40 Days and 40 Nights

Martin Morgan L. Shawn Matott

2020-04-20

Contents

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*.

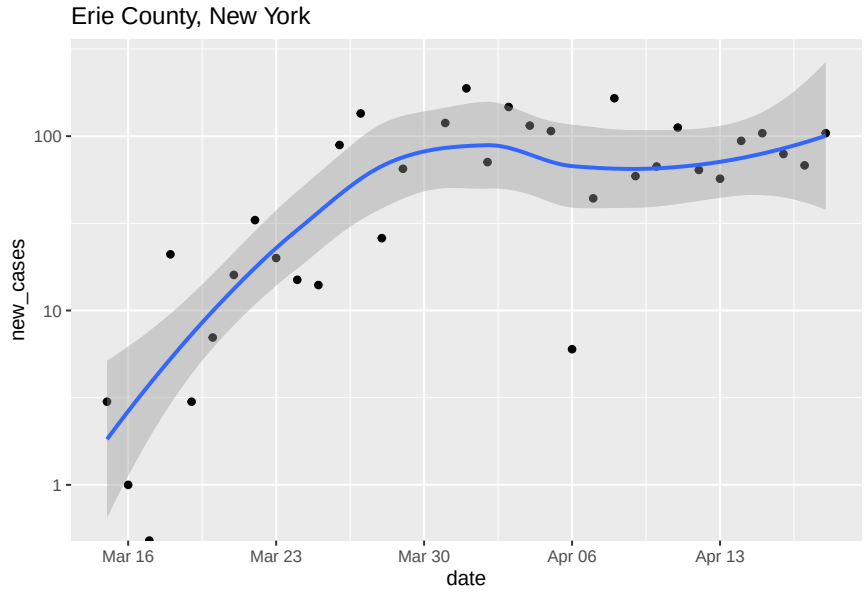
```
activity <- c("check e-mail", "breakfast", "conference call", "webinar", "walk")
minutes_per_activity <- c(20, 30, 60, 60, 60)
minutes_per_activity >= 60
## [1] FALSE FALSE  TRUE  TRUE  TRUE
activity[minutes_per_activity >= 60]
## [1] "conference call" "webinar"         "walk"
```

- 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.
cases <- read.csv(url)
erie <- subset(cases, county == "Erie" & state == "New York")
tail(erie)
##             date county    state  fips cases deaths
## 58128 2020-04-14   Erie New York 36029  1668     99
## 60844 2020-04-15   Erie New York 36029  1751    110
## 63572 2020-04-16   Erie New York 36029  1850    115
## 66316 2020-04-17   Erie New York 36029  1929    115
## 69074 2020-04-18   Erie New York 36029  1997    115
## 71842 2020-04-19   Erie New York 36029  2070    146
```

- 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: Maps. This week will be a specialized topic, tackling relatively advanced challenges associated with spatial visualization.
- 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?

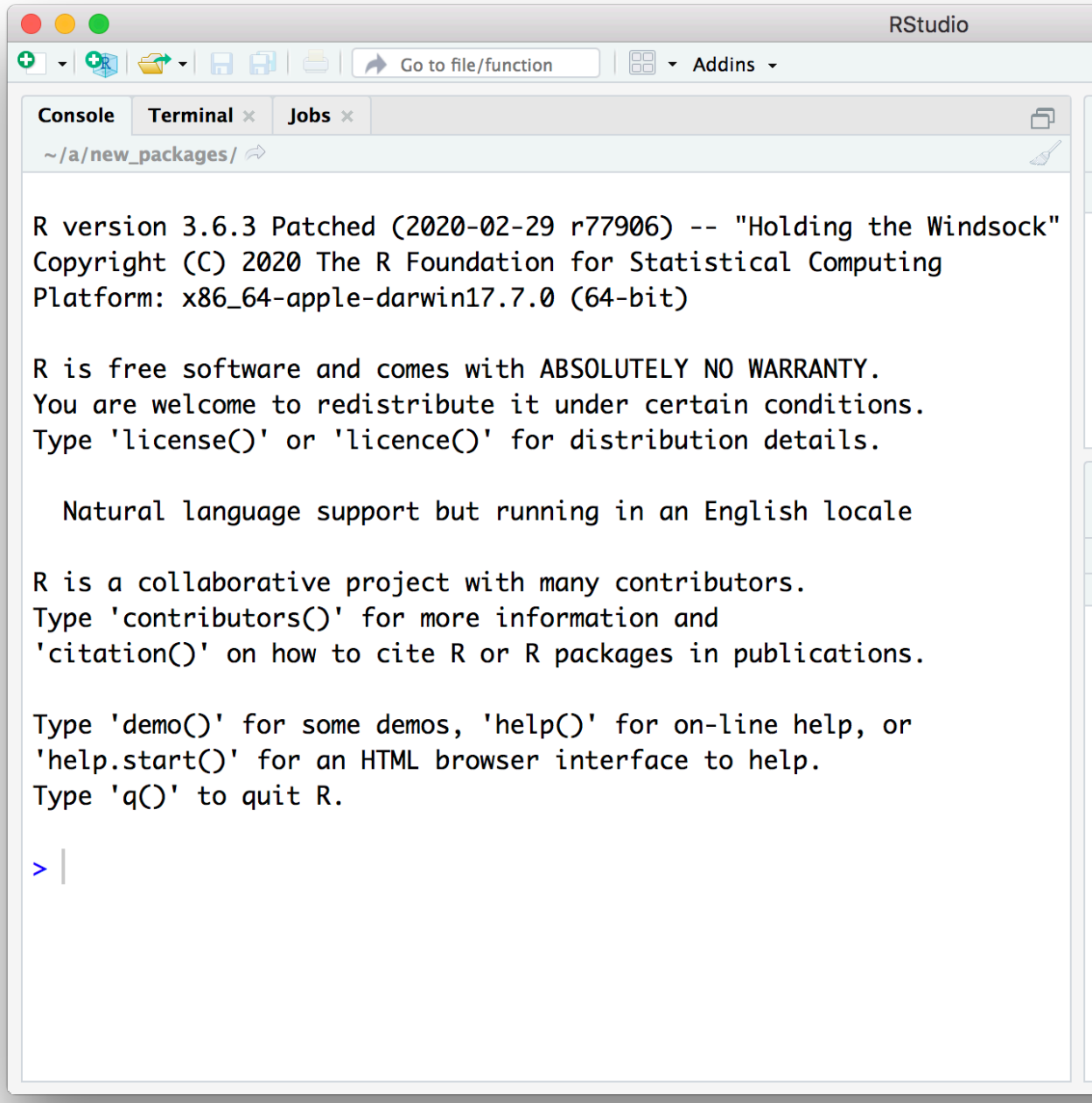
- RStudio provides an integrated editor and shell environment to make R programming easier. Some of the more useful features include:
  - Syntax highlighting and color coding
  - Easy switching between shell and editor
  - Dynamic help and docs

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

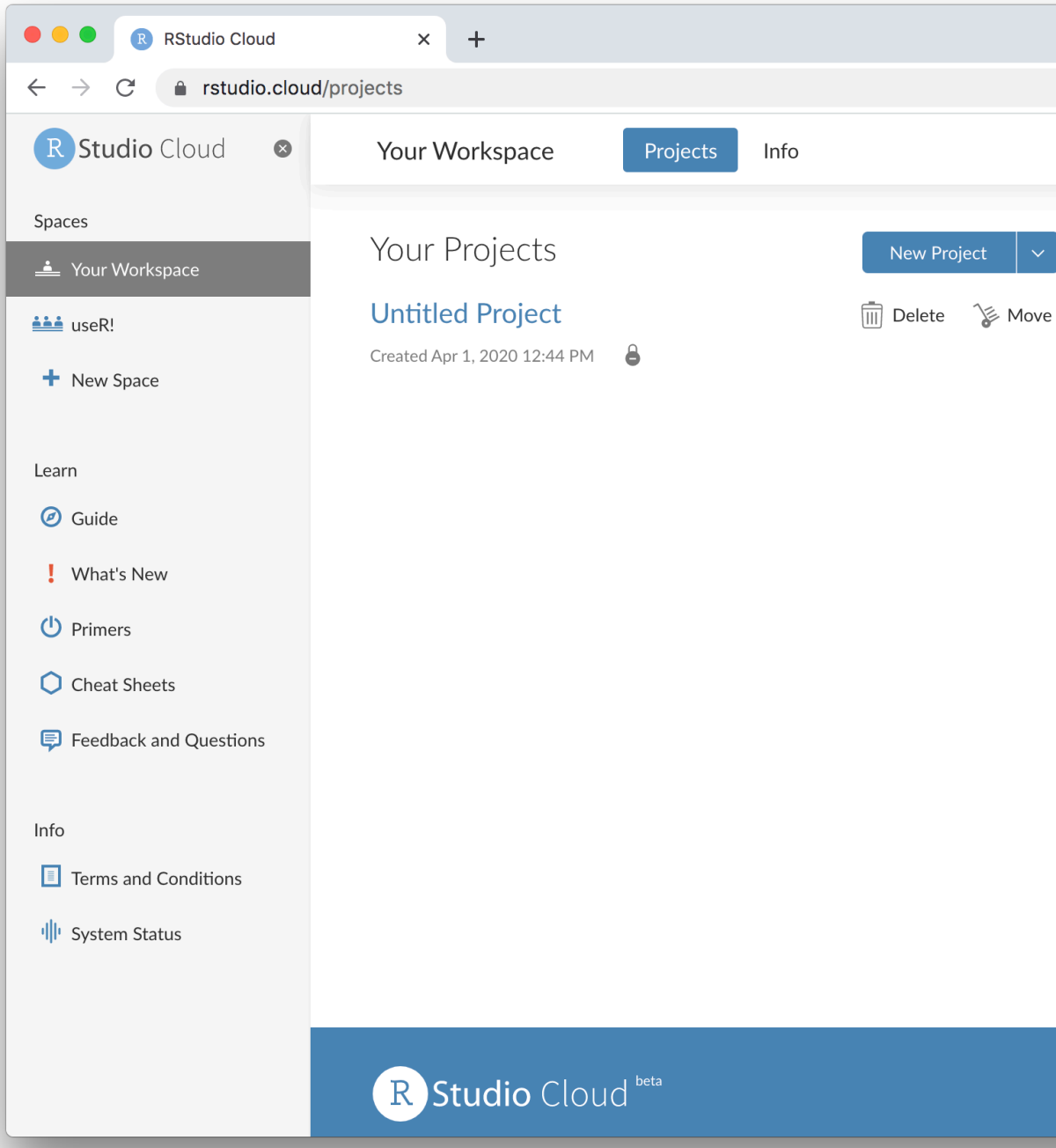
- Windows Users:
  - Download R for Windows and run the installer. Avoid, if possible, installing as administrator.
  - Download RStudio for Windows and run the installer.
  - Test the installation by launching RStudio. You should end up with a window like the screen shot below.
- Mac Users:
  - Download 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.


RStudio
Go to file/function
Addins
Console
Terminal
Jobs
~/a/new_packages/
R version 3.6.3 Patched (2020-02-29 r77906) -- "Holding the Windsock"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.7.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
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.
>

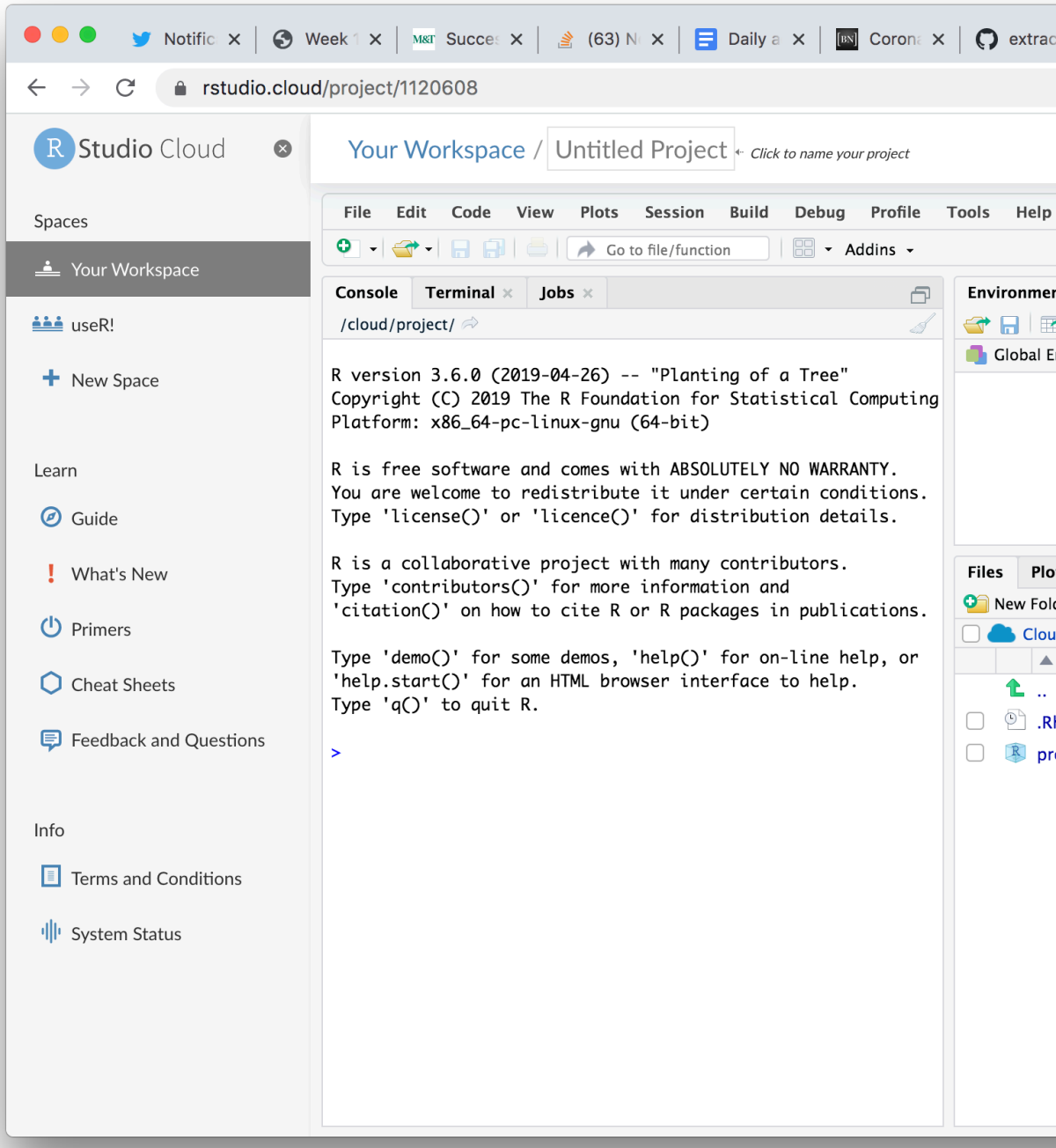
An ALTERNATIVE, if installing on your own computer does not work:

- 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.


RStudio Cloud
rstudio.cloud/projects
Studio Cloud
Spaces
Your Workspace
useR!
New Space
Learn
Guide
What's New
Primers
Cheat Sheets
Feedback and Questions
Info
Terms and Conditions
System Status
Your Workspace
Projects
Info
Your Projects
New Project
Untitled Project
Delete
Move
Created Apr 1, 2020 12:44 PM
Studio Cloud beta

- 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'.


rstudio.cloud/project/1120608
Studio Cloud
Your Workspace / Untitled Project
Click to name your project
Spaces
Your Workspace
useR!
New Space
Learn
Guide
What's New
Primers
Cheat Sheets
Feedback and Questions
Info
Terms and Conditions
System Status
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
/cloud/project/
R version 3.6.0 (2019-04-26) -- "Planting of a Tree"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
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.
>
Environmer
Global E
Files
Plo
New Fold
Clou

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 (`!=`):

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

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

