# Exploring data 2

# Other R objects: Matrices and lists

# Matrices

A matrix is like a data frame, but all the values in all columns must be of the same class (e.g., numeric, character). (Another way you can think of it is as a "wrapped" vector.)

Matrices can be faster and more memory-efficient than data frames. Also, a lot of statistical methods within R code is implemented using linear algebra and other mathematical techniques based on matrices.

# Matrices

We can use the `matrix()` function to construct a matrix:

```
foo <- matrix(1:10, ncol = 5)
foo
```

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

# Matrices

The `as.matrix()` function is used to convert an object to a matrix:

```
foo <- data.frame(col_1 = 1:2, col_2 = 3:4,
                  col_3 = 5:6, col_4 = 7:8,
                  col_5 = 9:10)
foo <- as.matrix(foo)
foo
```

```
##      col_1 col_2 col_3 col_4 col_5
## [1,]     1     3     5     7     9
## [2,]     2     4     6     8    10
```

# Matrices

You can index matrices with square brackets, just like data frames:

```
foo[1, 1:2]
```

```
## col_1 col_2
##     1     3
```

You cannot, however, use `dplyr` functions with matrices:

```
foo %>% filter(col_1 == 1)
```

```
Error in UseMethod("filter_") :
  no applicable method for 'filter_' applied to an object of
  class "c('matrix', 'integer', 'numeric')"
```

# Lists

# Lists

Lists are the "kitchen sink" of R objects. They can be used to keep together a variety of different R objects of different classes, dimensions, and structures in a single R object.

Because there are often cases where an R operation results in output that doesn't have a simple structure, lists can be a very useful way to output complex output from an R function.

Most lists are not "tidy" data. However, we'll cover some ways that you can easily "tidy" some common list objects you might use a lot in your R code, including the output of fitting linear and generalized linear models.

# Lists

```
example_list <- list(a = sample(1:10, 5),
                     b = tibble(letters = letters[1:3],
                                numbers = 1:3))
example_list
```

```
## $a
## [1]  8  6  3  7 10
##
## $b
## # A tibble: 3 x 2
##   letters numbers
##   <chr>     <int>
## 1 a             1
## 2 b             2
## 3 c             3
```

# Indexing lists

To pull an element out of a list, you can either use $ or [[]] indexing:

```
example_list$a
```

```
## [1]  8  6  3  7 10
```

```
example_list[[2]]
```

```
## # A tibble: 3 x 2
##   letters numbers
##   <chr>     <int>
## 1 a             1
## 2 b             2
## 3 c             3
```

# Indexing lists

To access a specific value within a list element we can index the element using double, double brackets:

```
example_list[["b"]][["numbers"]]
```

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

Again, we can index using names or numeric indices:

```
example_list[["b"]][[1]]
```

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

# Exploring lists

If an R object is a list, running `class` on the object will return "list":

```
class(example_list)
```

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

Often, lists will have names for each element (similar to column names for a dataframe). You can get the names of all elements of a list using the `names` function:

```
names(example_list)
```

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

# Exploring lists

The `str` function is also useful for exploring the structure of a list object:

```
str(example_list)
```

```
## List of 2
##  $ a: int [1:5] 8 6 3 7 10
##  $ b:Classes 'tbl_df', 'tbl' and 'data.frame':   3 obs. of  2 variables:
##   ..$ letters: chr [1:3] "a" "b" "c"
##   ..$ numbers: int [1:3] 1 2 3
```

# Exploring lists

A list can even contain other lists. We can use the `str` function to see the structure of a list:

