

Finding your way in R

Data Wrangling, Session 2

Kieran Healy

Code Horizons

January 2026

Writing documents

Use Quarto to
produce and
reproduce your
work

Where we want to end up

Covid Cases

Kieran Healy

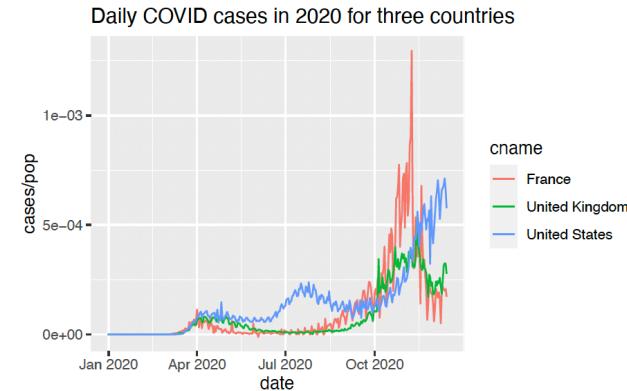
Introduction

We will look at some data from the `covdata` package.

cname	cases
France	2376852
United Kingdom	1849403
United States	16256754

A little graph

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.



PDF out

Where we want to end up

Covid Cases

AUTHOR
Kieran Healy

Introduction

We will look at some data from the `covdata` package.

cname	cases
France	2376852
United Kingdom	1849403
United States	16256754

A little graph

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Daily COVID cases in 2020 for three countries

The graph displays the daily COVID cases per capita for three countries in 2020. The y-axis represents the number of cases per capita on a logarithmic scale, ranging from 0e+00 to 1e-03. The x-axis represents the date from January 2020 to October 2020. France (red line) shows a significant spike in late October, reaching approximately 1e-03 cases per capita. The United States (blue line) and United Kingdom (green line) show more gradual increases, with the United States reaching approximately 5e-04 cases per capita by October.

HTML out

Where we want to end up

Covid Cases

Kieran Healy

Introduction

We will look at some data from the covdata package.

cname	cases
France	2376852
United Kingdom	1849403
United States	16256754

A little graph

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Daily COVID cases in 2020 for three countries

The graph displays the daily COVID cases per capita for three countries in 2020. The y-axis is logarithmic, indicating the scale of the data. The x-axis represents the date from January to December 2020. France (red line) shows the highest peak, reaching approximately 1e-03 cases per capita in early October. The United Kingdom (green line) and the United States (blue line) follow, both peaking around 5e-04 cases per capita in the same period. The data shows a general upward trend in cases throughout the year, with a major surge occurring in the fall of 2020.

Word out

How to get there?

```
# COVID      covidcases.R
# Get data from ECDC
# FIXME Write a fn to
# do this
data_raw <- read_csv(url)

# Clean it
# Notes on the cleaning
# process.

covid <- data_raw %>%
  mutate(...) %>%
  select(...)

# Make some plots
covid %>%
  ggplot(...) +
  geom_line(...)
```

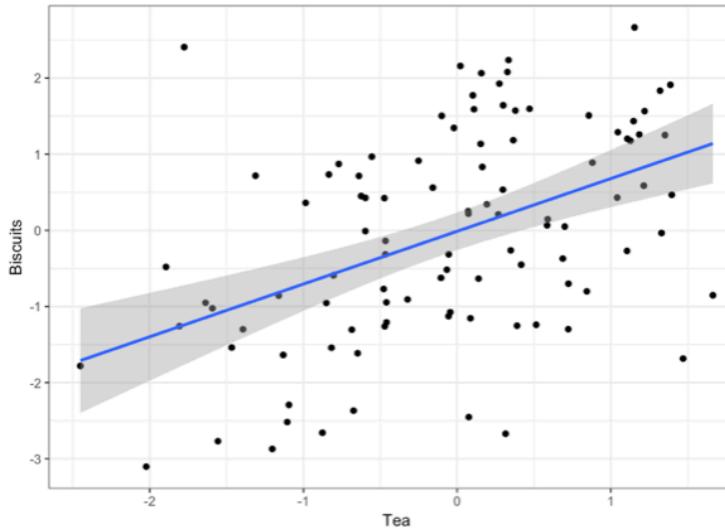
We could write an **R script** with some notes inside, using it to create some figures and tables, paste them into our document.

This will work, but we can do better.

We can **make** this ...

1. Lorem Ipsum

Lore*m* ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.



Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

... by writing this

Lorem Ipsum

 Lorem ipsum dolor sit amet, consectetur adipisicing elit,
 sed do eiusmod tempor incididunt ut labore et dolore magna
 aliqua. Ut enim ad minim veniam, quis nostrud exercitation
 ullamco laboris nisi ut aliquip ex ea commodo consequat.

```
library(ggplot2)
tea <- rnorm(100)
biscuits <- tea + rnorm(100, 0, 1.3)
data <- data.frame(tea, biscuits)
p <- ggplot(data, aes(x = tea, y = biscuits)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Tea", y = "Biscuits") + theme_bw()
print(p)
```

 Duis aute irure dolor in reprehenderit in voluptate velit esse
 cillum dolore eu fugiat nulla pariatur. Excepteur sint
 occaecat cupidatat non proident, sunt in culpa qui officia
 deserunt mollit anim id est laborum.

The code gets replaced by its output

Lorem Ipsum

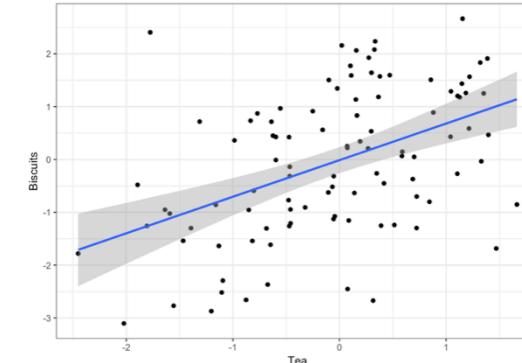
 Lorem ipsum dolor sit amet, consectetur adipisicing elit,
 sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

```
library(ggplot2)
tea <- rnorm(100)
biscuits <- tea + rnorm(100, 0, 1.3)
data <- data.frame(tea, biscuits)
p <- ggplot(data, aes(x = tea, y = biscuits)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Tea", y = "Biscuits") + theme_bw()
print(p)
```

Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.

1. Lorem Ipsum

 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.



Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.

```
---
```

```
title: "Covid Cases"
author: "Kieran Healy"
format: html
```

```
---
```

```
```{r}
#| label: setup
#| echo: false
#| message: false

Don't include code chunks in the document
knitr::opts_chunk$set (echo = FALSE)

library(tidyverse)
library(covdata)

```
## Introduction

We will look at some data from the `covdata` package.
```

```
```{r}
#| label: data

covnat_daily %>
 filter(iso3 %in% c("USA", "GBR", "FRA")) %>
 group_by(cname) %>
 summarize(cases = sum(cases)) %>
 knitr::kable()
```
```

Quarto document

Quarto document annotated

Header section with metadata

Chunks can have labels or options

Text with markdown formatting

When rendered, code chunks are replaced by their output

Code chunk (or cell)

Code chunks can be “played” one at a time

```
---
```

```
title: "Covid Cases"
author: "Kieran Healy"
format: html
---
```

```
```{r}
#| label: setup
#| echo: false
#| message: false

Don't include code chunks in the document
knitr::opts_chunk$set (echo = FALSE)

library(tidyverse)
library(covdata)

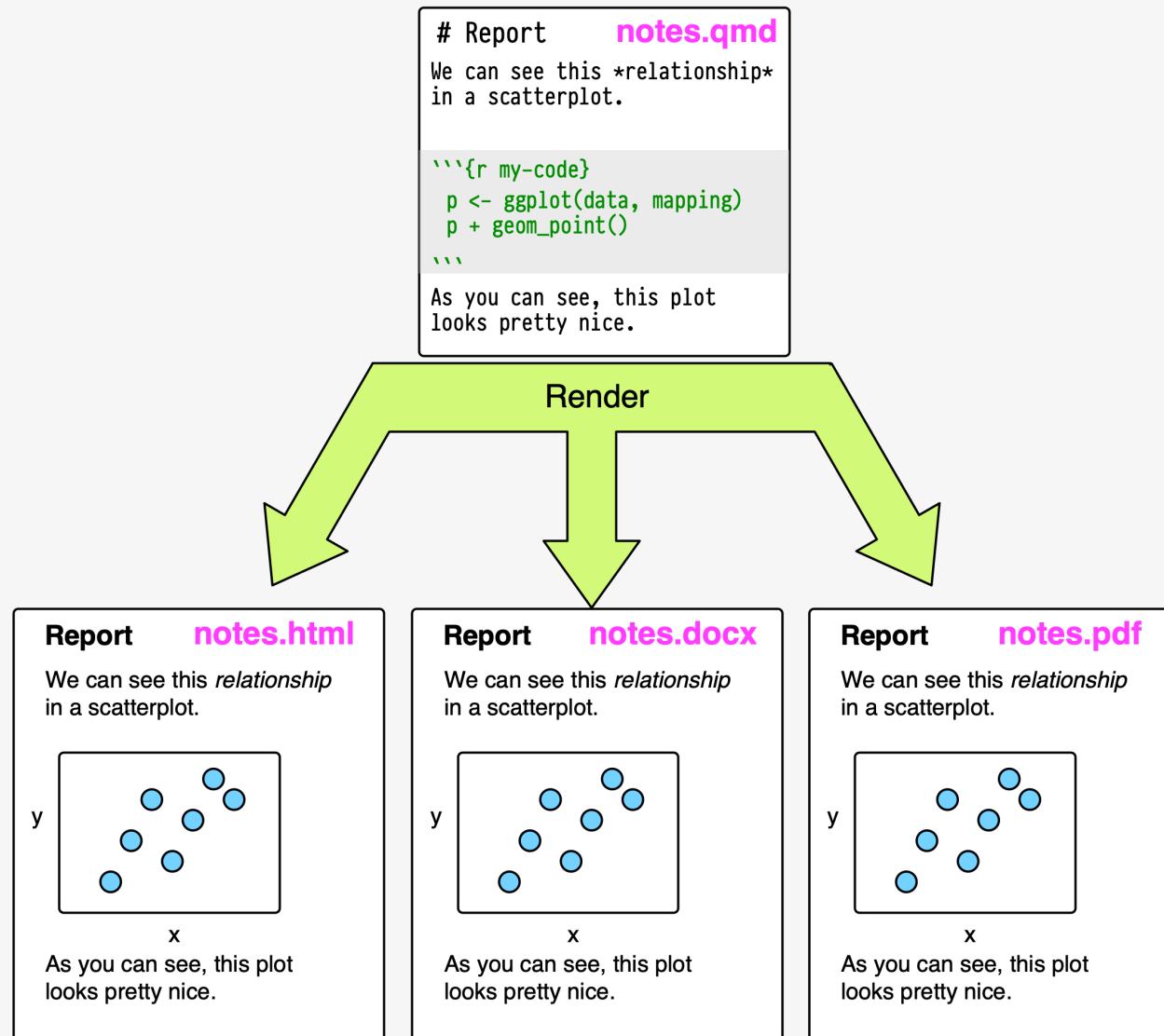
```
## Introduction

We will look at some data from the `covdata` package.

```{r}
#| label: data

covnat_daily %>
 filter(iso3 %in% c("USA", "GBR", "FRA")) %>
 group_by(cname) %>
 summarize(cases = sum(cases)) %>
 knitr::kable()
```

```



This approach has its limitations, but it's *very* useful and has many benefits.

Basic markdown summary

| To get this | You write this |
|----------------------------------|--|
| Heading 1 | # Heading 1 |
| Heading 2 | ## Heading 2 |
| Heading 3 | ### Heading 3 (Actual heading styles will vary.) |
| Paragraph | Just start typing |
| Bold | **Bold** |
| <i>Italic</i> | <i>*Italic*</i> |
| Images | [Alternate text for image](path/image.jpg) |
| Hyperlinks | [Link text](https://www.visualizingsociety.com/) |
| Unordered Lists | |
| - First | - First |
| - Second. | - Second |
| - Third | - Third |
| Ordered Lists | |
| 1. First | 1. First |
| 2. Second. | 2. Second |
| 3. Third | 3. Third |
| Footnote. ¹ | Footnote[^notelabel] |
| ¹ The note's content. | [^notelabel] The note's content. |

