

Welcome!

An overview of the course & intro
to data types

Daniel Anderson

Week 1

Agenda

- Getting on the same page
- Syllabus
- Intro to data types





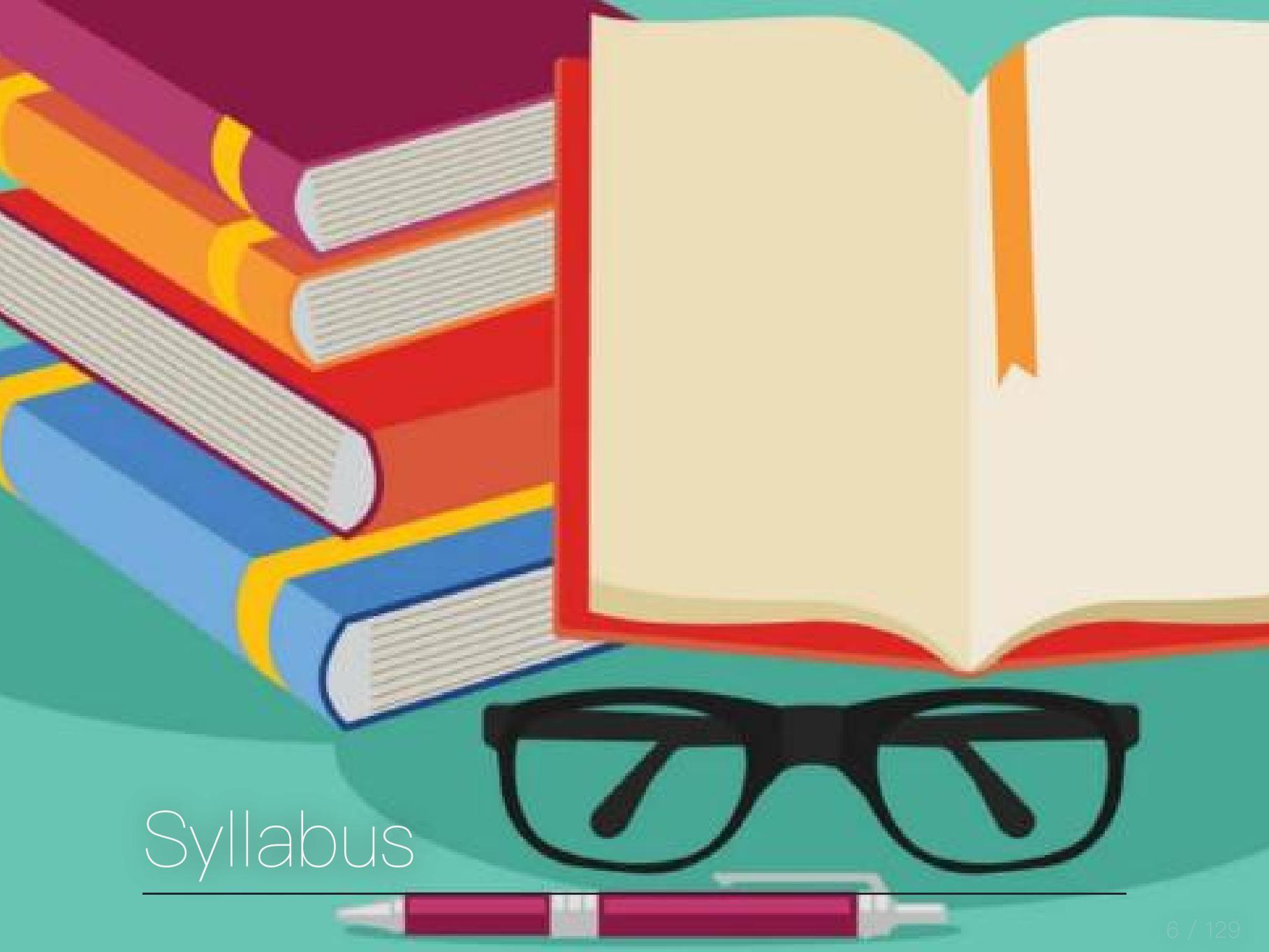
Getting on the same page

Introduce yourself!

- We mostly know each other, but it's always good to hear from each other.
- Tell us why you're taking the class
- What's one fun thing you've done recently outside of school stuff?
- What pronouns would you like us to use for you in this class?

This term for me

- Full-time employed with Abl starting next week.
- This is my only UO responsibilities
- Doesn't mean I'm less committed to you
- Does mean time is going to be tricky
 - Friday afternoons will be best for meetings – can do evenings too if needed.



Syllabus

Course Website(s)

The screenshot shows a course website with a dark background and light text. At the top, there is a navigation bar with five items: Home, Schedule, Assignments, Syllabus, Tags, and Class repo. The 'Home' item is highlighted with a teal background. Below the navigation bar, the title 'Functional Programming' is displayed in large, bold, white font. Underneath the title, the subtitle 'for educational data science' is written in a smaller, gray, italicized font. A main paragraph in white text provides an overview of the course: 'Welcome to the second course on multilevel modeling taught through the University of Oregon's College of Education. This course will be taught through R, a free and open-source statistical computing environment. We will dive deeper into the variance-covariance matrices of multilevel models, how predictions are made from multilevel models, and how to simulate data assuming a multilevel data structure. Both frequentist and Bayesian approaches to estimation will be covered. Please see the [schedule](#) for a complete listing of topics.'

Course learning objectives

- Understand and be able to describe the differences in R's data structures and when each is most appropriate for a given task
- Explore `purrr::map` and its variants, how they relate to base R functions, and why the `{purrr}` variants are often preferable.
- Work with lists and list columns using `purrr::nest` and `purrr:unnest`
- Understand how `dplyr::rowwise()` can help you avoid some of the above

Course learning objectives

- Convert repetitive tasks into functions
- Understand elements of good functions, and things to avoid
- Write effective and clear functions with the mantra of "Don't Repeat Yourself"

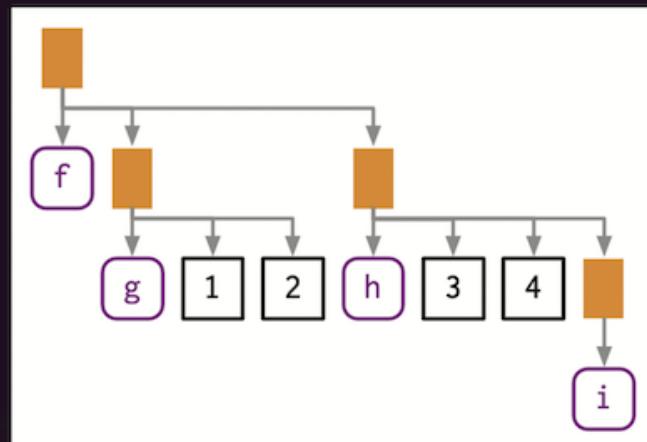
This Week's learning objectives

- Understand the requirements of the course
- Understand the requirements of the final project
- Understand the fundamental difference between lists and atomic vectors
- Understand how atomic vectors are coerced, implicitly or explicitly
- Understand various ways to subset vectors, and how subsetting differs for lists
- Understand attributes and how to set/modify