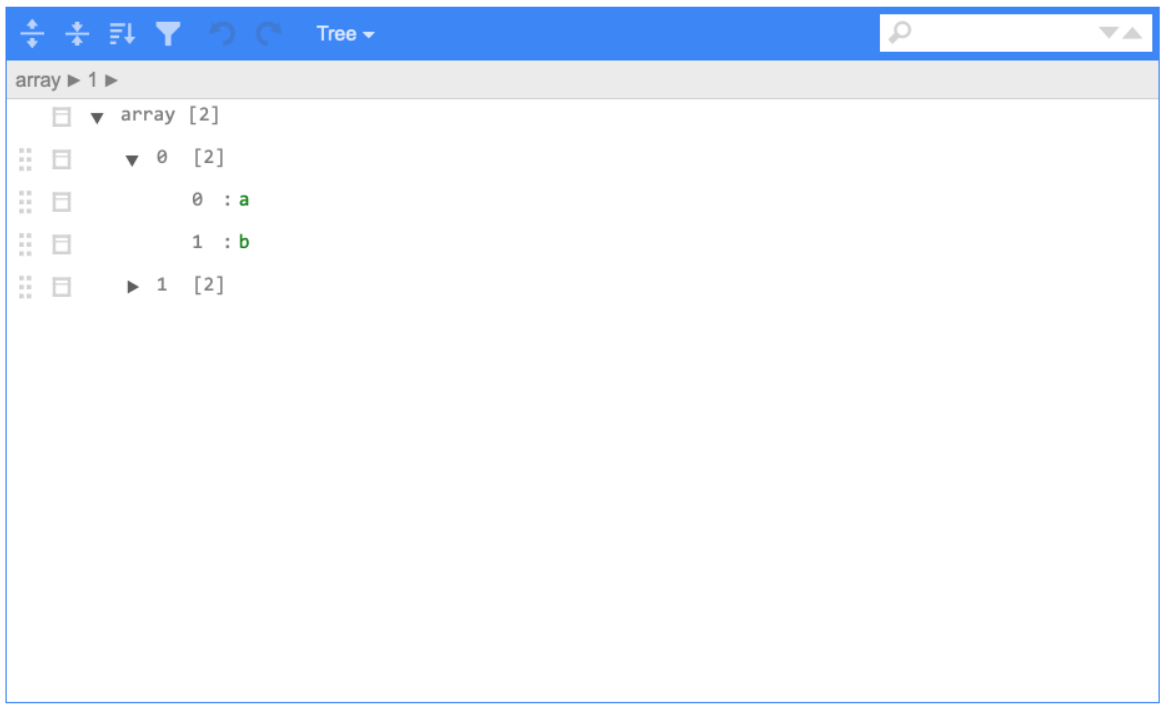
```
a_list <- list(list("a", "b"), list(1, 2))

str(a_list)
```

```
## List of 2
##  $ :List of 2
##   ..$ : chr "a"
##   ..$ : chr "b"
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
```

# Exploring lists

Using `str` to print out the list's structure doesn't produce the easiest to digest output. We can use the `jsonedit` function from the `listviewer` package to create a widget in the Viewer pane to more esily explore our list.

```
library(listviewer)
jsonedit(a_list)
```



# Lists versus dataframes

As a note, a dataframe is actually just a very special type of list. It is a list where every element (column in the dataframe) is a vector of the same length, and the object has a special attribute specifying that it is a dataframe.

```
example_df <- tibble(letters = letters[1:3],
                     number = 1:3)
class(example_df)
```

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

```
is.list(example_df)
```

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

# In-course Exercise

We'll take a break now to do section 1 of the In-course Exercise for Chapter 7.

# Simple statistical tests in R

# Statistical tests

R has many different functions for different statistical tests.

So far, I have not found a statistical test I wanted to perform that did not have a function to do it in R.

# Statistical tests

We'll start with a pretty simple test called the "Shapiro-Wilk test of normality."

The **null hypothesis** for this test is that the data follow a normal distribution. If the p-value from running the test is lower than a specified threshold (often 0.05), then you reject the null hypothesis.

# Statistical tests

Often, a parametric test will require the assumption that one or more variables follow a normal distribution. This test can help check that assumption.

If you find that the distribution does not seem to be normal, you can make adjustments, like transforming the variable or using a non-parametric test (see here for a nice overview).