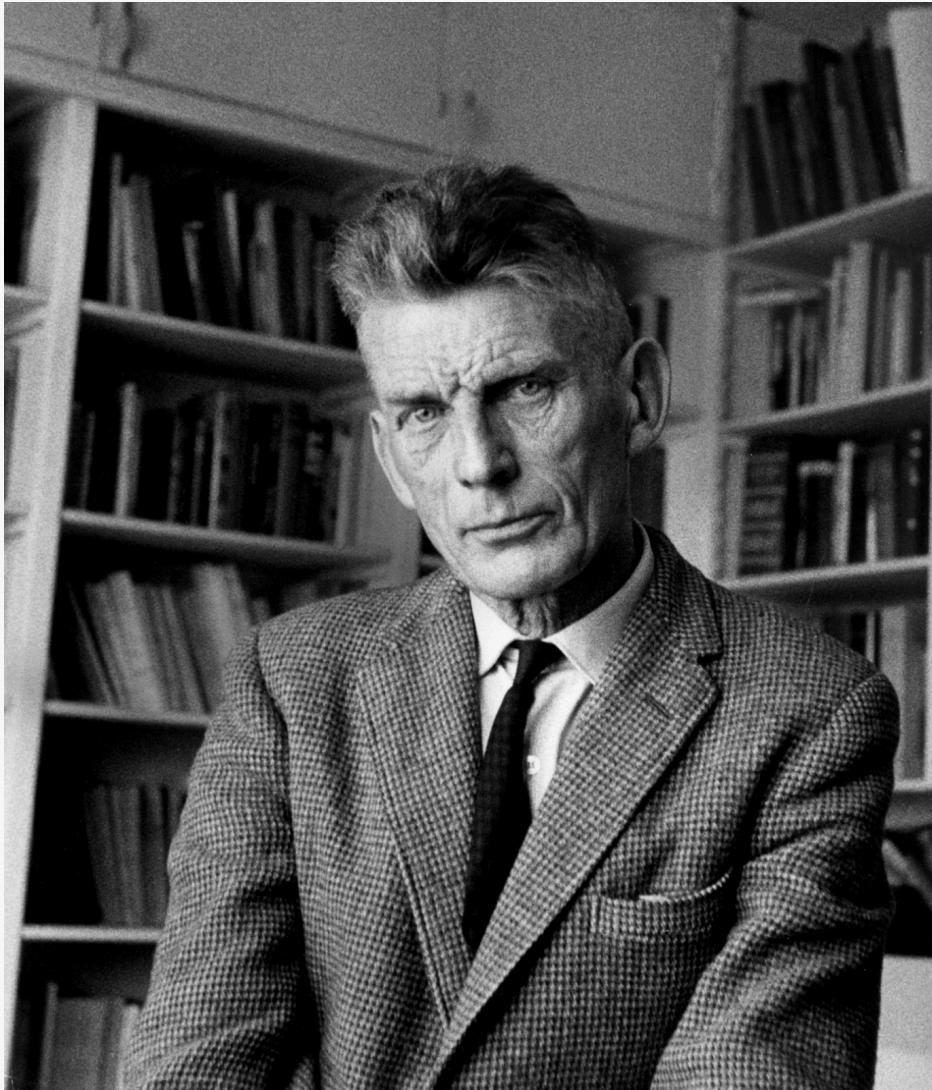
The right frame of mind

This is like learning how to drive a car, or how to cook in a kitchen ... or learning to speak a language.

After some orientation to what's where, you will learn best by *doing*.

Software is a pain, but you won't crash the car or burn your house down.

**TYPE OUT
YOUR CODE
BY HAND**



Samuel Beckett

Ever tried.
Ever failed.
No matter.
Try again.
Fail again.
Fail better.

Samuel Beckett,
early data analyst

GETTING ORIENTED

Loading the tidyverse libraries

```
library(tidyverse)
```

```
— Attaching core tidyverse packages ————— tidyverse 2.0.0 —
✓ dplyr     1.1.4    ✓ readr     2.1.5
✓forcats   1.0.0    ✓ stringr   1.5.1
✓ ggplot2   3.5.1    ✓ tibble    3.2.1
✓ lubridate 1.9.3    ✓ tidyrr    1.3.1
✓ purrr    1.0.2
— Conflicts ————— tidyverse_conflicts() —
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package to force all conflicts to become errors
```

The tidyverse has several components.

We'll return to this message about Conflicts later.

Tidyverse components

```
library(tidyverse)
```

Call the package and ...

```
Loading tidyverse: ggplot2
```

◁ Draw graphs

```
Loading tidyverse: tibble
```

◁ Nicer data tables

```
Loading tidyverse: tidyr
```

◁ Tidy your data

```
Loading tidyverse: readr
```

◁ Get data into R

```
Loading tidyverse: purrr
```

◁ Fancy Iteration

```
Loading tidyverse: dplyr
```

◁ Action verbs for tables

What R looks like

Code you can type and run:

```
## Inside code chunks, lines beginning with a # character are comments  
## Comments are ignored by R  
  
my_numbers ← c(1, 1, 2, 4, 1, 3, 1, 5) # Anything after a # character is ignored as well
```

Output:

```
my_numbers  
[1] 1 1 2 4 1 3 1 5
```

This is equivalent to running the code above, typing `my_numbers` at the console, and hitting enter.

What R looks like

By convention, code output in documents is prefixed by `##`

Also by convention, outputting vectors, etc, gets a counter keeping track of the number of elements. For example,

```
letters
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
[20] "t" "u" "v" "w" "x" "y" "z"
```

Some things to know about R

0. It's a calculator

Arithmetic

```
(31 * 12) / 2^4
```

```
[1] 23.25
```

```
sqrt(25)
```

```
[1] 5
```

```
log(100)
```

```
[1] 4.60517
```

```
log10(100)
```

```
[1] 2
```

0. It's a calculator

Arithmetic

```
(31 * 12) / 2^4
```

```
[1] 23.25
```

```
sqrt(25)
```

```
[1] 5
```

```
log(100)
```

```
[1] 4.60517
```

```
log10(100)
```

```
[1] 2
```

Logic

```
4 < 10
```

```
[1] TRUE
```

```
4 > 2 & 1 > 0.5 # The "&" means "and"
```

```
[1] TRUE
```

```
4 < 2 | 1 > 0.5 # The "/" means "or"
```

```
[1] TRUE
```

```
4 < 2 | 1 < 0.5
```

```
[1] FALSE
```

Boolean and Logical operators

Logical equality and inequality (yielding a `TRUE` or `FALSE` result) is done with `=` and `!=`. Other logical operators include `<`, `>`, `≤`, `≥`, and `!` for negation.

```
## A logical test  
2 = 2 # Write `=` twice
```

```
[1] TRUE
```

```
## This will cause an error, because R will think you are trying to assign a value  
2 = 2
```

```
## Error in 2 = 2 : invalid (do_set) left-hand side to assignment
```

```
3 != 7 # Write `!` and then `=` to make `!=`
```

```
[1] TRUE
```

1. Everything in R has a name

```
my_numbers # We created this a few minutes ago
```

```
[1] 1 1 2 4 1 3 1 5
```

```
letters # This one is built-in
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
pi # Also built-in
```

```
[1] 3.141593
```

Some names are forbidden

Or it's a *really* bad idea to try to use them

```
TRUE  
FALSE  
Inf  
NaN  
NA  
NULL
```

```
for  
if  
while  
break  
function
```

2. Everything is an object

There are a few built-in objects:

letters

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
[20] "t" "u" "v" "w" "x" "y" "z"
```

pi