Textbooks

Advanced R

Second Edition



Link

Hadley Wickham



CRC Press
Taylor & Francis Group

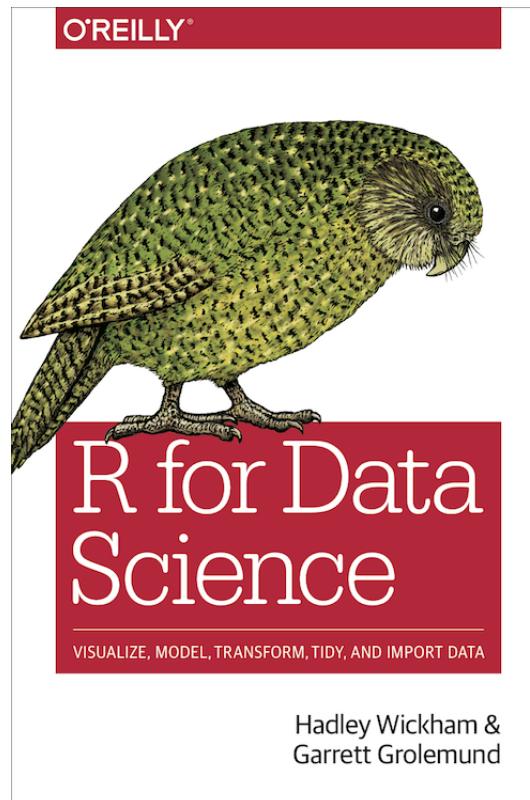
A CHAPMAN & HALL BOOK

Other books (also free)

Bryan



Wickham & Grolemund



Structure of the course

- First 5 weeks – mostly iteration
 - Data types
 - Base R iterations
 - {purrr}
 - Batch processes and working with list columns
 - Parallel iterations (and a few extras)
- Second 5 weeks – Writing functions and shiny
 - Writing functions 1–3
 - Shiny 1–3

Labs

15%

3 @ 10 points each

Two labs on iteration and one on functions.

If possible – please try to be in-class on Lab days. I understand this is not always possible, but it helps me help you.

| Lab | Date Assigned | Date Due | Topic |
|-----|---------------|---------------|---------------------------------------------------------|
| 1 | Mon, April 04 | Mon, April 11 | Subsetting lists and base R <code>for()</code> loops |
| 2 | Mon, April 11 | Mon, April 18 | Multiple models and API calls with <code>{purrr}</code> |
| 3 | Mon, May 09 | Mon, May 16 | Create and apply functions |

Midterm

70 points total (35%)

Two parts:

- Small quiz on canvas to demonstrate knowledge (4/22; 10 points)
 - Identifying bugs in code
 - Multiple choice/fill-in the blank questions
 - Free response
- Take-home portion to demonstrate ability to write the correct code (assigned 4/22; 60 points)
 - Write loops to solve problems

Take home midterm

- Group project: 3–5 people
- Shared GitHub repo
- Divide it up to ease the workload, then just check each other's work

Already posted!

Final Project

100 points total (60%)

5 parts

| Component | Due | Points | Percentage of final grade |
|-------------------|----------|--------|---------------------------|
| Groups finalized | 04-04-22 | 0 | 0 |
| Outline | 04-18-22 | 5 | 2.5 |
| Draft data script | 05-16-22 | 10 | 5 |
| Peer Review | 05-23-22 | 15 | 7.5 |
| Final Product | 06-06-22 | 70 | 35 |

What is it?

Two basic options

Data Product

Similar to first class

- Brief research manuscript (can be APA or not, I don't really care 
- Shiny app
- Dashboard
 - Probably unlikely to work well though, unless you make it a shiny dashboard
- Blog post
- For the ambitious – a documented R package

Tutorial

- Probably best done through a blog post or series of blog posts
- Approach as if you're teaching others about the content I'll ask you to cover
- BONUS: You can actually release the blog post(s) and may get some traffic 



Make it your own

What you have to have

- Everything on GitHub
- Publicly available dataset
- Team of 2–5

What you have to cover

Unfortunately, this still is a class assignment. I have to be able to evaluate that you can actually apply the content within a messy, real-world setting.

The grading criteria (which follow) may force you into some use cases that are a bit artificial. This is okay.

Grading criteria

- No code is used repetitively (no more than twice) 10 points
- More than one variant of `purrr::map` is used 5 points
- At least one {purrr} function outside the basic `map` family (`walk_*`, `reduce`, `modify_*`, etc.) 5 points
- At least one instance of parallel iteration (e.g., `map2_*`, `pmap_*`) 5 points
- At least one use case of `purrr::nest %>% mutate()` 5 points

Grading criteria

- At least two custom functions 20 points; 10 points each
 - Each function must do exactly one thing
 - The functions **may** replicate the behavior of a base function – as noted above this is about practicing the skills you learn in class
- Code is fully reproducible and housed on GitHub 10 points
- No obvious errors in chosen output format 5 points
- Deployed on the web and shareable through a link 5 points

Outline

Due 4/18/21

Four components:

- Description of data source (must be publicly available)
- Purpose (tutorial or substantive)
- Chosen format
- Lingering questions
 - How can I help?

Please include all components – including the question(s) section!

Draft

Due 5/16/21, before class

- Expected to still be a work in progress
 - This means some of your code may be rough and/or incomplete. However:
- Direction should be obvious
- Most, if not all, grading elements should be present
- Provided to your peers so they can learn from you as much as you can learn from their feedback

Peer Review

- Exact same process we've used before
- If, during your peer review, you find grading elements not present, definitely note them

Utilizing GitHub (required)

- You'll be assigned two groups to review
- Fork their repo
- Embed comments, suggest changes to their code
 - Please do both of these
- Submit a PR
 - Summarize your overall review in the PR

Grading

200 points total

- 3 labs at 10 points each (30 points; 15%)
- Midterm in-class (10 points; 5%)
- Midterm take-home (60 points; 30%)
- Final Project (100 points; 50%)
 - Outline (5 points; 2.5%)
 - Draft (10 points; 5%)
 - Peer review (15 points; 7.5%)
 - Product (70 points; 35%)

Grading

| Lower percent | Lower point range | Grade | Upper point range | Upper percent |
|---------------|-------------------|-------|-------------------|---------------|
| 0.970+ | (194 pts or more) | A+ | | |
| 0.930 | (186 pts) | A | (193 pts) | 0.969 |
| 0.900 | (180 pts) | A- | (185 pts) | 0.929 |
| 0.870 | (174 pts) | B+ | (179 pts) | 0.899 |
| 0.830 | (166 pts) | B | (173 pts) | 0.869 |
| 0.800 | (160 pts) | B- | (165 pts) | 0.829 |
| 0.770 | (154 pts) | C+ | (159 pts) | 0.799 |
| 0.730 | (146 pts) | C | (153 pts) | 0.769 |
| 0.700 | (140 pts) | C- | (145 pts) | 0.739 |
| | | F | (139 pts or less) | 0.699 |

A note on feedback

- Last term, I gave you feedback on everything you submitted
- This term, I will give you feedback on the midterm and the final only
- Labs scored on a completion basis
- We will go over everything in class

Week 6

Fully remote

I will be traveling and plan to teach from my hotel room.



