friday) Zoom check-in

5.5.1 Review and trouble shoot (25 minutes)

5.5.2 Next week (25 minutes)

5.6 Day 34

5.7 Day 35

Self-directed activities.

Chapter 6

Collaboration

6.1 5 Days (Monday) Zoom check-in

6.2 4 Days

6.3 3 Days

6.4 2 Days

6.5 Today! (Friday) Zoom check-in

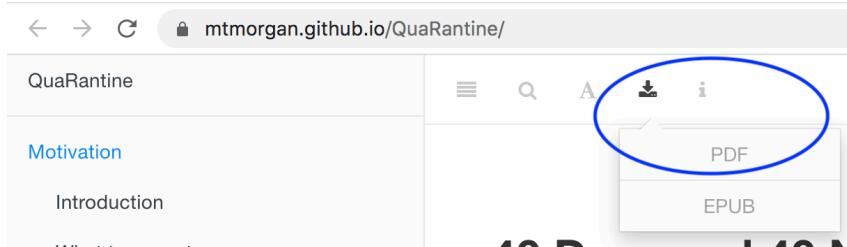
Course review and next steps

Frequently asked questions

1. Is the course material available in PDF?

Yes, click the ‘Download’ icon and PDF format in the title bar of the main document, as illustrated in the figure.

Remember that the course material is a ‘work in progress’, so the PDF will need to be updated frequently throughout the course. Also, the book is not pretty; that’s a task for a separate quarantine!



2. Whenever I press the ‘enter’ key, the RStudio console keeps saying + and doesn’t evaluate my expression! See the figure below.

A screenshot of the RStudio Console. The tabs at the top are 'Console', 'Terminal', 'R Markdown', and 'Jobs'. The current tab is 'Console'. The URL in the address bar is '~ /a/github/QuaRantine/'. The console window shows several lines of code input:

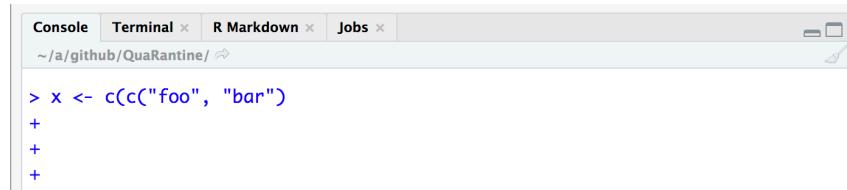
```
> x = "something"  
+  
+  
+  
+ sdfada  
+  
+  
+ ss  
+
```

The '+' signs appear at the end of each line of input, suggesting the user has not yet completed the character string 'something'.

Notice that you’ve started a character string with a double ", and tried to terminate it with a single quote '. Because the quotes do not match, R thinks you’re still trying to complete the entry of the variable, and

it's letting you know that it is expecting more with the + prompt at the begining of the line.

A common variant of this is to open more parentheses than you close, as shown in



The screenshot shows the RStudio interface with the 'Console' tab selected. The current working directory is shown as ~a/github/QuaRantine/. The console window displays the following text:

```
> x <- c("foo", "bar")
+
+
+
+
```

The text ends with four '+' symbols, indicating that the R interpreter is waiting for more input to balance the parentheses.

The solution is either to complete your entry (by entering a " or balancing the parentheses with)) or abandon your attempt by pressing **control-C** or the escape key (usually in the top left corner of the keyboard)

3. Should I save scripts, individual objects (`saveRDS()`) or multiple objects / the entire workspace (`save()`, `save.image()`, `quit(save = "yes")`)?

Reproducible research requires that one knows *exactly* how data was transformed, so writing and saving a script should be considered an essential ‘best practice’.

A typical script starts with some data generated by some third-party process, e.g., by entry into a spreadsheet or generated by an experiment. Often it makes sense to transform this through a series of steps to a natural ‘way-point’. As a final step in the script, it might make sense to save the transformed object (e.g., a `data.frame`) using `saveRDS()`, but making sure that the file name is unambiguous, e.g., matching the name of the object in the script, and with a creation date stamp.

I can't really imagine a situation when it would be good to use `save.image()` or `quit(save = "yes")` – I'll just end up with a bunch of objects whose content and provenance are completely forgotten in the mists of time (e.g., since yesterday!).