```
18 > 44
## [1] FALSE
```

```
3 < 204
## [1] TRUE
```

Logical Or (|):

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

Logical And (&):

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

Objects and Vectors in R

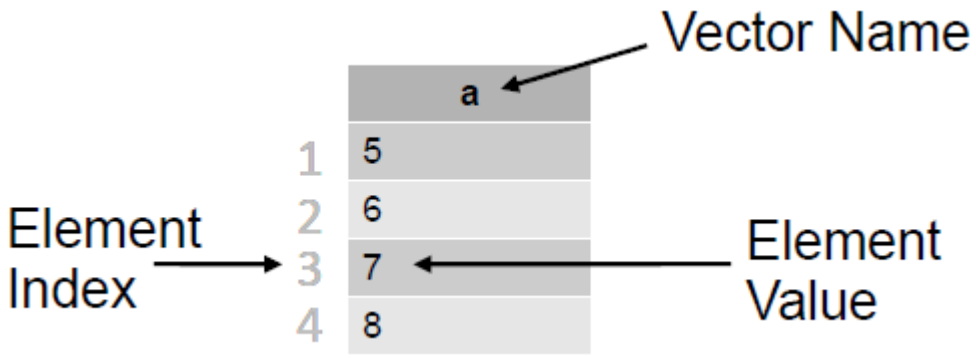
Objects

- 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 hesiate 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 `rep`licate 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.Data()` 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 ideaas 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()`, `sqrt()` (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-basaed 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 = 6892503,
```

```
    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
##     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 buton to open a dropdown menu

- In the dropdown menu, sectect `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 serveral weeks and collected lot's 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

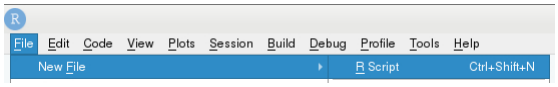
Wrapping up day 5

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 exstension (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 20, 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)

```
getcwd()
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 `.RDaata` 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] "/Users/ma38727/a/github/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 intereaction. 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 # referesh 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 spreadhseets