COMMUNICATION

Break

05 : 00

Basic data types

Vectors

05:00

Pop quiz

Discuss in small breakout groups

- What are the four basic types of atomic vectors?
- What function creates a vector?
- **T/F:** A list (an R list) is not a vector.
- What is the fundamental difference between a matrix and a data frame?
- What does *coercion* mean, and when does it come into play?

Vector types

4 basic types

Note there are two others (complex and raw), but we almost never care about them.

- Integer
- Double
- Logical
- Character

Integer and double vectors are both numeric.

Creating vectors

Vectors are created with **c**. Below are examples of each of the four main types of vectors.

```
# L explicitly an integer, not double
integer <- c(5L, 7L, 3L, 94L)
double <- c(3.27, 8.41, Inf, -Inf)
logical <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)
character <- c("red", "orange", "yellow", "green", "blue",
             "violet", "rainbow")
```

Coercion

- Vectors **must** be of the same type.
- If you try to mix types, implicit coercion will occur
- Implicit coercion defaults to the most flexible type
 - which is... ?

```
c(7L, 3.25)
```

```
## [1] 7.00 3.25
```

```
c(TRUE, 5)
```

```
## [1] 1 5
```

```
c(3.24, TRUE, "April")
```

```
## [1] "3.24"  "TRUE"  "April"
```

Explicit coercion

- You can alternatively define the coercion to occur

```
as.integer(c(7L, 3.25))
```

```
## [1] 7 3
```

```
as.logical(c(3.24, TRUE, "April"))
```

```
## [1] NA TRUE NA
```

```
as.character(c(TRUE, 5)) # still maybe a bit unexpected?
```

```
## [1] "1" "5"
```

Checking types

- Use `typeof` to verify the type of vector

```
typeof(c(7L, 3.25))
```

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

```
typeof(as.integer(c(7L, 3.25)))
```

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

Coercing to logical

```
as.logical(c(0, 1, 1, 0))
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

Any number that is not zero gets coerced to **FALSE**

```
as.logical(c(0, 5L, 7.4, -1.6, 0))
```

```
## [1] FALSE  TRUE  TRUE  TRUE FALSE
```

Wait... why the **NA** here?

```
as.logical(c(3.24, TRUE, "April"))
```

```
## [1] NA  TRUE  NA
```

Piping

- Although traditionally used within the tidyverse (not what we're doing here), it can still be useful. The following are equivalent

```
library(magrittr)  
  
typeof(as.integer(c(7L, 3.25)))  
  
## [1] "integer"
```

```
c(7L, 3.25) %>%  
  as.integer() %>%  
  typeof()
```

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

Don't actually need magrittr

Base pipe! |>

This is a bigger topic than I want to get into here, but for reference...

```
c(7L, 3.25) |>  
  as.integer() |>  
  typeof()
```

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

Pop quiz

Without actually running the code, predict which type each of the following will coerce to.

```
c(1.25, TRUE, 4L)  
c(1L, FALSE)  
c(7L, 6.23, "eight")  
c(TRUE, 1L, 0L, "False")
```

01 : 00

Answers

```
typeof(c(1.25, TRUE, 4L))
```

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

```
typeof(c(1L, FALSE))
```

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

```
typeof(c(7L, 6.23, "eight"))
```

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

```
typeof(c(TRUE, 1L, 0L, "False"))
```

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

Lists

- Lists are vectors, but not *atomic* vectors
- Fundamental difference – each element can be a different type

```
list("a", 7L, 3.25, TRUE)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] 3.25
##
## [[4]]
## [1] TRUE
```

Lists

- Each element of the list is another vector, possibly atomic, possibly not
- The prior example included all *scalar* vectors
- Lists do not require all elements to be the same length

```
list(  
  c("a", "b", "c"),  
  rnorm(5),  
  c(7L, 2L),  
  c(TRUE, TRUE, FALSE, TRUE)  
)
```

```
## [[1]]  
## [1] "a" "b" "c"  
##  
## [[2]]  
## [1] -0.2693514 -0.3909654 1.3487070  
## [4] -0.0227647  0.2442259  
##  
## [[3]]  
## [1] 7 2  
##  
## [[4]]  
## [1] TRUE TRUE FALSE TRUE
```

Summary

- Atomic vectors must all be the same type
 - implicit coercion occurs if not (and you haven't specified the coercion explicitly)
- Lists are also vectors, but not atomic vectors
 - Each element can be of a different type and length
 - Incredibly flexible, but often a little more difficult to get the hang of

Challenge

Work with a partner

One of you share your screen:

- Create four atomic vectors, one for each of the fundamental types
- Combine two or more of the vectors. Predict the implicit coercion of each.
- Apply explicit coercions, and predict the output for each.

(basically quiz each other)

05 : 00

Attributes

Attributes

- What are attributes?
 - metadata... what's metadata?
 - Data about the data

Other data types

Atomic vectors by themselves make up only a small fraction of the total number of data types in R

What are some other data types?

- Data frames
- Matrices & arrays
- Factors
- Dates

Remember, atomic vectors are the atoms of R. Many other data structures are built from atomic vectors.

- We use attributes to create other data types from atomic vectors

Attributes

Common

- Names
- Dimensions

Less common

- Arbitrary metadata

Examples

Please follow along!

- See **all** attributes associated with a give object with **attributes**

```
library(palmerpenguins)
attributes(penguins[1:50, ]) # limiting rows just for slides
```

```
## $names
## [1] "species"                 "island"
## [3] "bill_length_mm"          "bill_depth_mm"
## [5] "flipper_length_mm"       "body_mass_g"
## [7] "sex"                      "year"
##
## $row.names
##  [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14
## [15] 15 16 17 18 19 20 21 22 23 24 25 26 27 28
## [29] 29 30 31 32 33 34 35 36 37 38 39 40 41 42
## [43] 43 44 45 46 47 48 49 50
##
## $class
## [1] "tbl_df"        "tbl"           "data.frame"
```

```
head(penguins)
```

```
## # A tibble: 6 × 8
##   species island    bill_length_mm
##   <fct>   <fct>              <dbl>
## 1 Big     one      Torgersen        39.1
## 2 Big     one      Torgersen        39.5
## 3 Big     one      Torgersen        40.3
## 4 Big     one      Torgersen       NA
## 5 Big     one      Torgersen        36.7
## 6 Big     one      Torgersen        39.3
## # ... with 5 more variables:
## #   bill_depth_mm <dbl>,
## #   flipper_length_mm <int>,
## #   body_mass_g <int>, sex <fct>, year <int>
```

Get specific attribute

- Access just a single attribute by naming it within `attr`

```
attr(penguins, "class")
```

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

```
attr(penguins, "names")
```

```
## [1] "species"          "island"  
## [3] "bill_length_mm"   "bill_depth_mm"  
## [5] "flipper_length_mm" "body_mass_g"  
## [7] "sex"               "year"
```

Note – this is not generally how you would pull these attributes. Rather, you would use `class()` and `names()`.

Be specific

- Note in the prior slides, I'm asking for attributes on the entire data frame.
- Is that what I want?... maybe. But the individual vectors may have attributes as well