4. Where does RStudio create temporary files?

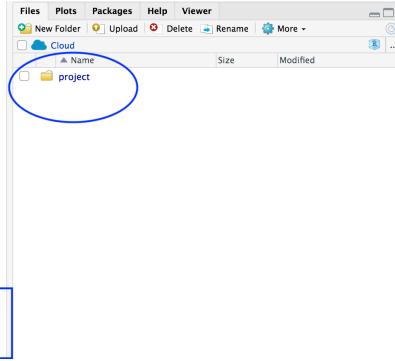
The screenshot below shows that the *R* session seems to have created a temporary file path, and it seems like it's possible to `write.csv()` / `read.csv()` to that file (no errors in the blue square box at the bottom left!) but the file doesn't show up in the file widget (circle in **Files** tab on the right).

```

'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> tempfile()
[1] '/tmp/RtmpvYph7S/filef625012737'
> activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
> minutes <- c(20, 30, 60, 60, 60)
> is.work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
>
> levels <- c("connect", "exercise", "consult", "hobby", "essential")
> classification <- factor(
+   c("connect", "essential", "connect", "consult", "exercise"),
+   levels = levels
+ )
>
> dates <- rep("04-14-2020", length(activity))
> date <- as.Date(dates, format = "%m-%d-%Y")
> activities <- data.frame(
+   activity, minutes, is.work, classification, date,
+   stringsAsFactors = FALSE
+ )
> temporary_file_path <- tempfile(fileext = ".csv")
> temporary_file_path
[1] '/tmp/RtmpvYph7S/filef6636094e.csv'
> write.csv(activities, temporary_file_path, row.names = FALSE)
> imported_activities <- read.csv(temporary_file_path, stringsAsFactors = FALSE)

```



The file widget is pointing to a particular directory; what you'd like to do is navigate to the directory where the temporary file is created. Do this by clicking on the three dots . . . (blue circle) in the **Files** tab, and enter the directory part of the temporary file path (blue squares).

```

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> tempfile()
[1] '/tmp/RtmpvYph7S/filef625012737'
> activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
> minutes <- c(20, 30, 60, 60, 60)
> is.work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
>
> levels <- c("connect", "exercise", "consult", "hobby", "essential")
> classification <- factor(
+   c("connect", "essential", "connect", "consult", "exercise"),
+   levels = levels
+ )
>
> dates <- rep("04-14-2020", length(activity))
> date <- as.Date(dates, format = "%m-%d-%Y")
> activities <- data.frame(
+   activity, minutes, is.work, classification, date,
+   stringsAsFactors = FALSE
+ )
> temporary_file_path <- tempfile(fileext = ".csv")
> temporary_file_path
[1] '/tmp/RtmpvYph7S/filef6636094e.csv'
> write.csv(activities, temporary_file_path, row.names = FALSE)
> imported_activities <- read.csv(temporary_file_path, stringsAsFactors = FALSE)

```

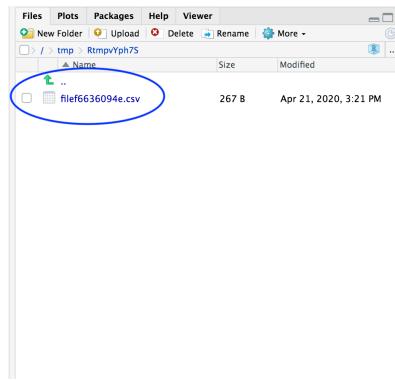
Once the file widget is pointed to the correct location, the file (last part of the **temporary_file_path**) appears...

```

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> tempfile()
[1] "/tmp/RtmpvYph7S/filef625012737"
> activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
> minutes <- c(20, 30, 60, 60, 60)
> is.work <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
>
> levels <- c("connect", "exercise", "consult", "hobby", "essential")
> classification <- factor(
+   c("connect", "essential", "connect", "consult", "exercise"),
+   levels = levels
+ )
>
> dates <- rep("04-14-2020", length(activity))
> date <- as.Date(dates, format = "%m-%d-%Y")
> activities <- data.frame(
+   activity, minutes, is.work, classification, date,
+   stringsAsFactors = FALSE
+ )
> temporary_file_path <- tempfile(fileext = ".csv")
> temporary_file_path
[1] "/tmp/RtmpvYph7S/filef6636094e.csv"
> write.csv(activities, temporary_file_path, row.names = FALSE)
> imported_activities <- read.csv(temporary_file_path, stringsAsFactors = FALSE)

```



Navigate back to the original directory by clicking on the three dots . . . in the **Files** tab and enter /cloud/project.

Remember that the temporary file path is, well, temporary, and when you start a new *R* session (or maybe restart your cloud session) the temporary path and anything saved there may no longer be available

5. When I try to install a package, *R* says a binary version exists but a newer version exists ‘from source’. Do I want to install the binary version or the source version?

Usually, the answer is to install the older binary version.

The newer source version likely represents a bug fix or implements new features in the package, so at first blush it seems preferable. However, installing source versions of packages often require additional software that is not necessarily easy to install and maintain. You might spend a considerable amount of time configuring your system and working through arcane error messages to get the ‘latest’ source version.

Almost always, the updated binary version becomes available in a day or so, and the best strategy is to install the old binary version now, and re-install the package the next time you remember, in a week or so.