```
[1] 3.141593
```

LETTERS

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"  
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

3. You can create objects

In fact, this is mostly what we will be doing.

Objects are created by **assigning** a thing to a name:

```
## name ... gets ... this stuff  
my_numbers ← c(1, 2, 3, 1, 3, 5, 25, 10)  
  
## name ... gets ... the output of the function `c()`  
your_numbers ← c(5, 31, 71, 1, 3, 21, 6, 52)
```

The **c()** function *combines* or *concatenates* things

The assignment operator

The assignment operator performs the action of creating objects

Use a keyboard shortcut to write it:

Press **option** and **-** on a Mac

Press **alt** and **-** on Windows

4. Do things to objects with functions

```
## this object... gets ... the output of this function  
my_numbers <- c(1, 2, 3, 1, 3, 5, 25, 10)
```

```
your_numbers <- c(5, 31, 71, 1, 3, 21, 6, 52)
```

```
my_numbers
```

```
[1] 1 2 3 1 3 5 25 10
```

4. Do things to objects with functions

Functions can be identified by the parentheses after their names.

```
my_numbers
```

```
[1] 1 2 3 1 3 5 25 10
```

```
## If you run this you'll get an error  
mean()
```

What functions usually do

They take **inputs** to **arguments**

They perform **actions**

They produce, or return, **outputs**

mean(x = my_numbers)

What functions usually do

They take **inputs** to **arguments**

They perform **actions**

They produce, or return, **outputs**

mean(x = my_numbers)

[1] **6.25**

What functions usually do

```
## Get the mean of what? Of x.  
## You need to tell the function what x is  
mean(x = my_numbers)
```

```
[1] 6.25
```

```
mean(x = your_numbers)
```

```
[1] 23.75
```

What functions usually do

If you don't *name* the arguments, R assumes you are providing them in the order the function expects.

```
mean(your_numbers)
```

```
[1] 23.75
```

What functions usually do

What arguments? Which order? Read the function's help page

```
help(mean)
```

```
## quicker  
?mean
```

How to read an R help page?

What functions usually do

Arguments often tell the function what to do in specific circumstances

```
missing_numbers ← c(1:10, NA, 20, 32, 50, 104, 32, 147, 99, NA, 45)  
mean(missing_numbers)  
[1] NA
```

```
mean(missing_numbers, na.rm = TRUE)  
[1] 32.44444
```

Or select from one of several options

```
## Look at ?mean to see what `trim` does  
mean(missing_numbers, na.rm = TRUE, trim = 0.1)  
[1] 27.25
```

What functions usually do

There are all kinds of functions. They return different things.

```
summary(my_numbers)
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|------|---------|--------|------|---------|-------|
| 1.00 | 1.75 | 3.00 | 6.25 | 6.25 | 25.00 |

What functions usually do

You can assign the output of a function to a name, which turns it into an object.
(Otherwise it'll send its output to the console.)

```
my_summary ← summary(my_numbers)
```

```
my_summary
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|------|---------|--------|------|---------|-------|
| 1.00 | 1.75 | 3.00 | 6.25 | 6.25 | 25.00 |

What functions usually do

Objects hang around in your work environment until they are overwritten by you, or are deleted.

```
## rm() function removes objects  
rm(my_summary)  
  
my_summary  
  
## Error: object 'my_summary' not found
```

Functions can be nested

```
c(1:20)
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
mean(c(1:20))
```

```
[1] 10.5
```

```
summary(mean(c(1:20)))
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|------|---------|--------|------|---------|------|
| 10.5 | 10.5 | 10.5 | 10.5 | 10.5 | 10.5 |

```
names(summary(mean(c(1:20))))
```

```
[1] "Min."      "1st Qu."   "Median"    "Mean"      "3rd Qu."   "Max."
```

```
length(names(summary(mean(c(1:20)))))
```

```
[1] 6
```

Nested functions are evaluated from the inside out.

Use the pipe operator: |>

Instead of deeply nesting functions in parentheses, we can use the *pipe operator*:

```
c(1:20) |> mean() |> summary() |> names() |> length()
```

```
[1] 6
```

Read this operator as “*and then*”

Like the assignment operator, this is two characters. To write it, type the pipe character | and then the > character. Better, use a keyboard shortcut. On a Mac, **Command Shift M**. On a Windows PC, **Ctrl Shift M**.

If the pipe looks like this, %>% (the **Magrittr pipe**) don’t worry, it will work the same as the one shown here, |>. But to change it, go to Global Options > Code and *check* the “Use native pipe operator” box.

Use the pipe operator: |>

Better, vertical space is free in R:

```
c(1:20) |>  
  mean() |>  
  summary() |>  
  names() |>  
  length()
```

```
[1] 6
```

Pipelines make code more readable

Not great, Bob:

```
serve(stir(pour_in_pan(whisk(crack_eggs(get_from_fridge(eggs), into = "bowl"), len = 40), temp = "med-high")))
```

Notice how the first thing you read is the last operation performed.

Pipelines make code more readable

We can use vertical space and indents, but it's really not much better:

```
serve(
  stir(
    pour_in_pan(
      whisk(
        crack_eggs(
          get_from_fridge(eggs),
          into = "bowl"),
        len = 40),
      temp = "med-high")
    )
)
```

Pipelines make code more readable

Much nicer:

```
eggs ▷  
  get_from_fridge() ▷  
  crack_eggs(into = "bowl") ▷  
  whisk(len = 40) ▷  
  pour_in_pan(temp = "med-high") ▷  
  stir() ▷  
  serve()
```

We'll still use nested parentheses quite a bit, often in the context of a function working inside a pipeline. But it's good not to have too many levels of nesting.

Functions are bundled into packages

Packages are loaded into your working environment using the `library()` function:

```
## A package containing a dataset rather than functions
library(gapminder)

gapminder

# A tibble: 1,704 × 6
  country   continent year lifeExp      pop gdpPercap
  <fct>     <fct>    <int>   <dbl>    <int>     <dbl>
1 Afghanistan Asia      1952    28.8  8425333     779.
2 Afghanistan Asia      1957    30.3  9240934     821.
3 Afghanistan Asia      1962    32.0  10267083    853.
4 Afghanistan Asia      1967    34.0  11537966    836.
5 Afghanistan Asia      1972    36.1  13079460    740.
6 Afghanistan Asia      1977    38.4  14880372    786.
7 Afghanistan Asia      1982    39.9  12881816    978.
8 Afghanistan Asia      1987    40.8  13867957    852.
9 Afghanistan Asia      1992    41.7  16317921    649.
10 Afghanistan Asia     1997    41.8  22227415    635.
# i 1,694 more rows
```