```
attributes(penguins$species)
```

```
## $levels  
## [1] "Big one"      "Little one"    "Funny one"  
##  
## $class  
## [1] "factor"
```

```
attributes(penguins$bill_length_mm)
```

```
## NULL
```

Set attributes

- Just redefine them within `attr`

```
attr(penguins$species, "levels") <- c("Big one",
                                         "Little one",
                                         "Funny one")
```

```
head(penguins)
```

```
## # A tibble: 6 × 8
##   species island    bill_length_mm
##   <fct>   <fct>            <dbl>
## 1 Big one Torgersen      39.1
## 2 Big one Torgersen      39.5
## 3 Big one Torgersen      40.3
## 4 Big one Torgersen       NA
## 5 Big one Torgersen      36.7
## 6 Big one Torgersen      39.3
## # ... with 5 more variables:
## #   bill_depth_mm <dbl>,
## #   flipper_length_mm <int>,
## #   body_mass_g <int>, sex <fct>, year <int>
```

Dimensions

- Let's create a matrix (please do it with me)

```
m <- matrix(1:6, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]     1     4
## [2,]     2     5
## [3,]     3     6
```

- Notice how the matrix fills
- Check out the attributes

```
attributes(m)
```

```
## $dim
## [1] 3 2
```

Modify the attributes

- Let's change it to a 2×3 matrix, instead of 3×2 (you try first)

```
attr(m, "dim") <- c(2, 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]     1     3     5
## [2,]     2     4     6
```

- is this the result you expected?

Alternative creation

- Create an atomic vector, assign a dimension attribute

```
v <- 1:6  
v
```

```
## [1] 1 2 3 4 5 6
```

```
attr(v, "dim") <- c(3, 2)  
v
```

```
##      [,1] [,2]  
## [1,]     1     4  
## [2,]     2     5  
## [3,]     3     6
```

Aside

- What if we wanted it to fill by row?

```
matrix(6:13,  
       ncol = 2,  
       byrow = TRUE)
```

```
##      [,1] [,2]  
## [1,]     6    7  
## [2,]     8    9  
## [3,]    10   11  
## [4,]    12   13
```

```
vect <- 6:13  
dim(vect) <- c(2, 4)  
vect
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]     6    8   10   12  
## [2,]     7    9   11   13
```

t(vect)

```
##      [,1] [,2]  
## [1,]     6    7  
## [2,]     8    9  
## [3,]    10   11  
## [4,]    12   13
```

Names

- The following (this slide and the next) are equivalent

```
dim_names <- list(  
  c("the first", "second", "III"),  
  c("index", "value"))  
  
attr(v, "dimnames") <- dim_names  
  
v
```

```
##           index value  
## the first     1     4  
## second       2     5  
## III          3     6
```

Names

```
v2 <- 1:6
attr(v2, "dim") <- c(3, 2)
rownames(v2) <- c("the first", "second", "III")
colnames(v2) <- c("index", "value")
v2
```

```
##           index value
## the first      1     4
## second        2     5
## III           3     6
```

Arbitrary metadata

```
attr(v, "matrix_mean") <- mean(v)  
v
```

```
##           index value  
## the first      1     4  
## second        2     5  
## III           3     6  
## attr(,"matrix_mean")  
## [1] 3.5
```

```
attr(v, "matrix_mean")
```

```
## [1] 3.5
```

- Note that *anything* can be stored as an attribute (including matrices or data frames, etc.)

Why would we do this?

This is a short example that is based on a real example

- Imagine we're accessing a database that has many years of data
- The tables in the database are the same, but the values (of course) differ
- We might want to return the data, but store the year as an attribute

This example is overly complicated for the first day, but it will give you an idea of what we're building toward.

Made up db

I'm using a list to mimic a database

```
db <- list(  
  data.frame(  
    color = c("red", "orange", "green"),  
    transparency = c(0.80, 0.65, 0.93)  
  ),  
  data.frame(  
    color = c("blue", "pink", "cyan"),  
    transparency = c(0.40, 0.35, 0.87)  
  )  
)
```

db

```
## [1]
##   color transparency
## 1   red      0.80
## 2 orange    0.65
## 3 green     0.93
##
## [2]
##   color transparency
## 1 blue      0.40
## 2 pink     0.35
## 3 cyan     0.87
```

Write a function

Let's write a function that grabs one of these tables. If it's "1920" we'll grab the first one, otherwise we'll grab the second one.

```
pull_color_data <- function(year) {  
  to_pull <- if(year == "1920") {  
    out <- db[[1]]  
  } else {  
    out <- db[[2]]  
  }  
  out  
}
```

Does it work?

```
pull_color_data(1920)
```

Yes!

```
##   color transparency
## 1   red      0.80
## 2 orange    0.65
## 3 green     0.93
```

```
pull_color_data(2021)
```

```
##   color transparency
## 1   blue     0.40
## 2   pink     0.35
## 3   cyan     0.87
```

```
pull_color_data(2122)
```

```
##   color transparency
## 1   blue     0.40
## 2   pink     0.35
## 3   cyan     0.87
```

Build a second function

Let's say we want to make a second function that does something with the previous output.

BUT

What we do with it is going to depend on the data frame we get back.

We need to know the year.

Modify our original function to store the year as an attribute!

Notice we redefine the attributes so we're including all the prior attributes it already had.

```
pull_color_data <- function(year) {  
  to_pull <- if(year == "1920") {  
    out <- db[[1]]  
  } else {  
    out <- db[[2]]  
  }  
  attributes(out) <- c(  
    attributes(out),  
    db = year  
  )  
  out  
}
```

Try

```
pull_color_data(1920)
```

```
##      color transparency
## 1     red          0.80
## 2 orange          0.65
## 3 green          0.93
```

```
attr(pull_color_data(2021), "db")
```

```
## [1] 2021
```

Build our second function

Now, we can make our second function, and have it do something different depending on the data that is passed to it.

```
print_colors <- function(color_data) {  
  title <- paste0("Colors for ", attr(color_data, "db"))  
  colorspace::swatchplot(color_data$color)  
  mtext(title) # base plotting function  
}
```

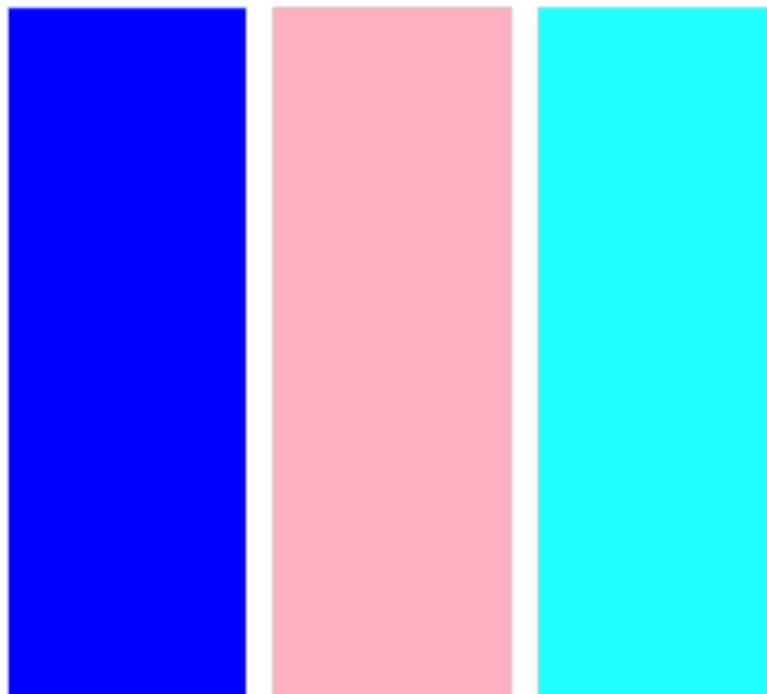
```
pull_color_data(1920) %>%  
  print_colors()
```

Colors for 1920