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/ma38737/MyQuarantine/df_all.csv"
write.csv(df_all, file, row.names = FALSE)
```

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

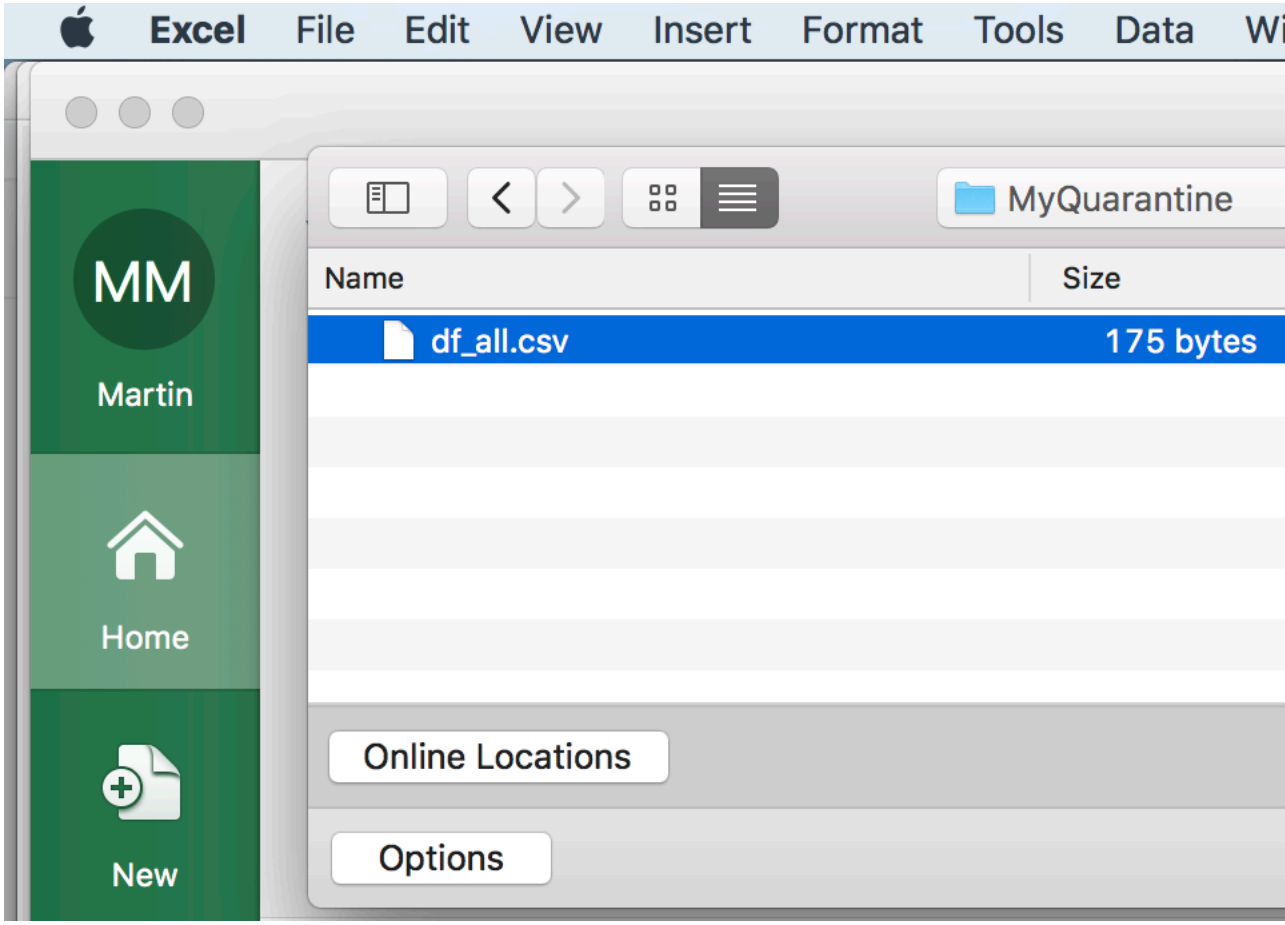
```
df_all_from_file <- read.csv(file, stringsAsFactors = FALSE)
df_all_from_file
##       person hgt_inches wgt_lbs age_years
## 1   John Doe         72     190        44
## 2  Pat Jones         65     130        35
## 3 Sara Grant         68     150        37
## 4   Bob Kane         61     101    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

R and spreadsheets

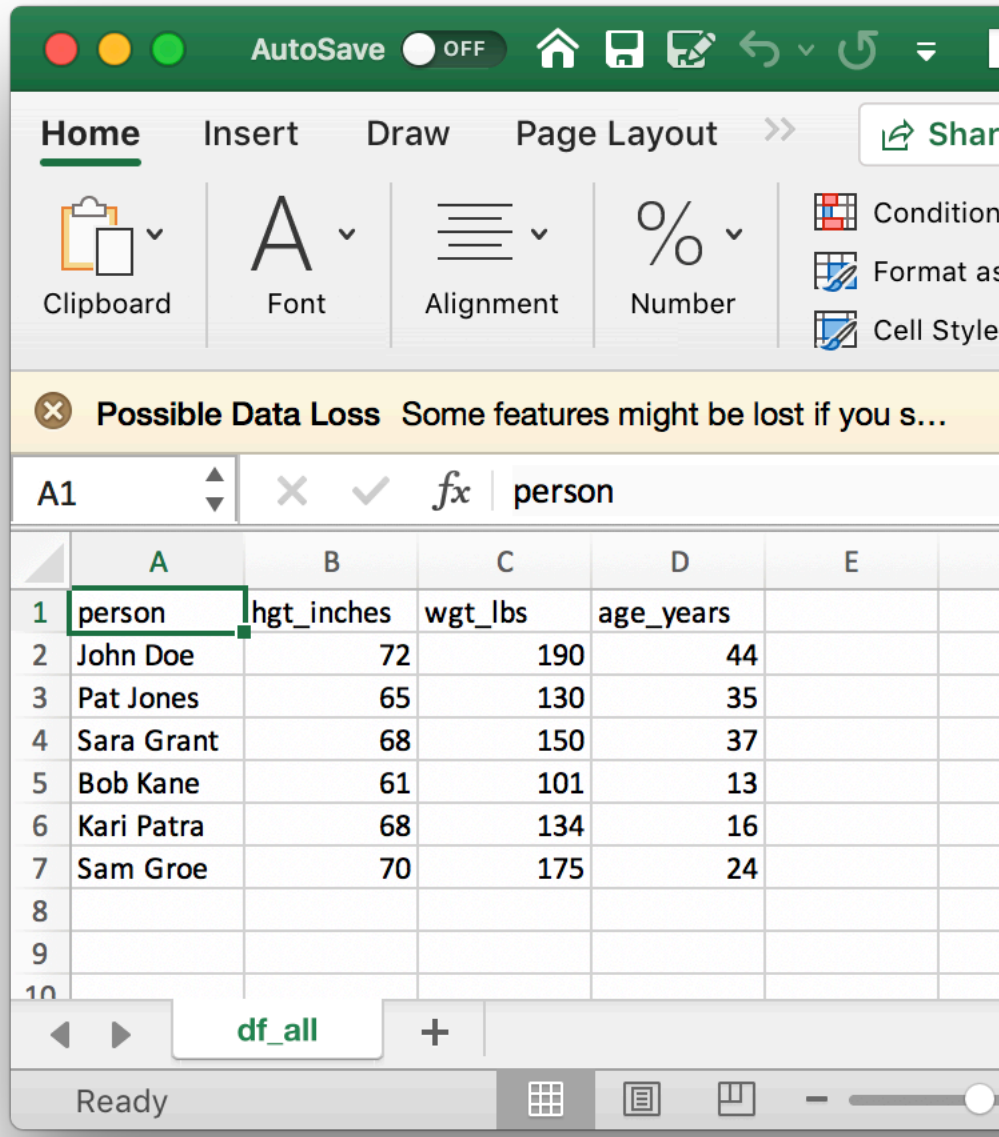
- A CSV file is a common way to move data from *R* to a spreadsheet, and vice versa. Following along with the example above, write `df_all` to a CSV file.

```
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

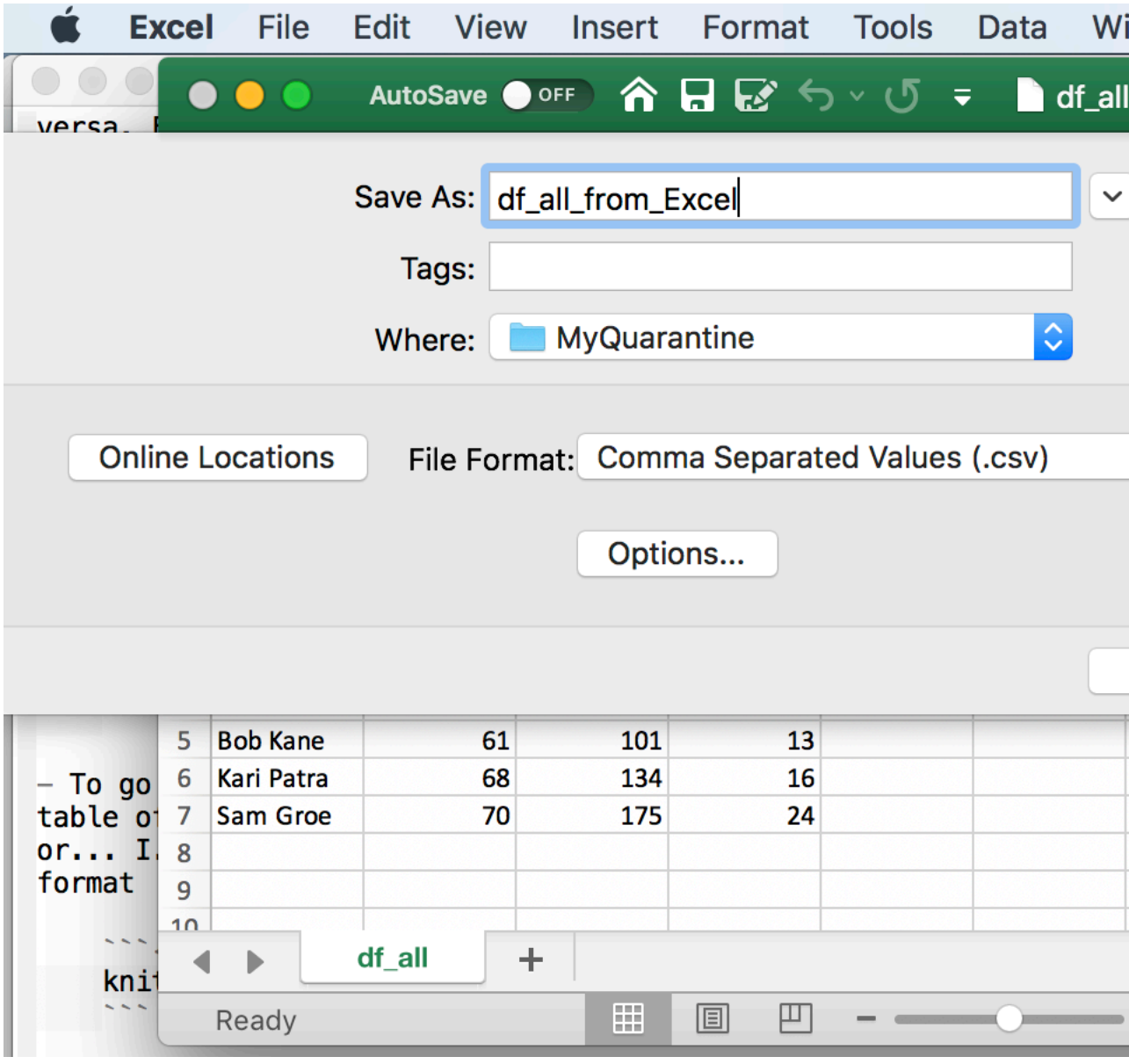


- … and open the file



	A	B	C	D	E
1	person	hgt_inches	wgt_lbs	age_years	
2	John Doe	72	190	44	
3	Pat Jones	65	130	35	
4	Sara Grant	68	150	37	
5	Bob Kane	61	101	13	
6	Kari Patra	68	134	16	
7	Sam Groe	70	175	24	
8					
9					
10					

- 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 formatting, 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*

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

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 clasasification
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 omiting 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)
## getcwd()
## 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] 72988     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] 1956    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
## 23079 2020-03-31   Erie New York 36029   438      8
## 25283 2020-04-01   Erie New York 36029   553     12
## 27544 2020-04-02   Erie New York 36029   734     19
## 29866 2020-04-03   Erie New York 36029   802     22
## 32254 2020-04-04   Erie New York 36029   945     26
## 34687 2020-04-05   Erie New York 36029  1059     27
## 37160 2020-04-06   Erie New York 36029  1163     30
## 39674 2020-04-07   Erie New York 36029  1163     36
## 42227 2020-04-08   Erie New York 36029  1205     38
## 44803 2020-04-09   Erie New York 36029  1362     46
## 47417 2020-04-10   Erie New York 36029  1409     58
## 50071 2020-04-11   Erie New York 36029  1472     62
## 52744 2020-04-12   Erie New York 36029  1571     75
## 55428 2020-04-13   Erie New York 36029  1624     86
## 58128 2020-04-14   Erie New York 36029  1668     99
## 60844 2020-04-15   Erie New York 36029  1751    110
## 63572 2020-04-16   Erie New York 36029  1850    115
## 66316 2020-04-17   Erie New York 36029  1929    115
## 69074 2020-04-18   Erie New York 36029  1997    115
## 71842 2020-04-19   Erie New York 36029  2070    146
```