Functions are bundled into packages

You need only *install* a package once (and occasionally update it):

```
## Do at least once for each package. Once done, not needed each time.  
install.packages("palmerpenguins", repos = "http://cran.rstudio.com")  
  
## Needed sometimes, especially after an R major version upgrade.  
update.packages(repos = "http://cran.rstudio.com")
```

Functions are bundled into packages

But you must *load* the package in each R session before you can access its contents:

```
## To load a package, usually at the start of your Quarto document or script file
library(palmerpenguins)
penguins

# A tibble: 344 × 8
  species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>        <dbl>        <dbl>          <int>        <int>
1 Adelie  Torgersen     39.1         18.7          181        3750
2 Adelie  Torgersen     39.5         17.4          186        3800
3 Adelie  Torgersen     40.3         18            195        3250
4 Adelie  Torgersen      NA           NA            NA          NA
5 Adelie  Torgersen     36.7         19.3          193        3450
6 Adelie  Torgersen     39.3         20.6          190        3650
7 Adelie  Torgersen     38.9         17.8          181        3625
8 Adelie  Torgersen     39.2         19.6          195        4675
9 Adelie  Torgersen     34.1         18.1          193        3475
10 Adelie Torgersen      42           20.2          190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

Grabbing a single function with ::

```
# A little glimpse of what we'll do soon
penguins %>
  count(species, sex, year) %>
  pivot_wider(names_from = year, values_from = n) %>
  tinytable::tt()
```

| species | sex | 2007 | 2008 | 2009 |
|-----------|--------|------|------|------|
| Adelie | female | 22 | 25 | 26 |
| Adelie | male | 22 | 25 | 26 |
| Adelie | NA | 6 | NA | NA |
| Chinstrap | female | 13 | 9 | 12 |
| Chinstrap | male | 13 | 9 | 12 |
| Gentoo | female | 16 | 22 | 20 |
| Gentoo | male | 17 | 23 | 21 |
| Gentoo | NA | 1 | 1 | 3 |

Remember those conflicts?

```
library(tidyverse)
— Attaching core tidyverse packages ━━━━━━━━━━━━━━━━ tidyverse 2.0.0 ━━━━━
✓ dplyr     1.1.4    ✓ readr     2.1.5
✓ forcats   1.0.0    ✓ stringr   1.5.1
✓ ggplot2   3.5.1    ✓ tibble    3.2.1
✓ lubridate 1.9.3    ✓ tidyverse  1.3.1
✓ purrr    1.0.2
— Conflicts ━━━━━━━━━━━━━━━━ tidyverse_conflicts() ━━━
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package to force all conflicts to become errors
```

Some functions in different packages have the same names.

Related concepts of *namespaces* and *environments*.

The scope of names

```
x ← c(1:10)  
y ← c(90:100)
```

```
x
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
y
```

```
[1] 90 91 92 93 94 95 96 97 98 99 100
```

```
mean()
```

```
## Error in mean.default() : argument "x" is missing, with no default
```

The scope of names

```
mean(x) # argument names are internal to functions
```

```
[1] 5.5
```

```
mean(x = x)
```

```
[1] 5.5
```

```
mean(x = y)
```

```
[1] 95
```

```
x
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
y
```

```
[1] 90 91 92 93 94 95 96 97 98 99 100
```

5. Vector **types**; Object **classes**

I'm going to speak somewhat loosely here for now, and gloss over some distinctions between object classes and data structures, as well as kinds of objects and their attributes.

5. Vector types; Object classes

Objects are made of one or more vectors. A vector can, in effect, have a single type: integer, double, logical, character, factor, date, etc. That is, vectors are “atomic”. Complex objects are mostly lists of vectors of different sorts, or nested lists of other simpler objects that are themselves ultimately made up of vectors of

5. Vector **types**; Object **classes**

The object inspector in RStudio is your friend.

You can ask an object what it is at the console, too:

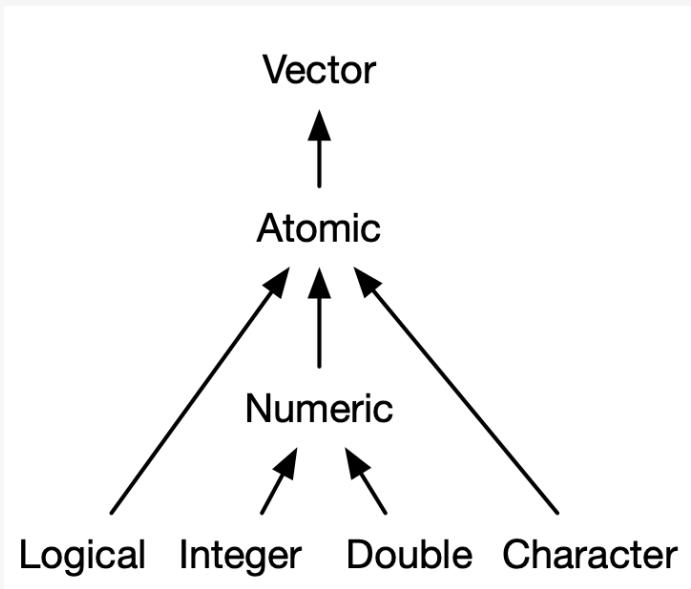
```
class(my_numbers)
```

```
[1] "numeric"
```

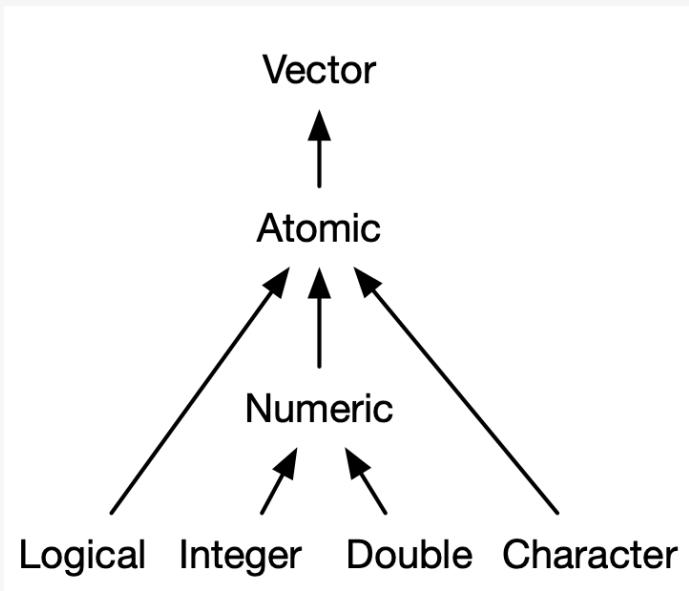
```
typeof(my_numbers)
```