```
pull_color_data(2021) %>%  
  print_colors()
```

Colors for 2021



Another example

Fit a multilevel model and pull the variance–covariance matrix

```
m <- lme4::lmer(Reaction ~ 1 + Days + (1 + Days|Subject),  
                 data = lme4::sleepstudy)
```

```
lme4::VarCorr(m)$Subject
```

```
##                  (Intercept)      Days  
## (Intercept)  612.100158  9.604409  
## Days          9.604409 35.071714  
## attr(,"stddev")  
## (Intercept)      Days  
##   24.740658    5.922138  
## attr(,"correlation")  
##                  (Intercept)      Days  
## (Intercept)  1.00000000  0.06555124  
## Days          0.06555124 1.00000000
```

Matrices vs Data frames

Usually we want to work with data frames because they represent our data better.

Sometimes a matrix is more efficient because you can operate on the **entire** matrix at once.

```
set.seed(42)
m <- matrix(rnorm(100, 200, 10), ncol = 10)
m
```

```
##          [,1]      [,2]      [,3]      [,4]
## [1,] 213.7096 213.0487 196.9336 204.5545
## [2,] 194.3530 222.8665 182.1869 207.0484
## [3,] 203.6313 186.1114 198.2808 210.3510
## [4,] 206.3286 197.2121 212.1467 193.9107
## [5,] 204.0427 198.6668 218.9519 205.0496
## [6,] 198.9388 206.3595 195.6953 182.8299
## [7,] 215.1152 197.1575 197.4273 192.1554
## [8,] 199.0534 173.4354 182.3684 191.4909
## [9,] 220.1842 175.5953 204.6010 175.8579
## [10,] 199.3729 213.2011 193.6001 200.3612
##          [,5]      [,6]      [,7]      [,8]
```

```
sum(m)
```

```
## [1] 20032.51
```

```
mean(m)
```

```
## [1] 200.3251
```

```
rowSums(m)
```

```
## [1] 2048.470 1993.774 2041.155 2025.924  
## [5] 1978.173 2007.265 1998.086 1960.291  
## [9] 1952.476 2026.901
```

```
colSums(m)
```

```
## [1] 2054.730 1983.654 1982.192 1963.610  
## [5] 1997.978 2001.839 2053.908 1978.212  
## [9] 2025.111 1991.281
```

```
# standardize the matrix  
z <- (m - mean(m)) / sd(m)
```

z

```
##          [,1]      [,2]      [,3]
## [1,]  1.28528802  1.2218239 -0.3256841
## [2,] -0.57349498  2.1646089 -1.7417882
## [3,]  0.31748345 -1.3649263 -0.1963133
## [4,]  0.57650528 -0.2989403  1.1352110
## [5,]  0.35698951 -0.1592501  1.7887033
## [6,] -0.13313334  0.5794704 -0.4445968
## [7,]  1.42026916 -0.3041875 -0.2782756
## [8,] -0.12212321 -2.5821792 -1.7243635
## [9,]  1.90703954 -2.3747685  0.4106013
## [10,] -0.09144695  1.2364622 -0.6458013
##          [,4]      [,5]      [,6]
## [1,]  0.40613865  0.1665940  0.27791666
## [2,]  0.64562157 -0.3779416 -0.78393268
## [3,]  0.96277141  0.6968297  1.48192480
## [4,] -0.61596668 -0.7290676  0.58614338
## [5,]  0.45367758 -1.3451640  0.05497234
## [6,] -1.68004206  0.3844054  0.23434417
## [7,] -0.78452812 -0.8103926  0.62108770
## [8,] -0.84833774  1.3555260  0.05504171
## [9,] -2.34955213 -0.4455350 -2.90544454
## [10,]  0.00346451  0.5983857  0.24234547
##          [,7]      [,8]      [,9]
## [1,] -0.3838736 -1.0329155  1.42140711
## [2,]  0.1466507 -0.1178282  0.21645471
## [3,]  0.5274934  0.5675319  0.05370436
## [4,]  1.3129235 -0.9468782 -0.14731870
## [5,] -0.7296315 -0.5524942 -1.17812024
```

Stripping attributes

- Many operations will strip attributes (which makes storing important things in them a bit precarious)

```
v
```

```
##           index value
## the first      1     4
## second        2     5
## III           3     6
## attr(,"matrix_mean")
## [1] 3.5
```

```
rowSums(v)
```

```
## the first      second      III
##                 5          7         9
```

```
attributes(rowSums(v))
```

```
## $names
## [1] "the first" "second"   "III"
```

- Generally **names** are maintained
- Sometimes, **dim** is maintained, sometimes not
- All else is stripped

More on names

- The **names** attribute corresponds to the individual elements within a vector

```
names(v)
```

```
## NULL
```

```
names(v) <- letters[1:6]
v
```

```
##           index value
## the first      1     4
## second        2     5
## III           3     6
## attr(,"matrix_mean")
## [1] 3.5
## attr(,"names")
## [1] "a" "b" "c" "d" "e" "f"
```

- Perhaps more straightforward

```
v3a <- c(a = 5, b = 7, c = 12)  
v3a
```

```
##   a   b   c  
##   5   7  12
```

```
names(v3a)
```

```
## [1] "a" "b" "c"
```

```
attributes(v3a)
```

```
## $names  
## [1] "a" "b" "c"
```

Alternatives

```
v3b <- c(5, 7, 12)
names(v3b) <- c("a", "b", "c")
v3b
```

```
##   a   b   c
##   5   7  12
```

```
v3c <- setNames(c(5, 7, 12), c("a", "b", "c"))
v3c
```

```
##   a   b   c
##   5   7  12
```

- Note that `names` is **not** the same thing as `colnames`, but, somewhat confusingly, both work to rename the variables (columns) of a data frame. We'll talk more about why this is momentarily.

Why names might be helpful

```
v
```

```
##           index value
## the first      1     4
## second        2     5
## III           3     6
## attr(,"matrix_mean")
## [1] 3.5
## attr(,"names")
## [1] "a" "b" "c" "d" "e" "f"
```

```
v["b"]
```

```
## b
## 2
```

```
v["e"]
```

```
## e
## 5
```

Implementation of factors

```
fct <- factor(c("a", "a", "b", "c"))
typeof(fct)
```

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