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] 1956    6
tail(ny_state)
##             date      county    state  fips cases deaths
## 71881 2020-04-19      Warren New York 36113    96      6
## 71882 2020-04-19  Washington New York 36115    63      0
## 71883 2020-04-19       Wayne New York 36117    50      0
## 71884 2020-04-19 Westchester New York 36119 23803    831
## 71885 2020-04-19     Wyoming New York 36121    36      3
## 71886 2020-04-19       Yates New York 36123     9      0

erie <- subset(us, (state == "New York") & county == "Erie")
dim(erie)
## [1] 36  6
tail(erie)
##             date county    state  fips cases deaths
## 58128 2020-04-14   Erie New York 36029  1668     99
## 60844 2020-04-15   Erie New York 36029  1751    110
## 63572 2020-04-16   Erie New York 36029  1850    115
## 66316 2020-04-17   Erie New York 36029  1929    115
## 69074 2020-04-18   Erie New York 36029  1997    115
## 71842 2020-04-19   Erie New York 36029  2070    146
```

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] 2070
```

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] 57.5
```

`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.csv"
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   682
## 2    Allegany    30
## 3      Broome   192
## 4 Cattaraugus    34
## 5      Cayuga    36
## 6  Chautauqua    25
subset(
    max_cases_by_county,
    county %in% c("New York City", "Westchester", "Erie")
)
##           county  cases
## 14          Erie   2070
```

```
## 29 New York City 134446
## 57   Westchester  23803
```

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
us <- read.csv(url, stringsAsFactors = FALSE)
```

```
class(us)
## [1] "data.frame"
dim(us)
## [1] 72988     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] 36  6
```

- Subset, method 2

```
erie <- subset(us, (county == "Erie") & (state == "New York"))
dim(erie)
## [1] 36  6
```

Manipulation

- Goal: calculate `new_cases` as the difference between succesive 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 intial 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] 2070
with(erie, max(cases)) # ... another
## [1] 2070
```

- 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   682
## 2    Allegany    30
## 3      Broome   192
## 4 Cattaraugus    34
## 5      Cayuga    36
## 6  Chautauqua    25
```

- `subset()` to select counties

```
subset(
    max_cases_by_county,
    county %in% c("New York City", "Westchester", "Erie")
)
##           county  cases
## 14          Erie   2070
## 29 New York City 134446
## 57   Westchester  23803
```

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] 2801    3
head(max_cases_by_county_state)
##    county   state cases
## 1 Autauga Alabama    27
## 2 Baldwin Alabama   114
## 3 Barbour Alabama    20
## 4    Bibb Alabama    28
## 5  Blount Alabama    21
## 6 Bullock Alabama     9
```

- `aggregate()` a second time, using `max_cases_by_county_state` and aggregtaing 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   682
## 2     Alaska   154
## 3    Arizona  2589
## 4   Arkansas   377
## 5 California 12341
## 6   Colorado  1784
subset(
    max_cases_by_state,
    state %in% c("California", "Illinois", "New York", "Washington")
)
##         state  cases
## 5  California  12341
## 15   Illinois  21272
## 34   New York 134446
## 52 Washington   5174
```

2.5 Day 12 (Friday) Zoom check-in

Review and troubleshoot

```
## 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
## 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
## 6711  2020-03-21   Erie New York 36029    38      0         7
## 7805  2020-03-22   Erie New York 36029    54      0        16
## 9003  2020-03-23   Erie New York 36029    87      0        33
## 10314 2020-03-24   Erie New York 36029   107      0        20
## 11754 2020-03-25   Erie New York 36029   122      0        15
## 13367 2020-03-26   Erie New York 36029   134      2        12
## 15111 2020-03-27   Erie New York 36029   219      6        85
## 16951 2020-03-28   Erie New York 36029   354      6       135
## 18888 2020-03-29   Erie New York 36029   380      6        26
## 20938 2020-03-30   Erie New York 36029   443      8        63
## 23079 2020-03-31   Erie New York 36029   438      8        -5
## 25283 2020-04-01   Erie New York 36029   553     12       115
## 27544 2020-04-02   Erie New York 36029   734     19       181
## 29866 2020-04-03   Erie New York 36029   802     22        68
## 32254 2020-04-04   Erie New York 36029   945     26       143
## 34687 2020-04-05   Erie New York 36029  1059     27       114
## 37160 2020-04-06   Erie New York 36029  1163     30       104
## 39674 2020-04-07   Erie New York 36029  1163     36         0
## 42227 2020-04-08   Erie New York 36029  1205     38        42
## 44803 2020-04-09   Erie New York 36029  1362     46       157
## 47417 2020-04-10   Erie New York 36029  1409     58        47
```

```
## 50071 2020-04-11   Erie New York 36029  1472     62         63
## 52744 2020-04-12   Erie New York 36029  1571     75         99
## 55428 2020-04-13   Erie New York 36029  1624     86         53
## 58128 2020-04-14   Erie New York 36029  1668     99         44
## 60844 2020-04-15   Erie New York 36029  1751    110         83
## 63572 2020-04-16   Erie New York 36029  1850    115         99
## 66316 2020-04-17   Erie New York 36029  1929    115         79
## 69074 2020-04-18   Erie New York 36029  1997    115         68
## 71842 2020-04-19   Erie New York 36029  2070    146         73