# Statistical tests

We'll use this test as an example, but it has two characteristics that are common to most simple statistical tests functions in R:

1. It inputs a vector (or vectors, for some tests), rather than a tidy dataframe.
2. It outputs a list object, rather than a dataframe.

We'll talk about both of these characteristics and how we can still work statistical tests into a "tidy" workflow, where we're keeping the data in a tidy dataframe format most of the time.

# Statistical tests

To show how this test works, let's simulate some data that we know are normal.

There are several functions in R that let you simulate a vector of data from a specified distribution. For a normal distribution, the function is `rnorm`.

For this function, you'll specify how many values you want in the vector (`n`) and the values of the mean (`mean`) and standard deviation (`sd`) of the normal distribution.

# Statistical tests

For example, you can run this to create a vector with a random sample of 1,000 values from a normal distribution with mean 200 and standard deviation 50:

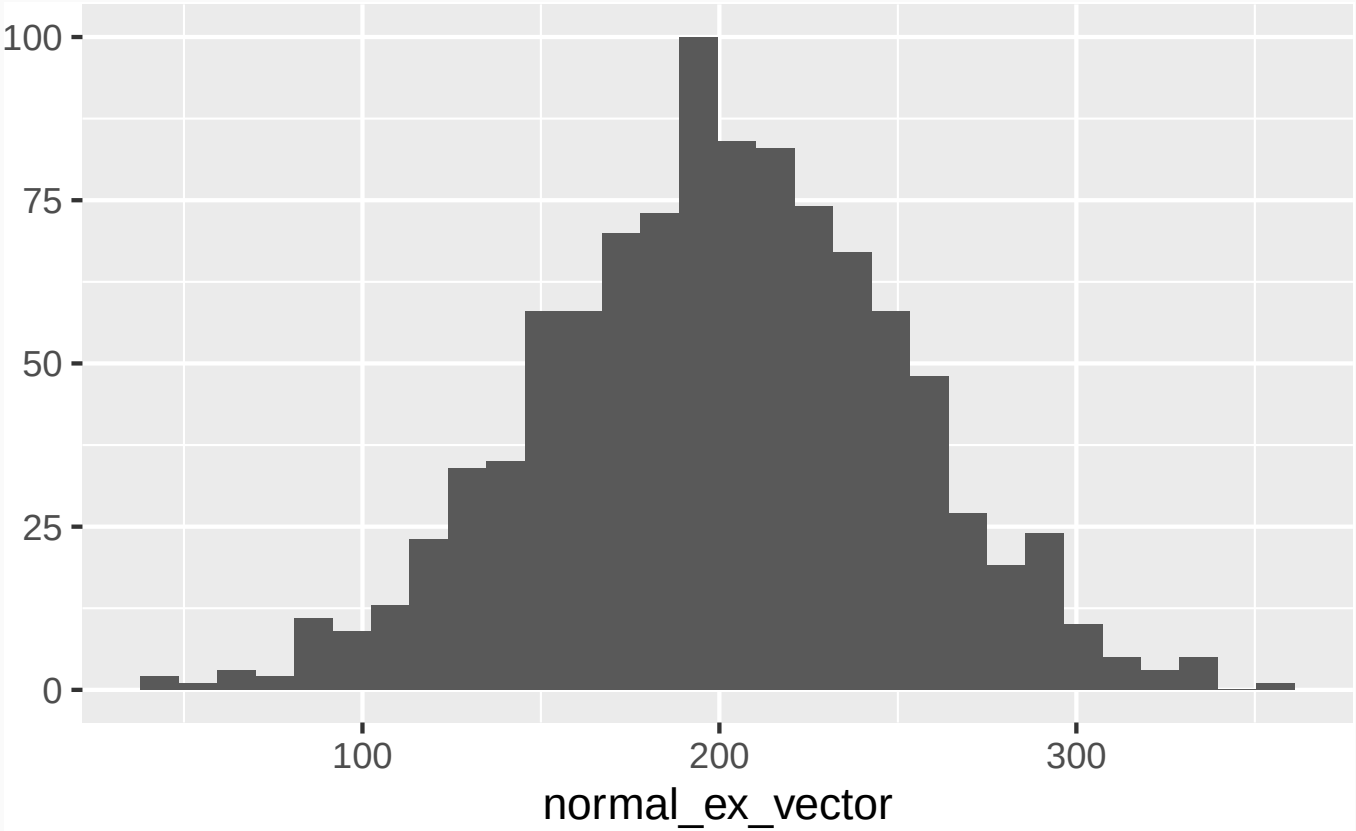
```
normal_ex_vector <- rnorm(n = 1000, mean = 200, sd = 50)
head(normal_ex_vector)
```

```
## [1] 260.3657 223.1650 142.3312 161.2520 198.9883 266.8476
```