```
attributes(fct)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

```
str(fct)
```

```
## Factor w/ 3 levels "a", "b", "c": 1 1 2 3
```

More manually

```
# First create integer vector
int <- c(1L, 1L, 2L, 3L, 1L, 3L)

# assign some levels
attr(int, "levels") <- c("red", "green", "blue")

# change the class to a factor
class(int) <- "factor"

int
```

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

This can make things tricky

```
age <- factor(sample(c("baby", 1:10), 100, replace = TRUE))  
str(age)
```

```
## Factor w/ 11 levels "1","10","2","3",...: 8 1 9 2 1 1 2 10 9 6 ...
```

```
age
```

```
## [1] 7     1     8     10    1     1     10    9  
## [9] 8     5     4     9     baby  baby  4     2  
## [17] 9     baby  4     2     5     9     10    3  
## [25] 9     baby  10    10    3     3     3     10  
## [33] baby  8     baby  baby  3     baby  baby  baby  
## [41] 1     7     2     4     baby  2     4     7  
## [49] 5     6     9     6     8     8     8     2  
## [57] 1     4     8     3     8     5     7     baby  
## [65] 5     10    8     5     9     5     5     3  
## [73] 6     5     9     4     10    4     1     8  
## [81] 5     baby  baby  10    1     6     1     9  
## [89] 9     3     3     6     baby  10    4     5  
## [97] 3     1     9     8  
## Levels: 1 10 2 3 4 5 6 7 8 9 baby
```

What if we wanted to convert this to numeric?

```
data.frame(age) %>%  
  count(age) %>%  
  mutate(age_numeric = as.numeric(age)) %>%  
  select(starts_with("age"), n)
```

```
##      age age_numeric   n  
## 1      1             1   9  
## 2     10            2  10  
## 3      2             3   5  
## 4      3             4 10  
## 5      4             5   9  
## 6      5             6 11  
## 7      6             7   5  
## 8      7             8   4  
## 9      8             9 11  
## 10     9            10 11  
## 11  baby           11 15
```

These are the integers associated with the factor levels, so **as.numeric()** will not give us the results we want.

Fix

First convert to character, then to numeric (you can ignore the warning in this case)

```
data.frame(age) %>%
  mutate(
    age_chr = as.character(age),
    age_num = ifelse(age_chr == "baby", 0, as.numeric(age_chr))
  ) %>%
  count(age, age_chr, age_num)
```

```
##      age age_chr age_num n
## 1      1       1      1   9
## 2     10      10     10  10
## 3      2       2      2   5
## 4      3       3      3  10
## 5      4       4      4   9
## 6      5       5      5  11
## 7      6       6      6   5
## 8      7       7      7   4
## 9      8       8      8  11
## 10     9       9      9  11
## 11  baby    baby      0  15
```

Implementation of dates

```
date <- Sys.Date()  
typeof(date)
```

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

```
attributes(date)
```

```
## $class  
## [1] "Date"
```

```
attributes(date) <- NULL  
date
```

```
## [1] 19079
```

- This number represents the days passed since January 1, 1970, known as the Unix epoch.

A bit more on classes

Why do these all print different things?

```
summary(mtcars[, 1:2])
```

```
##          mpg              cyl
##  Min.   :10.40   Min.   :4.000 
##  1st Qu.:15.43  1st Qu.:4.000 
##  Median :19.20  Median :6.000 
##  Mean   :20.09  Mean   :6.188 
##  3rd Qu.:22.80  3rd Qu.:8.000 
##  Max.   :33.90  Max.   :8.000 
```

```
summary(gss_cat$marital)
```

```
##      No answer Never married
##                 17             5416
##      Divorced Widowed
##                 3383            1807
```

```
m <- lm(mpg ~ cyl, mtcars)
summary(m)
```

```
## 
## Call:
## lm(formula = mpg ~ cyl, data = mtcars)
## 
## Residuals:
##     Min      1Q  Median      3Q     M
## -4.9814 -2.1185  0.2217  1.0717  7.51
## 
## Coefficients:
##                               Estimate Std. Error t val
## (Intercept) 37.8846    2.0738  18.0
## cyl         -2.8758    0.3224  -8.9
## 
## Pr(>|t|)    
## (Intercept) < 2e-16 ***
## cyl          6.11e-10 ***
## 
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
```

What are the classes?

```
class(mtcars[, 1:2])
```

```
## [1] "data.frame"
```

```
class(gss_cat$marital)
```

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

```
class(m)
```

```
## [1] "lm"
```

S3 methods

When you call `summary()`, it looks for a method (function) for that specific class.

`summary(mtcars[, 1:2])` becomes

`summary.data.frame(mtcars[, 1:2])`

`summary(gss_cat$marital)` becomes

`summary.factor(gss_cat$marital)`

`summary(m)` becomes `summary.lm(m)`

Naming function

Because S3 is so common in R, I recommend against including dots in function names.

`summary.data.frame()` is less clear than it would be if it were `summary.data_frame()`

Classes and methods is not something I'm going to expect you to have a deep knowledge on, but I want you to be aware of it.

Missing values

- Missing values breed missing values

```
NA > 5
```

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

```
NA * 7
```

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

- What about this one?

```
NA == NA
```

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

It is correct because there's no reason to presume that one missing value is or is not equal to another missing value.

When missing values don't propagate

```
NA | TRUE
```

```
## [1] TRUE
```

```
x <- c(NA, 3, NA, 5)
any(x > 4)
```

```
## [1] TRUE
```

How to test missingness?

- We've already seen the following doesn't work

```
x == NA
```

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

- Instead, use `is.na`

```
is.na(x)
```

```
## [1] TRUE FALSE TRUE FALSE
```

Lists

Lists

- Lists are vectors, but not *atomic* vectors
- Fundamental difference – each element can be a different type

```
list("a", 7L, 3.25, TRUE)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] 3.25
##
## [[4]]
## [1] TRUE
```

Lists

- Technically, each element of the list is a vector, possibly atomic
- The prior example included all *scalars*, which are vectors of length 1.
- Lists do not require all elements to be the same length

```
l <- list(  
  c("a", "b", "c"),  
  rnorm(5),  
  c(7L, 2L),  
  c(TRUE, TRUE, FALSE, TRUE))  
l
```

```
## [[1]]  
## [1] "a" "b" "c"  
##  
## [[2]]  
## [1] 0.19253010 -0.97561601 -0.060921  
## [4] 1.65972336 -0.07401147  
##  
## [[3]]  
## [1] 7 2  
##  
## [[4]]  
## [1] TRUE TRUE FALSE TRUE
```

Check the list

```
typeof(l)
```

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

```
attributes(l)
```

```
## NULL
```

```
str(l)
```

```
## List of 4
## $ : chr [1:3] "a" "b" "c"
## $ : num [1:5] 0.1925 -0.9756 -0.0609 1.6597 -0.074
## $ : int [1:2] 7 2
## $ : logi [1:4] TRUE TRUE FALSE TRUE
```