## 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   682
## 2    Allegany    30
## 3      Broome   192
## 4 Cattaraugus    34
## 5      Cayuga    36
## 6  Chautauqua    25
```

Visualization and functions

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()`.

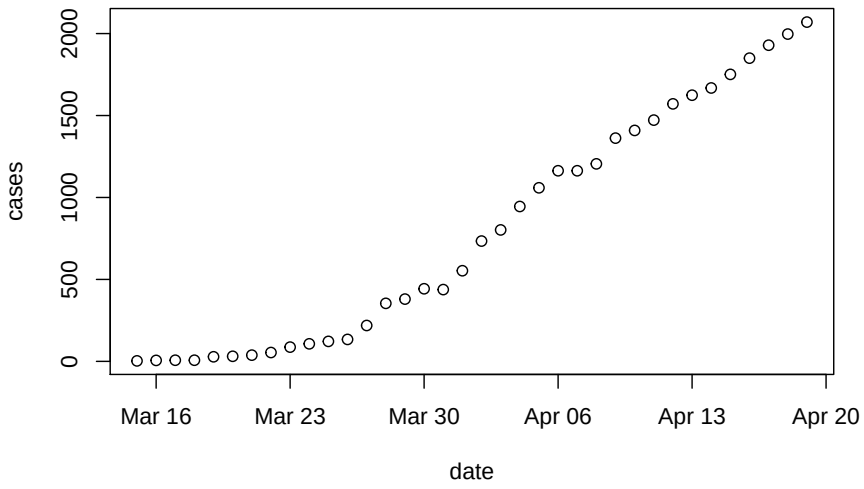
- The `formula` describes the independent (y-axis) variable as a function of the dependent (x-axis) variable

- 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.

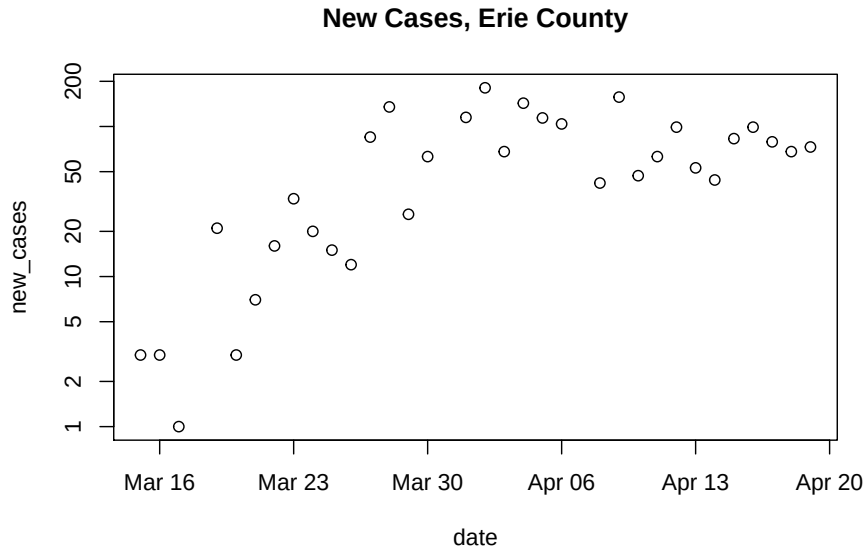
- Ok, here we go...

```
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

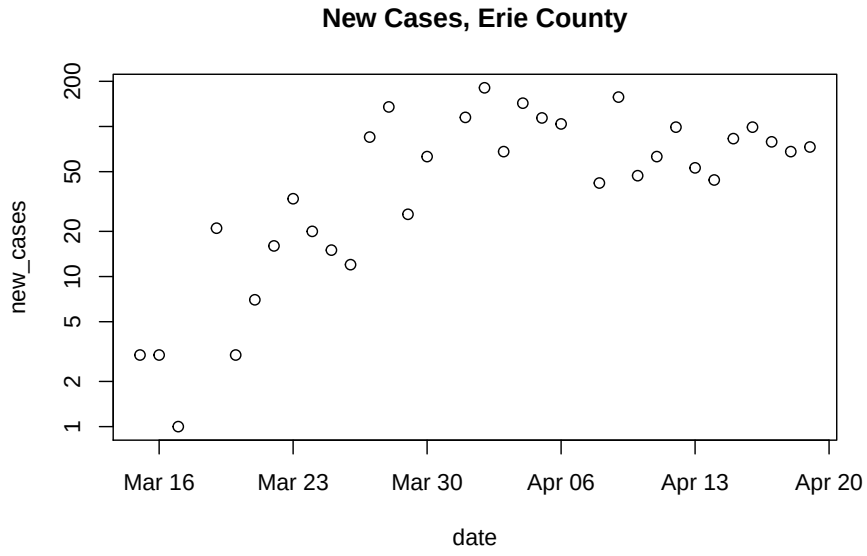
Yesterday we created a plot for Erie county. The steps to create this plot can be separated into two parts

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

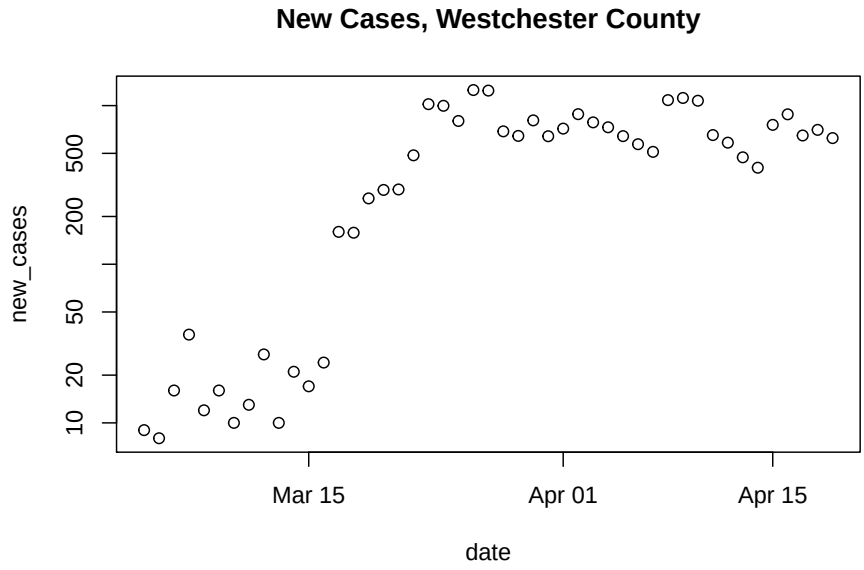
```
main_title <- paste("New Cases,", county_of_interest, "County")
```

- 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)
}
```

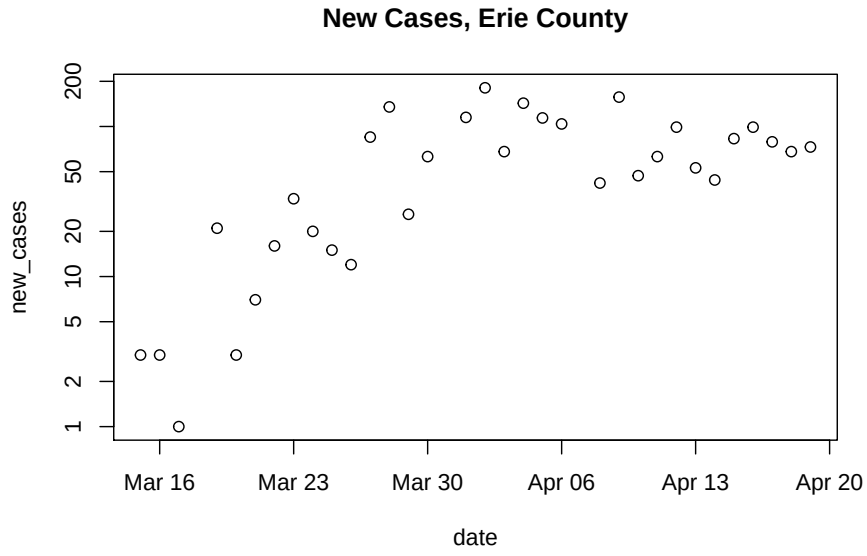
- Normally, the last evaluated line of the code (the `plot()` statement in our example) is returned from the function and can be captured by a variable. In our specific case, `plot()` creates the plot as a *side effect*, and the return value is actually the special symbol `NULL`.

- Here's the full definition

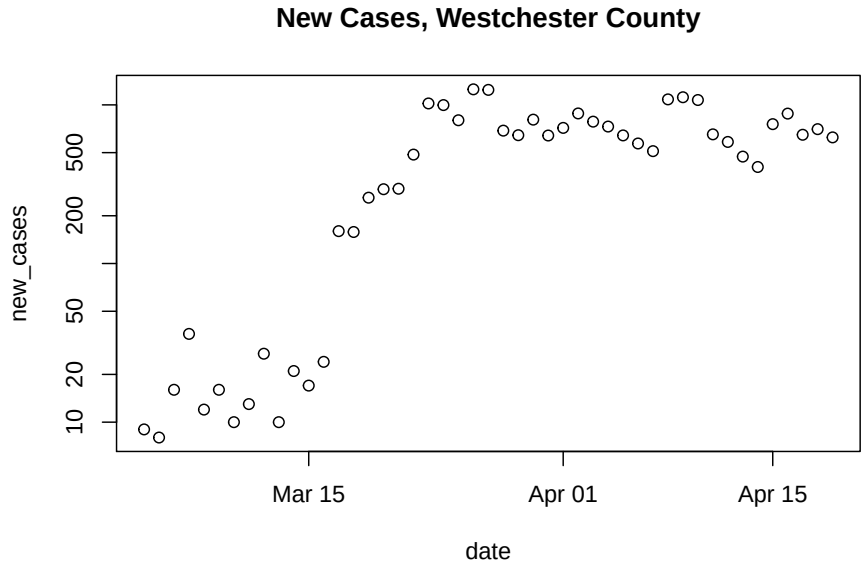
```
plot_county <-
    function(us_data, county_of_interest)
{
    main_title <- paste("New Cases,", county_of_interest, "County")
    county_data <-
        subset(us_data, (county == county_of_interest) & (state == "New York"))
    county_data <- within(county_data, {
        new_cases <- diff( c(0,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