```
[1] "double"
```

Types of vector



Types of vector



```
my_int <- c(1, 3, 5, 6, 10)  
is.integer(my_int)
```

```
[1] FALSE
```

```
is.double(my_int)
```

```
[1] TRUE
```

```
my_int <- as.integer(my_int)  
is.integer(my_int)
```

```
[1] TRUE
```

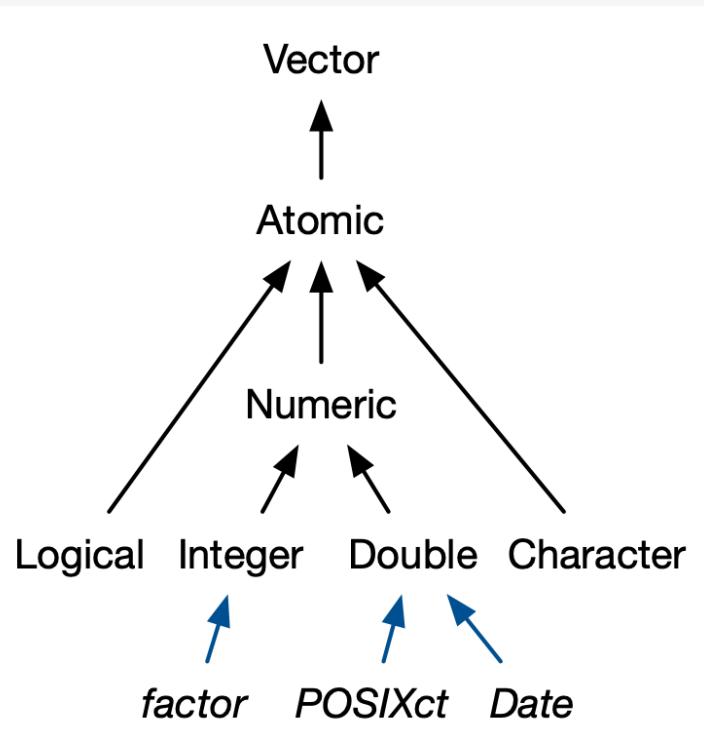
```
my_chr <- c("Mary", "had", "a", "little", "lamb")  
is.character(my_chr)
```

```
[1] TRUE
```

```
my_lgl <- c(TRUE, FALSE, TRUE)  
is.logical(my_lgl)
```

```
[1] TRUE
```

Types of vector



```
## Factors are for storing unordered or ordered categorical variables  
x <- factor(c("Yes", "No", "No", "Maybe", "Yes", "Yes", "Yes", "No"))  
x
```

```
[1] Yes No No Maybe Yes Yes Yes Yes No  
Levels: Maybe No Yes
```

```
summary(x) # Alphabetical order by default
```

```
Maybe No Yes  
1 3 4
```

```
typeof(x) # Underneath, a factor is a type of integer ...
```

```
[1] "integer"
```

```
attributes(x) # ... with labels for its numbers, or "levels"
```

```
$levels  
[1] "Maybe" "No" "Yes"  
  
$class  
[1] "factor"
```

```
levels(x)
```

```
[1] "Maybe" "No" "Yes"
```

```
is.ordered(x)
```

```
[1] FALSE
```

Vectors can't be heterogenous

Objects can be manually or automatically coerced from one class to another.
Take care.

```
class(my_numbers)
```

```
[1] "numeric"
```

```
my_new_vector ← c(my_numbers, "Apple")
```

```
my_new_vector # vectors are homogeneous/atomic
```

```
[1] "1"      "2"      "3"      "1"      "3"      "5"      "25"     "10"     "Apple"
```

```
class(my_new_vector)
```

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

Vectors can't be heterogenous

Objects can be manually or automatically coerced from one class to another.
Take care.

```
my_dbl <- c(2.1, 4.77, 30.111, 3.14519)  
is.double(my_dbl)
```

```
[1] TRUE
```

```
my_dbl <- as.integer(my_dbl)  
  
my_dbl
```

```
[1] 2 4 30 3
```

A table of data is a kind of list

```
gapminder # tibbles and data frames can contain vectors of different types
```

```
# A tibble: 1,704 × 6
  country      continent    year  lifeExp     pop gdpPercap
  <fct>        <fct>    <int>    <dbl>    <int>     <dbl>
1 Afghanistan Asia      1952    28.8  8425333    779.
2 Afghanistan Asia      1957    30.3  9240934    821.
3 Afghanistan Asia      1962    32.0 10267083    853.
4 Afghanistan Asia      1967    34.0 11537966    836.
5 Afghanistan Asia      1972    36.1 13079460    740.
6 Afghanistan Asia      1977    38.4 14880372    786.
7 Afghanistan Asia      1982    39.9 12881816    978.
8 Afghanistan Asia      1987    40.8 13867957    852.
9 Afghanistan Asia      1992    41.7 16317921    649.
10 Afghanistan Asia     1997    41.8 22227415    635.
# i 1,694 more rows
```

```
class(gapminder)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
typeof(gapminder) # hmm
```

```
[1] "list"
```

A table of data is a kind of list

Lists are collections of vectors of possibly different types and lengths, or collections of more complex objects that are themselves ultimately made out of vectors. Underneath, most complex R objects are some kind of list with different components that can be accessed by some function that knows the names of the things inside the list.

A *data frame* is a list of vectors of the same length, where the vectors can be of different types (e.g. numeric, character, logical, etc).

A data frame is a natural representation of what most real tables of data look like. Having it be a basic sort of entity in the programming language **IS ONE OF R's BEST IDEAS AND EASILY UNDERRATED!**

A *tibble* is an enhanced data frame

Some classes are versions of others

Base R's trusty `data.frame`

```
library(socviz)
titanic

  fate    sex    n percent
1 perished male 1364    62.0
2 perished female 126     5.7
3 survived male  367    16.7
4 survived female 344    15.6
```

```
class(titanic)

[1] "data.frame"
```

```
## The `\$` idiom picks out a named column here;
## more generally, the named element of a list
titanic$percent
```

```
[1] 62.0 5.7 16.7 15.6
```

Some classes are versions of others

Base R's trusty `data.frame`

```
library(socviz)
titanic

  fate   sex     n percent
1 perished male 1364    62.0
2 perished female 126    5.7
3 survived male  367   16.7
4 survived female 344   15.6
```

```
class(titanic)

[1] "data.frame"
```

```
## The `\$` idiom picks out a named column here;
## more generally, the named element of a list
titanic$percent
```

```
[1] 62.0 5.7 16.7 15.6
```

The Tidyverse's enhanced `tibble`

```
## tibbles are build on data frames
titanic_tb <- as_tibble(titanic)
titanic_tb