Data frames as lists

- A data frame is just a special case of a list, where all the elements are of the same length.

```
l_df <- list(  
  a = c("red", "blue"),  
  b = rnorm(2),  
  c = c(7L, 2L),  
  d = c(TRUE, FALSE)  
)  
l_df
```

```
data.frame(l_df)
```

```
##           a          b  c      d  
## 1   red  0.3921062 7  TRUE  
## 2 blue  2.0518441 2 FALSE
```

```
## $a  
## [1] "red"  "blue"  
##  
## $b  
## [1] 0.3921062 2.0518441  
##  
## $c  
## [1] 7 2  
##  
## $d  
## [1] TRUE FALSE
```

Subsetting

Lists

A nested list

Lists are often complicated objects. Let's create a somewhat complicated one

```
x <- c(a = 3, b = 5, c = 7)
l <- list(
  x = x,
  x2 = c(x, x),
  x3 = list(
    vect = x,
    squared = x^2,
    cubed = x^3)
)
```

Subsetting lists

Multiple methods

- Most common: `$`, `[`, and `[[`

```
l[1]
```

```
## $x  
## a b c  
## 3 5 7
```

```
typeof(l[1])
```

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

```
l[[1]]
```

```
## a b c  
## 3 5 7
```

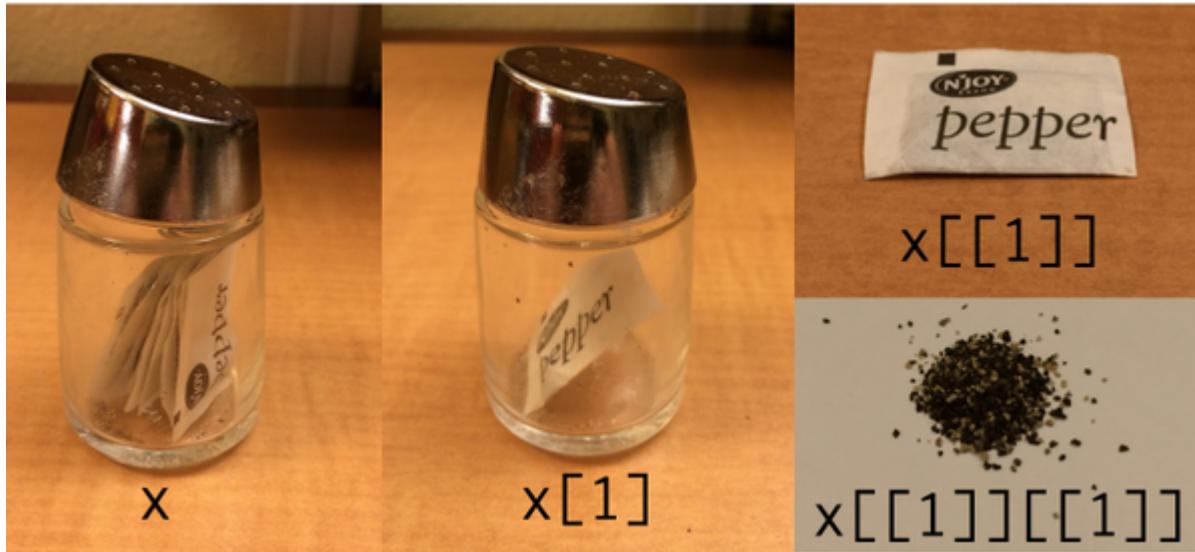
```
typeof(l[[1]])
```

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

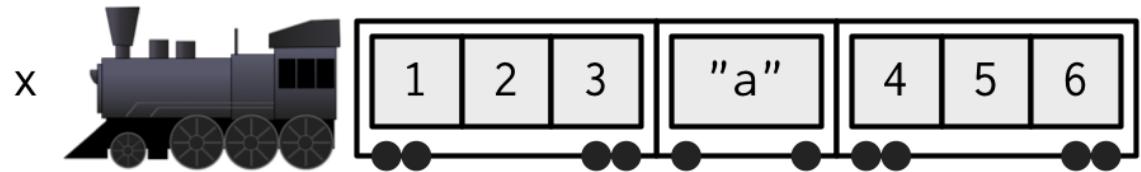
```
l[[1]][["c"]]
```

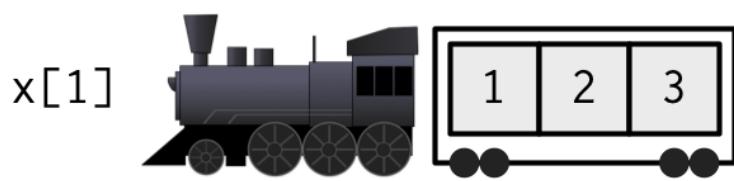
```
## c  
## 7
```

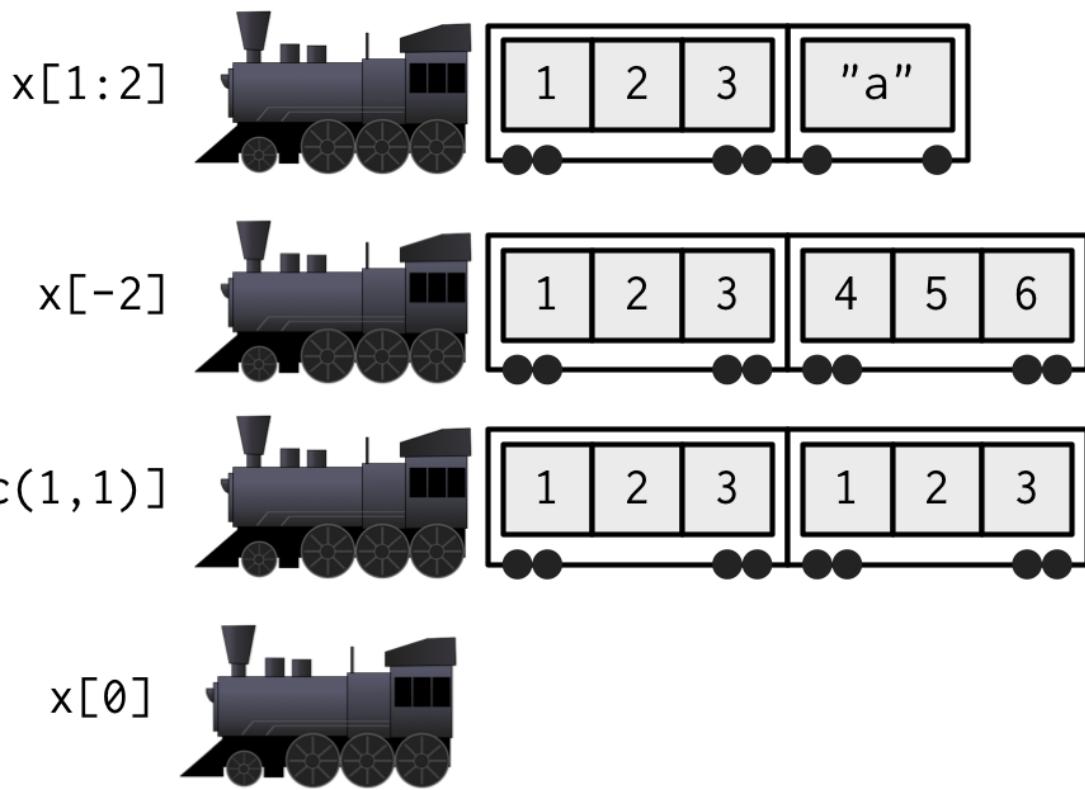
Which bracket to use?