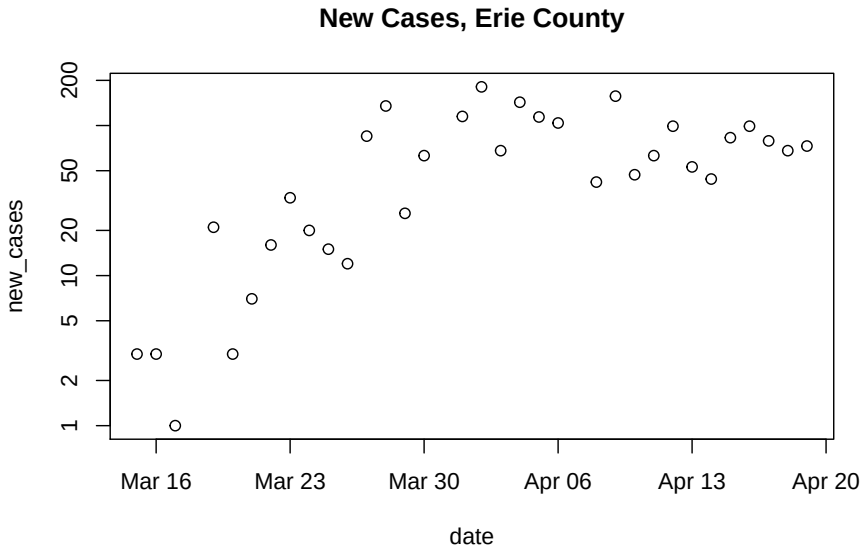
- Get and clean the US data; we don't need any arguments, and the return value (the last line of code evaluated) is the cleaned data

```
get_US_data <-
    function()
{
```

```
    url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-cou
    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

3.1.1 CRAN

3.1.2 The 'tidyverse' of packages

readr for fast data input

- The `tibble`: a nicer `data.frame`
- Example: US COVID data. N.B., `readr::read_csv()` rather than `read.csv()`

```
library(readr)

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: 72,988 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 72,978 more rows
```

- 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

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

- The pipe, `%>%`
- 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

```
us %>%
    filter(state == "New York", county == "Erie")
## # A tibble: 36 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 26 more rows
```

- `select()` specific columns

```
us %>%
    filter(state == "New York", county == "Erie") %>%
    select(state, county, date, cases)
## # A tibble: 36 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 26 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
- `arrange()` rows based on values in particular column(s); `desc()` in descending order.
- `count()` the number of times values occur

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
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: 72,988 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 72,978 more rows
```

- `filter()` rows that meet specific criteria

```
us %>%
    filter(state == "New York", county == "Erie")
## # A tibble: 36 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 26 more rows
```

- `select()` specific columns

```
us %>%
    filter(state == "New York", county == "Erie") %>%
    select(state, county, date, cases)
## # A tibble: 36 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 26 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: 36 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 26 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>               <dbl
## 1       36        2070                 181                 57.5                5
```

- `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: 2,801 x 3
```

```
## # Groups:   county [1,652]
##    county        state      total_cases
##    <chr>         <chr>            <dbl>
##  1 New York City New York        134446
##  2 Nassau        New York         30013
##  3 Suffolk       New York         26888
##  4 Westchester   New York         23803
##  5 Cook          Illinois         21272
##  6 Wayne         Michigan         13692
##  7 Bergen        New Jersey       12639
##  8 Los Angeles   California       12341
##  9 Hudson        New Jersey       10486
## 10 Essex         New Jersey       10304
## # ... with 2,791 more rows

us_state_cases %>%
    arrange(desc(total_cases))
## # A tibble: 55 x 2
##    state         total_cases
##    <chr>               <dbl>
##  1 New York           244908
##  2 New Jersey          89582
##  3 Massachusetts       38077
##  4 Pennsylvania        32994
##  5 California          31546
##  6 Michigan            31360
##  7 Illinois            30382
##  8 Florida             26315
##  9 Louisiana           23932
## 10 Texas               19451
## # ... with 45 more rows
```

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

```
us %>%
    count(county, state) %>%
    arrange(desc(n))
## # A tibble: 2,801 x 3
##    county        state            n
##    <chr>         <chr>        <int>
##  1 Snohomish     Washington      90
##  2 Cook          Illinois        87
##  3 Orange        California      86
##  4 Los Angeles   California      85
##  5 Maricopa      Arizona         85
##  6 Santa Clara   California      80
```

```
##  7 Suffolk       Massachusetts    79
##  8 San Francisco California       78
##  9 Dane          Wisconsin        75
## 10 San Diego     California       70
## # ... with 2,791 more rows
```

3.3 Day 17 Visualization with ggplot2

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_c
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.
- tidyr provides functions for manipulating a `tibble` into 'tidy' format.

- `tidyr::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 it's help page `?tidyr::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: 23,496 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 23,486 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 `County/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: 16,465 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 16,455 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 indesc()‘ending order

```
world %>%
    group_by(country) %>%
    summarize(n = max(cases)) %>%
    arrange(desc(n))
## # A tibble: 185 x 2
##    country             n
##    <chr>           <dbl>
##  1 US             759086
##  2 Spain          198674
##  3 Italy          178972
##  4 France         154097
##  5 Germany        145184
##  6 United Kingdom 121172
##  7 Turkey          86306
##  8 China           83805
##  9 Iran            82211
## 10 Russia          42853
## # ... with 175 more rows
```

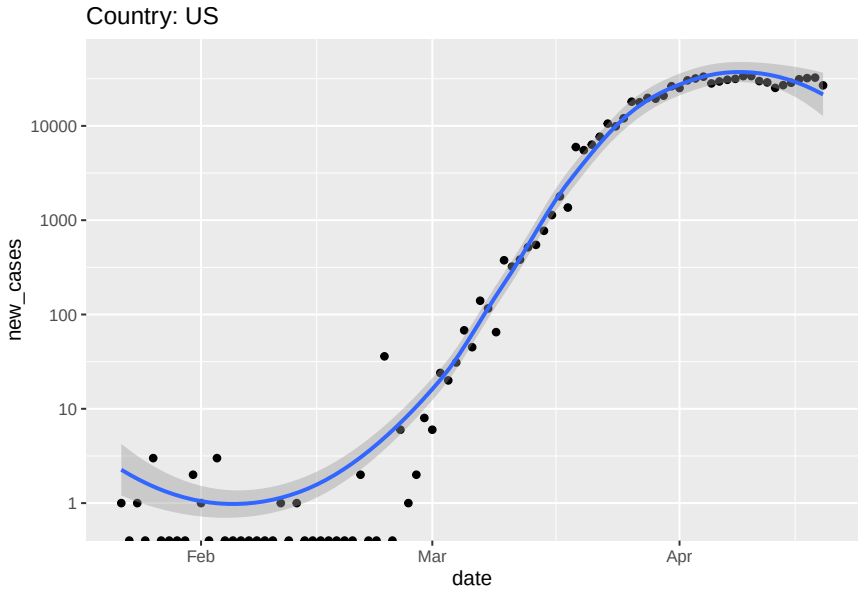
Visualization

- 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/cs
    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))
```

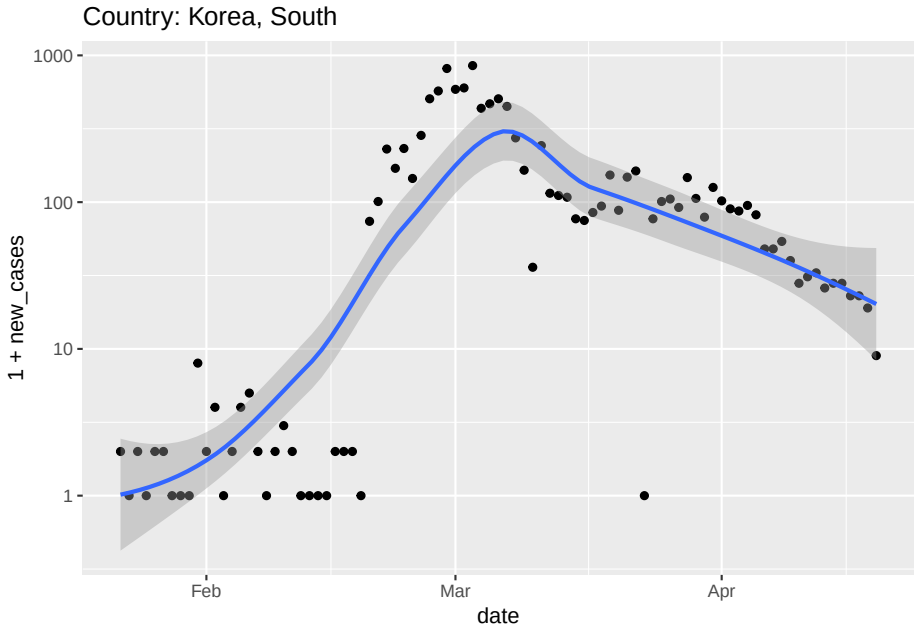
```
    ## add `new_cases`, and return the result
    world %>%
        group_by(country) %>%
        mutate(new_cases = diff(c(0, cases)))
}
```

- ...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)

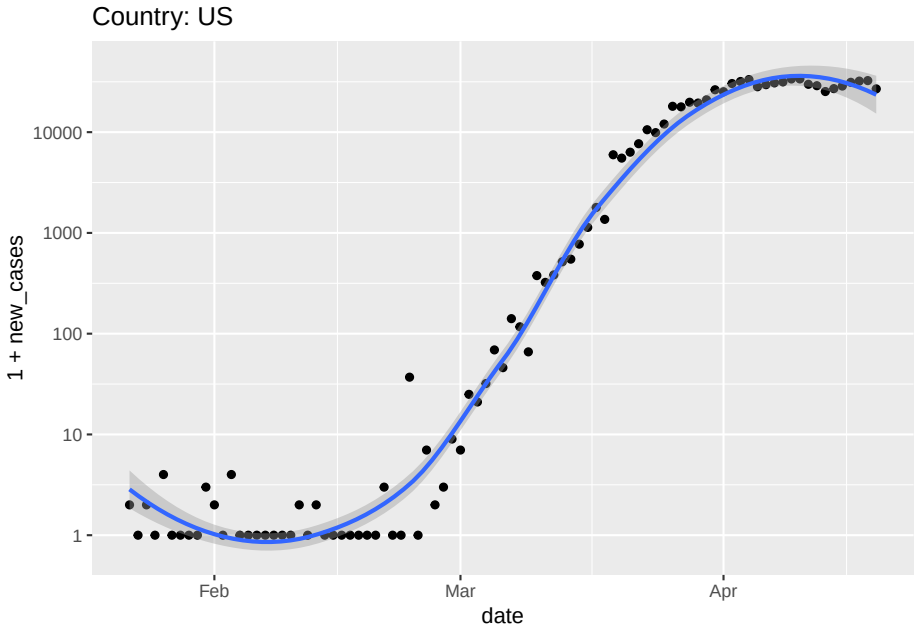
3.6 Day 20 Exploring the course of pandemic in different regions

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")
```