# A tibble: 4 × 4
  fate       sex     n percent
  <fct>     <fct>   <dbl>   <dbl>
1 perished male     1364    62
2 perished female   126     5.7
3 survived male     367    16.7
4 survived female   344    15.6
```

```
class(titanic_tb)

[1] "tbl_df"     "tbl"        "data.frame"
```

A data frame and a tibble are both fundamentally a list of vectors of the same length, where the vectors can be of different types (e.g. numeric, character, logical, etc)

All of this will be clearer in use

```
gss_sm
```

```
# A tibble: 2,867 × 32
  year   id ballot      age child� sibs degree race  sex   region income16
  <dbl> <dbl> <labelled> <dbl> <dbl> <labe> <fct> <fct> <fct> <fct> <fct>
1 2016    1 1           47     3 2  Bach... White Male  New E... $170000...
2 2016    2 2           61     0 3  High ... White Male  New E... $50000 ...
3 2016    3 3           72     2 3  Bach... White Male  New E... $75000 ...
4 2016    4 1           43     4 3  High ... White Fema... New E... $170000...
5 2016    5 3           55     2 2  Gradu... White Fema... New E... $170000...
6 2016    6 2           53     2 2  Junio... White Fema... New E... $60000 ...
7 2016    7 1           50     2 2  High ... White Male  New E... $170000...
8 2016    8 3           23     3 6  High ... Other Fema... Middl... $30000 ...
9 2016    9 1           45     3 5  High ... Black Male  Middl... $60000 ...
10 2016   10 3          71     4 1  Junio... White Male  Middl... $60000 ...
# i 2,857 more rows
# i 21 more variables: relig <fct>, marital <fct>, padeg <fct>, madeg <fct>,
# partyid <fct>, polviews <fct>, happy <fct>, partners <fct>, grass <fct>,
# zodiac <fct>, pres12 <labelled>, wtssall <dbl>, income_rc <fct>,
# agegrp <fct>, ageq <fct>, siblings <fct>, kids <fct>, religion <fct>,
# bigregion <fct>, partners_rc <fct>, obama <dbl>
```

Tidyverse tools are generally *type safe*, meaning their functions return the same type of thing every time, or fail if they cannot do this. So it's good to know about the various data types.

6. Arithmetic on vectors

In R, all numbers are vectors of different sorts. Even single numbers (“scalars”) are conceptually vectors of length 1.

Arithmetic on vectors (and arrays generally) follows a series of *recycling rules* that favor ease of expression of vectorized, “elementwise” operations.

See if you can predict what the following operations do:

6. Arithmetic on vectors

```
my_numbers
```

```
[1] 1 2 3 1 3 5 25 10
```

```
result1 ← my_numbers + 1
```

6. Arithmetic on vectors

```
my_numbers
```

```
[1] 1 2 3 1 3 5 25 10
```

```
result1 ← my_numbers + 1
```

```
result1
```

```
[1] 2 3 4 2 4 6 26 11
```

6. Arithmetic on vectors

```
result2 ← my_numbers + my_numbers
```

6. Arithmetic on vectors

```
result2 ← my_numbers + my_numbers
```

```
result2
```

```
[1]  2  4  6  2  6 10 50 20
```

6. Arithmetic on vectors

```
two_nums ← c(5, 10)  
result3 ← my_numbers + two_nums
```

6. Arithmetic on vectors

```
two_nums ← c(5, 10)  
  
result3 ← my_numbers + two_nums  
  
result3  
  
[1] 6 12 8 11 8 15 30 20
```

6. Arithmetic on vectors

```
three_nums ← c(1, 5, 10)  
  
result4 ← my_numbers + three_nums
```

Warning in my_numbers + three_nums: longer object length is not a multiple of
shorter object length

6. Arithmetic on vectors

```
three_nums ← c(1, 5, 10)  
  
result4 ← my_numbers + three_nums
```

```
Warning in my_numbers + three_nums: longer object length is not a multiple of  
shorter object length
```

```
result4  
  
[1]  2  7 13  2  8 15 26 15
```

Note that you get a *warning* here. It'll still do it, though! Don't ignore warnings until you understand what they mean.

7. R will be **frustrating**

The IDE tries its best to help you. Learn to attend to what it is trying to say.

```
Warning message:  
In my_numbers + two_nums :  
  longer object length is not a multiple of shorter object length
```

```
  38      #> So make a plot :  
  39  p <- ggplot(data = gapminder  
  40          ,  
  41          mapping = aes(x = gdpPercap,  
  42          y = lifeExp))
```

```
 39  p <- ggplot(data = gapminder,  
 40                  mapping = aes(x = gdpPercap,  
 41                  y = lifeExp)))  
 42  #> unexpected token ')'  
 43  #> |
```

Let's Go!

Appendix: A Few More R Details

Logic: Watch out!

Here's a gotcha. You might think you could write `3 < 5 & 7` and have it be interpreted as "Three is less than five and also less than seven [True or False?]:

```
3 < 5 & 7
```

```
[1] TRUE
```

It seems to work!

Logic: Watch out!

But now try `3 < 5 & 1`, where your intention is “Three is less than five and also less than one [True or False?]”

```
3 < 5 & 1
```

```
[1] TRUE
```

What's happening is that `3 < 5` is evaluated first, and resolves to `TRUE`, leaving us with the expression `TRUE & 1`.

R interprets this as `TRUE & as.logical(1)`.

In Boolean algebra, `1` resolves to `TRUE`. Any other number is `FALSE`. So,

Logic: Watch out!

```
TRUE & as.logical(1)
```

```
[1] TRUE
```

```
3 < 5 & 3 < 1
```

```
[1] FALSE
```

You have to make your comparisons explicit.

Logic and floating point arithmetic

Let's evaluate $0.6 + 0.2 = 0.8$

Logic and floating point arithmetic

Let's evaluate $0.6 + 0.2 = 0.8$

```
0.6 + 0.2 = 0.8
```

```
[1] TRUE
```

Logic and floating point arithmetic

Let's evaluate $0.6 + 0.2 = 0.8$

```
0.6 + 0.2 = 0.8
```

```
[1] TRUE
```

Now let's try $0.6 + 0.3 = 0.9$

Logic and floating point arithmetic

Let's evaluate $0.6 + 0.2 = 0.8$

```
0.6 + 0.2 = 0.8
```

```
[1] TRUE
```

Now let's try $0.6 + 0.3 = 0.9$

```
0.6 + 0.3 = 0.9
```

```
[1] FALSE
```

Er. That's not right.

Welcome to floating point math!

In Base 10, you can't precisely express fractions like $\frac{1}{3}$ and $\frac{1}{9}$. They come out as repeating decimals: 0.3333... or 0.1111... You *can* cleanly represent fractions that use a prime factor of the base, which in the case of Base 10 are 2 and 5.

Welcome to floating point math!

In Base 10, you can't precisely express fractions like $\frac{1}{3}$ and $\frac{1}{9}$. They come out as repeating decimals: 0.3333... or 0.1111... You *can* cleanly represent fractions that use a prime factor of the base, which in the case of Base 10 are 2 and 5.

Computers represent numbers as binary (i.e. Base 2) floating-points. In Base 2, the only prime factor is 2. So $\frac{1}{5}$ or $\frac{1}{10}$ in binary would be repeating.

Logic and floating point arithmetic

When you do binary math on repeating numbers and convert back to decimals you get tiny leftovers, and this can mess up *logical* comparisons of equality. The `all.equal()` function exists for this purpose.