Another analogy







Named list

- Because the elements of the list are named, we can also use `$`, just like with a data frame (which is a list)

```
l$x2
```

```
## a b c a b c  
## 3 5 7 3 5 7
```

```
l$x3
```

```
## $vect  
## a b c  
## 3 5 7  
##  
## $squared  
##   a   b   c  
##   9 25 49  
##  
## $cubed  
##      a      b      c  
##    27 125 343
```

Subsetting nested lists

- Multiple `$` if all named

```
l$x3$squared
```

```
##   a   b   c  
##   9  25  49
```

- Note this doesn't work on named elements of an atomic vector, just the named elements of a list

```
l$x3$squared$b
```

```
## Error in l$x3$squared$b: $ operator is invalid for atomic vectors
```

But we could do something like...

```
l$x3$squared["b"]
```

```
## b  
## 25
```

Alternatives

- You can always use logical
- Indexing works too

```
l[c(TRUE, FALSE, TRUE)]
```

```
## $x
## a b c
## 3 5 7
##
## $x3
## $x3$vect
## a b c
## 3 5 7
##
## $x3$squared
## a b c
## 9 25 49
##
## $x3$cubed
## a b c
## 27 125 343
```

```
l[c(1, 3)]
```

```
## $x
## a b c
## 3 5 7
##
## $x3
## $x3$vect
## a b c
## 3 5 7
##
## $x3$squared
## a b c
## 9 25 49
##
## $x3$cubed
## a b c
## 27 125 343
```

Careful with your brackets

```
l[[c(TRUE, FALSE, FALSE)]]
```

```
## Error in l[[c(TRUE, FALSE, FALSE)]]: recursive indexing failed at level
```

- Why doesn't the above work?

Subsetting in multiple dimensions

- Generally we deal with 2d data frames
- If there are two dimensions, we separate the `[` subsetting with a comma

```
head(mtcars)
```

```
##          mpg cyl disp  hp drat
## Mazda RX4     21.0   6 160 110 3.90
## Mazda RX4 Wag 21.0   6 160 110 3.90
## Datsun 710    22.8   4 108  93 3.85
## Hornet 4 Drive 21.4   6 258 110 3.08
## Hornet Sportabout 18.7   8 360 175 3.15
## Valiant       18.1   6 225 105 2.76
##             wt qsec vs am gear carb
## Mazda RX4     2.620 16.46  0  1    4    4
## Mazda RX4 Wag 2.875 17.02  0  1    4    4
## Datsun 710    2.320 18.61  1  1    4    1
## Hornet 4 Drive 3.215 19.44  1  0    3    1
## Hornet Sportabout 3.440 17.02  0  0    3    2
## Valiant       3.460 20.22  1  0    3    1
```

```
mtcars[3, 4]
```

Empty indicators

- An empty indicator implies "all"

Select the entire fourth column

```
mtcars[ ,4]
```

```
##   [1] 110 110  93 110 175 105 245   62   95 123
##  [11] 123 180 180 180 205 215 230   66   52  65
##  [21]  97 150 150 245 175   66   91 113 264 175
##  [31] 335 109
```

Select the entire 4th row

```
mtcars[4, ]
```

```
##          mpg cyl disp  hp drat    wt
## Hornet 4 Drive 21.4    6 258 110 3.08 3.215
##                 qsec vs am gear carb
## Hornet 4 Drive 19.44   1  0    3    1
```

Data types returned

- By default, each of the prior will return a vector, which itself can be subset

The following are equivalent

```
mtcars[4, c("mpg", "hp")]
```

```
##                 mpg   hp
## Hornet 4 Drive 21.4 110
```

```
mtcars[4, ][c("mpg", "hp")]
```

```
##                 mpg   hp
## Hornet 4 Drive 21.4 110
```

Return a data frame

- Often, you don't want the vector returned, but rather the modified data frame.
- Specify `drop = FALSE`

```
mtcars[ ,4]
```

```
##   [1] 110 110  93 110 175 105 245  62  95 123  
## [11] 123 180 180 180 205 215 230  66  52  65  
## [21]  97 150 150 245 175   66  91 113 264 175  
## [31] 335 109
```

```
mtcars[ ,4, drop = FALSE]
```

```
##                               hp  
## Mazda RX4                 110  
## Mazda RX4 Wag              110  
## Datsun 710                  93  
## Hornet 4 Drive              110  
## Hornet Sportabout            175  
## Valiant                     105
```

tibbles

- Note dropping the data frame attribute is the default for a **data.frame** but NOT a **tibble**.

```
mtcars_tbl <- tibble::as_tibble(mtcars)
mtcars_tbl[ ,4]
```

```
## # A tibble: 32 × 1
##       hp
##   <dbl>
## 1    110
## 2    110
## 3     93
## 4    110
## 5    175
## 6    105
## 7    245
## 8     62
## 9     95
## 10    123
## # ... with 22 more rows
```

You can override this

```
mtcars_tbl[ ,4, drop = TRUE]
```

```
## [1] 110 110  93 110 175 105 245  62  95 123  
## [11] 123 180 180 180 205 215 230  66  52  65  
## [21]  97 150 150 245 175   66   91 113 264 175  
## [31] 335 109
```

More than two dimensions

- Depending on your applications, you may not run into this much

```
array <- 1:12
dim(array) <- c(2, 3, 2)
array
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]     1     3     5
## [2,]     2     4     6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]     7     9    11
## [2,]     8    10    12
```

Subset array

Select just the second matrix

```
array[ , ,2]
```

```
##      [,1] [,2] [,3]
## [1,]     7     9    11
## [2,]     8    10    12
```

Select first column of each matrix

```
array[ ,1, ]
```

```
##      [,1] [,2]
## [1,]     1     7
## [2,]     2     8
```

Back to lists

Why are they so useful?

- Much more flexible
- Often returned by functions, for example, `lm`

```
m <- lm(mpg ~ hp, mtcars)
str(m)
```

```
## List of 12
## $ coefficients : Named num [1:2] 30.0989 -0.0682
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "hp"
## $ residuals     : Named num [1:32] -1.594 -1.594 -0.954 -1.194 0.541 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 7...
## $ effects       : Named num [1:32] -113.65 -26.046 -0.556 -0.852 0.67 ...
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "hp" "" ""
## $ rank          : int 2
## $ fitted.values: Named num [1:32] 22.6 22.6 23.8 22.6 18.2 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 7...
## $ assign         : int [1:2] 0 1
## $ qr            :List of 5
##   ..$ qr      : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ... 127 / 129
```

Summary

- Atomic vectors must all be the same type
 - implicit coercion occurs if not (and you haven't specified the coercion explicitly)
- Lists are also vectors, but not atomic vectors
 - Each element can be of a different type and length
 - Incredibly flexible, but often a little more difficult to get the hang of, particularly with subsetting

Next time

Loops with base R