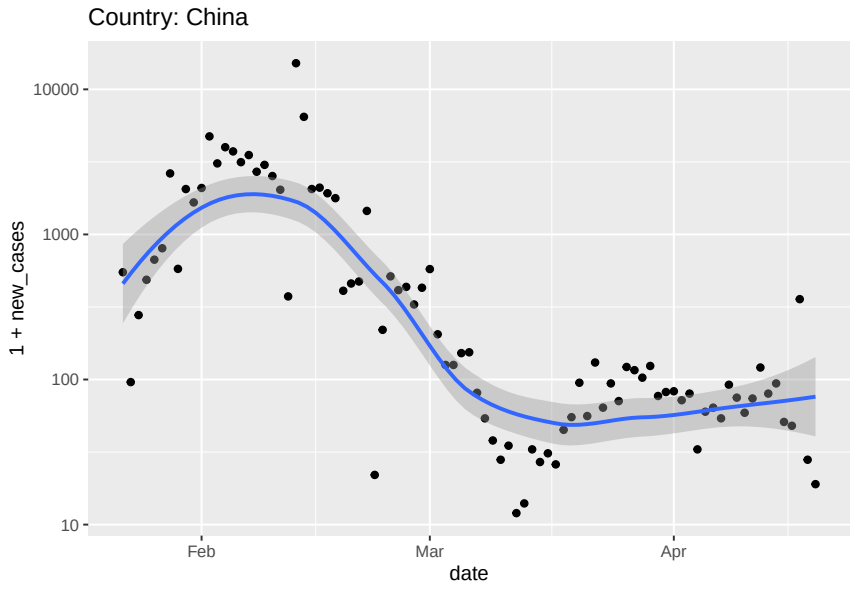
- When did 'stay at home' orders come into effect? Did they appear to be effective?

- 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.

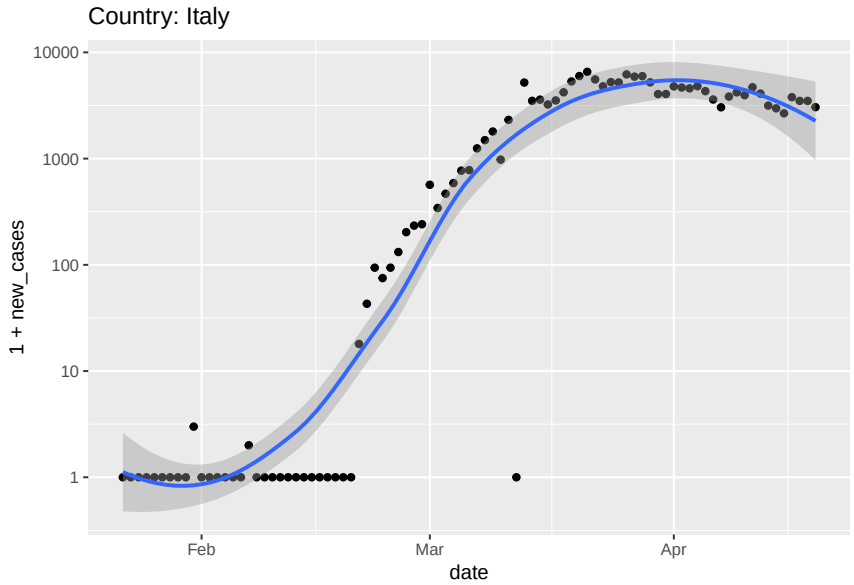
- The longest trajectory is probably displayed by China

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

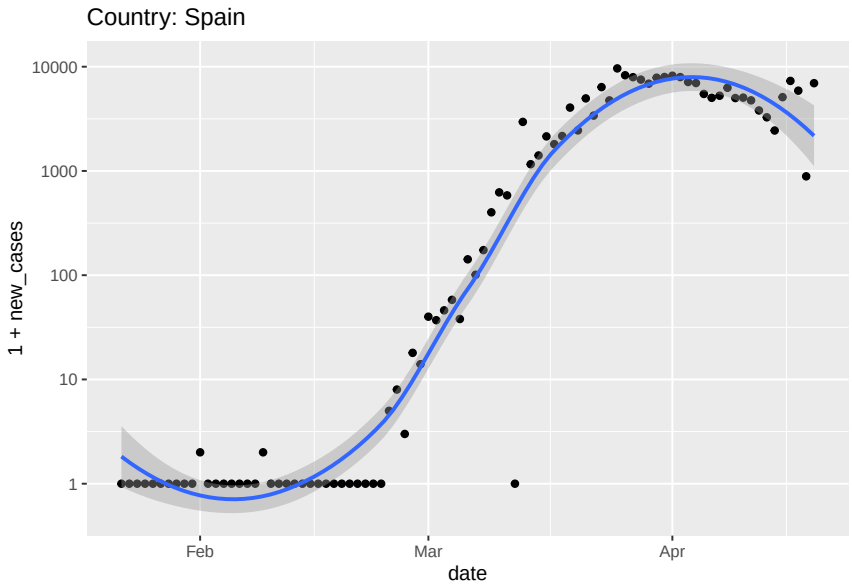


- Italy and Spain were hit very hard, and relatively early, by the pandemic

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

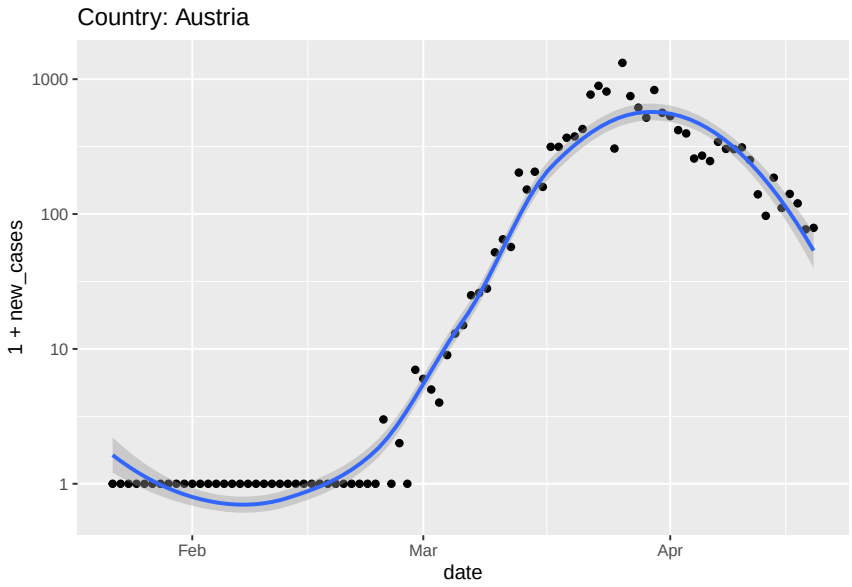



```
world %>% plot_country("Spain")
```



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

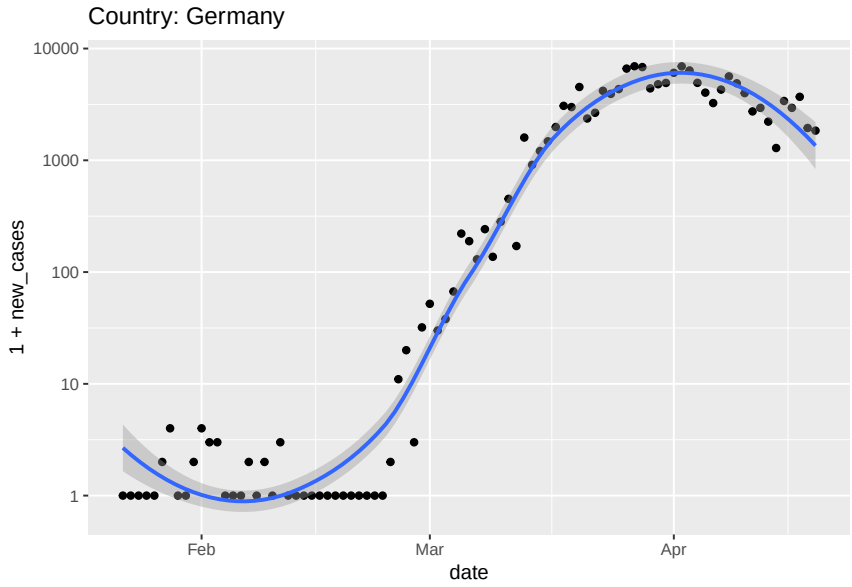
```
world %>% plot_country("Austria")
```



- Germany also had strong leadership (e.g., chancellor Angela Merkel provided clear and unambiguous rules for Germans to follow, and then self-isolated when she herself became infected) and an effective screening campaign (e.g., to make effective use of limited testing resources, in some

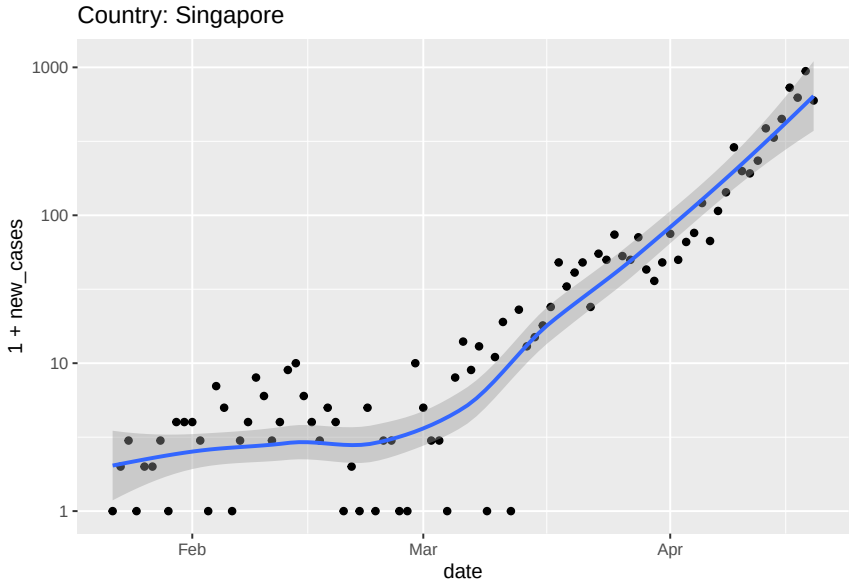
instances pools of samples were screened, and only if the pool indicated infection were the individuals in the pool screened.