```
print(.1 + .2)
```

```
[1] 0.3
```

```
print(.1 + .2, digits=18)
```

```
[1] 0.30000000000000044
```

```
all.equal(.1 + .2, 0.3)
```

```
[1] TRUE
```

See e.g. <https://0.3000000000000004.com>

More later on why
this might bite you,
and how to deal
with it

For now, “Be very careful about doing logical comparisons on floating-point numbers” is not a bad rule.

Assignment with =

You can use = as well as ← for assignment.

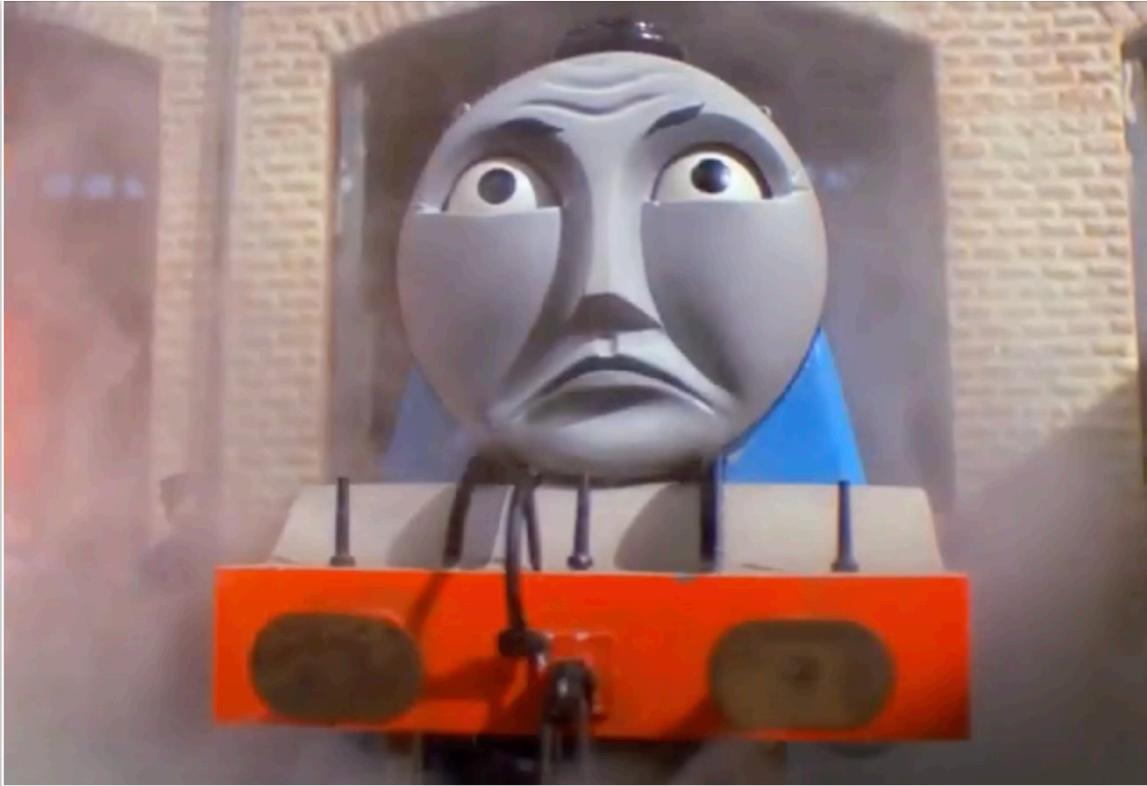
```
my_numbers = c(1, 2, 3, 1, 3, 5, 25)  
  
my_numbers  
  
[1] 1 2 3 1 3 5 25
```

On the other hand, = has a different meaning when used in functions.

I'm going to use ← for assignment throughout.

Be consistent either way.

Assignment with =



**It isn't *wrong*,
but we just
*don't do it.***

The other pipe: %>%

The Base R pipe operator, `>` is a relatively recent addition to R.

Piping operations were originally introduced in a package called `magrittr`, where it took the form `%>%`

The other pipe: %>%

The Base R pipe operator, `>` is a relatively recent addition to R.

Piping operations were originally introduced in a package called `magrittr`, where it took the form `%>%`

It's been so successful, a version of it has been incorporated into Base R. It *mostly* but does not *quite* work the same way as `%>%` in every case.

The other pipe: %>%

The Base R pipe operator, `>` is a relatively recent addition to R.

Piping operations were originally introduced in a package called `magrittr`, where it took the form `%>%`

It's been so successful, a version of it has been incorporated into Base R. It *mostly* but does not *quite* work the same way as `%>%` in every case. We'll use the Base R pipe in this course, but you'll see the Magrittr pipe a lot out in the world.

With the Base R pipe, you can only pass an object to the *first* argument in a function. This is fine for most tidyverse pipelines, where the first argument is usually (implicitly) the data. But it does mean that most Base R functions will continue not to be easily piped, as most of them do not follow the convention of passing the current data as the first argument

Object classes

Objects can have more than one (nested) class:

```
summary(my_numbers)
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-------|---------|--------|-------|---------|--------|
| 1.000 | 1.500 | 3.000 | 5.714 | 4.000 | 25.000 |

```
my_smry ← summary(my_numbers) # remember, outputs can be assigned to a name, creating an object
```

```
class(summary(my_numbers)) # functions can be nested, and are evaluated from the inside out
```

```
[1] "summaryDefault" "table"
```

```
class(my_smry) # equivalent to the previous line
```

```
[1] "summaryDefault" "table"
```

Object classes

```
typeof(my_smry)
```

```
[1] "double"
```

```
attributes(my_smry)
```

```
$names  
[1] "Min."     "1st Qu."  "Median"   "Mean"      "3rd Qu."  "Max."
```

```
$class  
[1] "summaryDefault" "table"
```

```
## In this case, the functions extract the corresponding attribute  
class(my_smry)
```

```
[1] "summaryDefault" "table"
```

```
names(my_smry)
```

```
[1] "Min."     "1st Qu."  "Median"   "Mean"      "3rd Qu."  "Max."
```