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



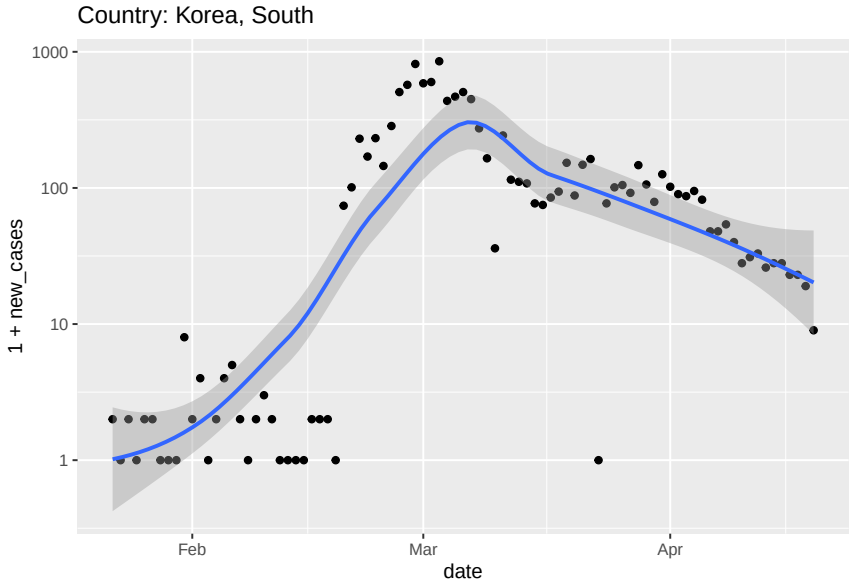
- At the start of the pandemic, Singapore had excellent surveillance (detecting individuals with symptoms) and contact tracing (identifying and placing in quarantine those individuals coming in contact with the infected individuals). New cases were initially very low, despite proximity to China, and 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

Maps and spatial statistics

4.1 Day 22 (Monday) Zoom check-in

4.2 Day 23

4.3 Day 24

4.4 Day 25

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

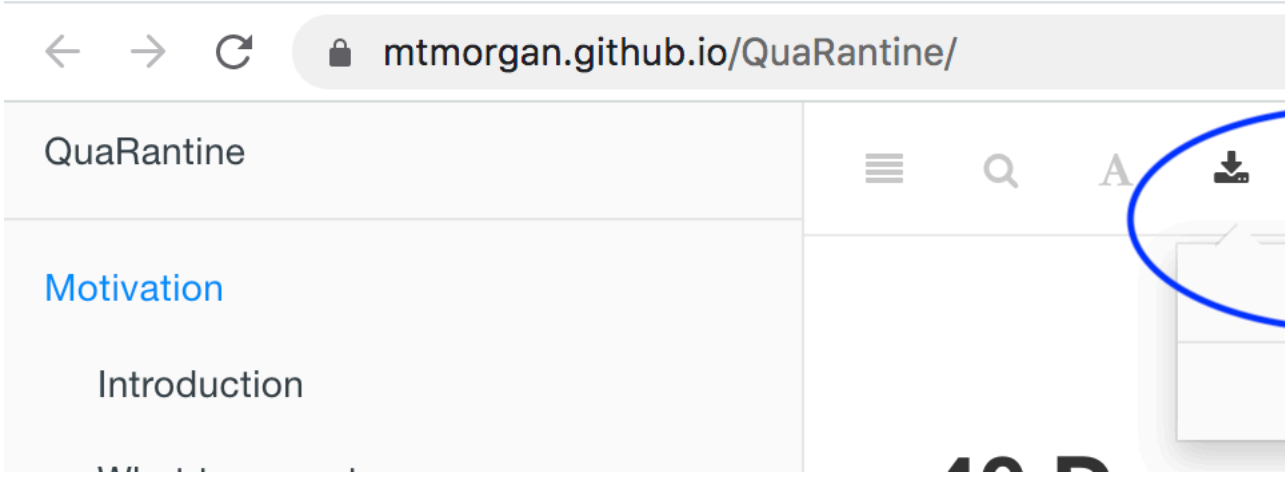
Course review and next steps

Frequently asked questions

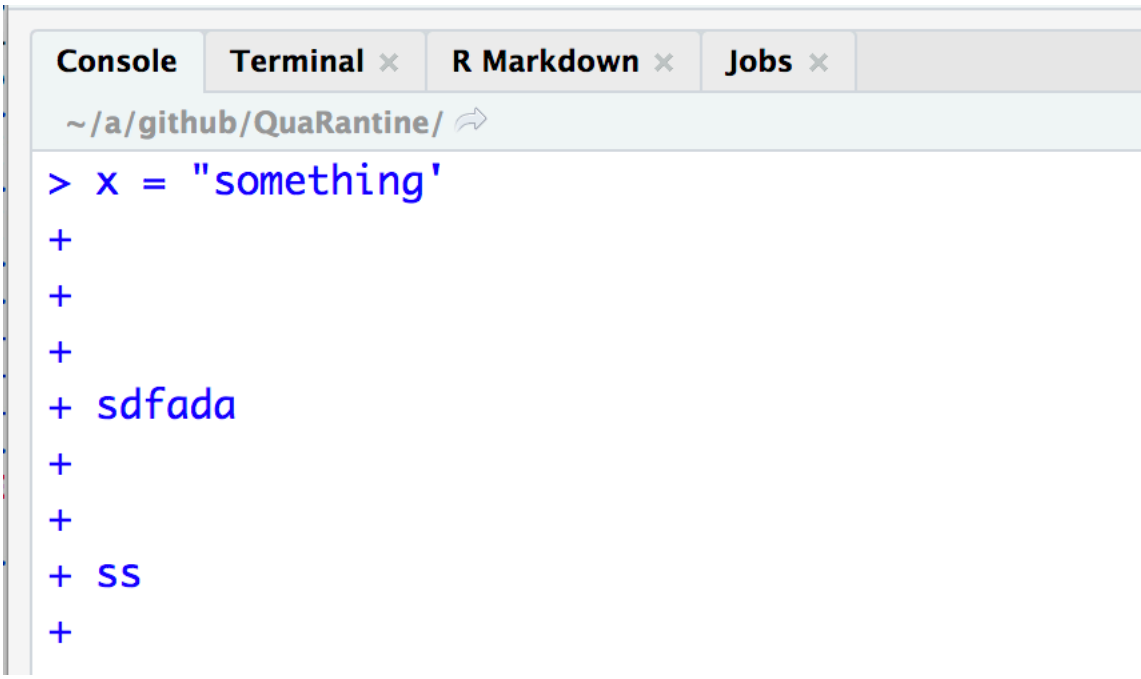
1. Is the course material available in PDF?

   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!

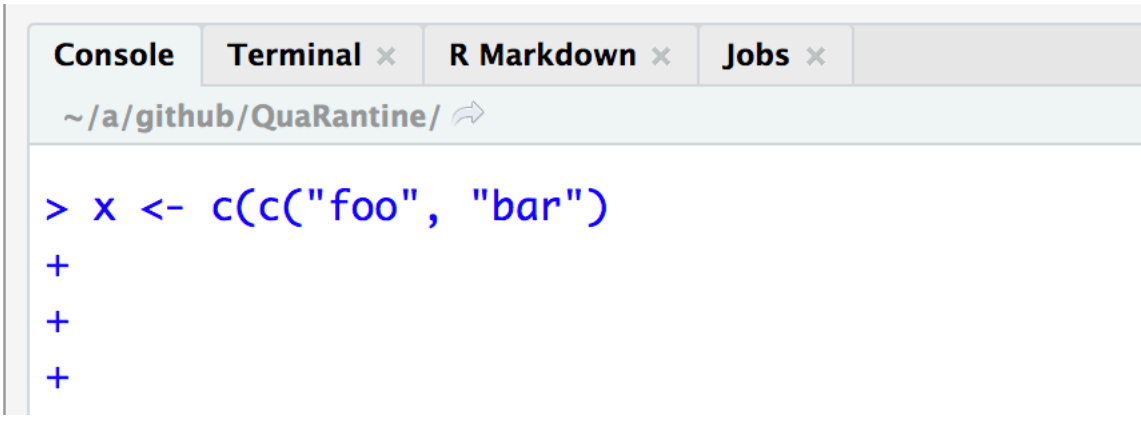


2. Whenever I press the 'enter' key, the RStudio console keeps saying + and doesn't evaluate my expression! See the figure below.



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 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!).