# Statistical tests

Let's check the histogram of this data. Since this data in in a vector, not a dataframe, we should use `qplot` from `ggplot2` instead of `ggplot`:

```
qplot(normal_ex_vector, geom = "histogram")
```



# Statistical tests

It looks pretty normal, but let's run the test. The function to run the test is `shapiro.test`. Its only argument is `x`, the vector that you want to check.

```
shapiro.test(x = normal_ex_vector)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  normal_ex_vector
## W = 0.99908, p-value = 0.9111
```

# Statistical tests

The default for this function is to print out the results in a way that's easy to read on the screen. However, you can also save the results from the test in a new object:

```
ex_sw_result <- shapiro.test(x = normal_ex_vector)
```

# Statistical tests

This object has a special class called “htest”:

```
class(ex_sw_result)
```

```
## [1] "htest"
```

Really, though, this is just a special kind of list:

```
is.list(ex_sw_result)
```

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

# Statistical tests

If you run `str` on the object, you can see it has all the information that is usually printed by the function tucked away in different slots of the list:

```
str(ex_sw_result)
```

```
## List of 4
##  $ statistic: Named num 0.999
##   ..- attr(*, "names")= chr "W"
##  $ p.value  : num 0.911
##  $ method   : chr "Shapiro-Wilk normality test"
##  $ data.name: chr "normal_ex_vector"
##  - attr(*, "class")= chr "htest"
```

# Statistical tests

There is a package called `broom` that can pull this information out and format it as a tidy dataframe. For example:

```
library(broom)
tidy(ex_sw_result)
```

```
## # A tibble: 1 x 3
##   statistic p.value method
##       <dbl>   <dbl> <chr>
## 1     0.999   0.911 Shapiro-Wilk normality test
```

This seems unexciting for this example, but trust me, it turns out that being able to do this is **very** exciting.

# Statistical tests

Now, let's look at how this process would work if you were starting with a vector that was a column in a tidy dataframe. Since you're using tidyverse tools, you'll probably find you want to do this often.

We'll create a very simple dataframe with only this column:

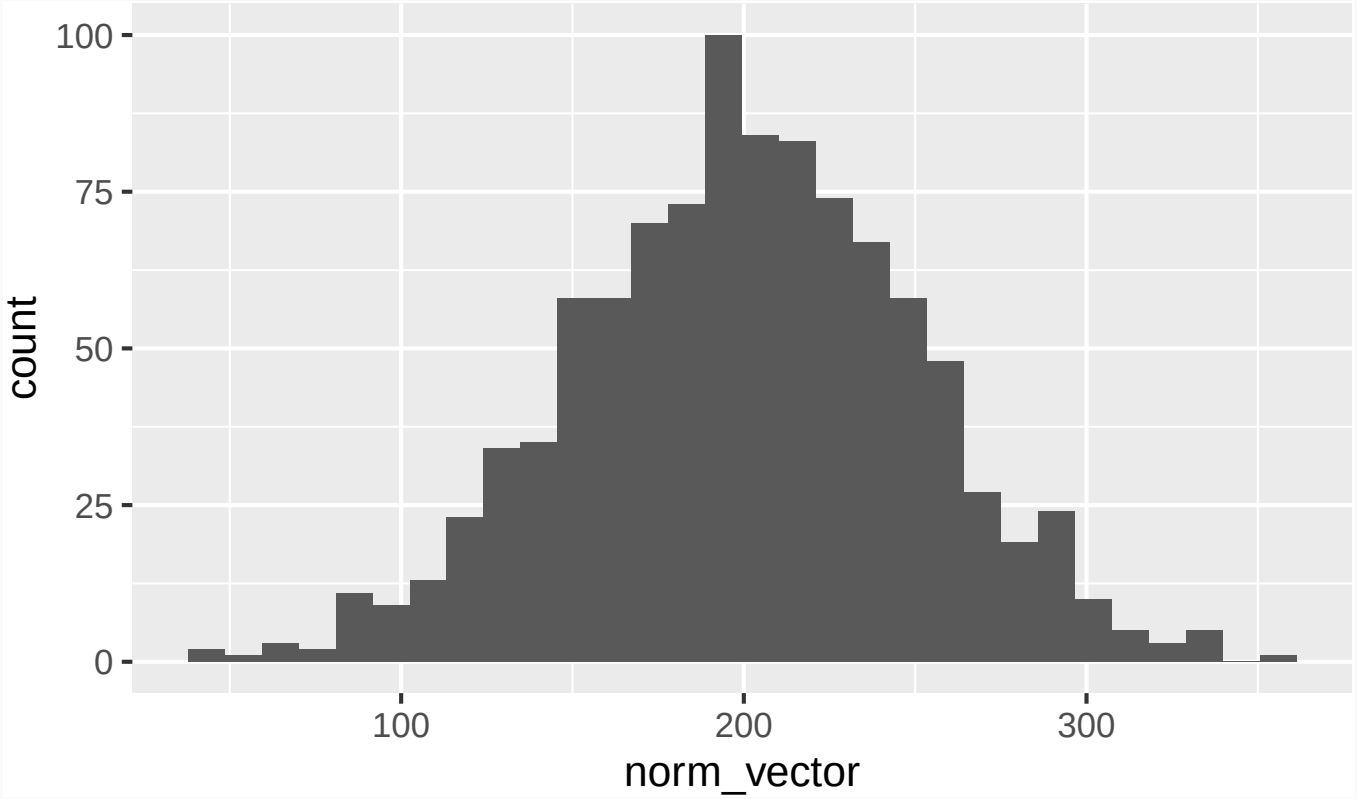
```
ex_df <- tibble(norm_vector = normal_ex_vector)
ex_df %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##   norm_vector
##         <dbl>
## 1        260.
## 2        223.
## 3        142.
```

# Statistical tests

Now you can use ggplot to make the histogram:

```
ggplot(ex_df, aes(x = norm_vector)) +
  geom_histogram()
```



# Statistical tests

To fit the test, you'll need to be able to pull this vector out of the dataframe. To do that, you can use the `pluck` function from the `purrr` package in a pipeline. That function "plucks" out a single column as a vector. For example:

```
library(purrr)
ex_df %>%
  pluck("norm_vector") %>%
  head()
```

```
## [1] 260.3657 223.1650 142.3312 161.2520 198.9883 266.8476
```

# Statistical tests

With that function, you can pipe right into the Shapiro-Wilk test function:

```
ex_df %>%
  pluck("norm_vector") %>%
  shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.99908, p-value = 0.9111
```

# Statistical tests

Now just add on the `tidy` function to get the test output in a tidy dataframe, and you're back to your typical format!

```
ex_df %>%
  pluck("norm_vector") %>%
  shapiro.test() %>%
  tidy()
```

```
## # A tibble: 1 x 3
##   statistic p.value method
##       <dbl>   <dbl> <chr>
## 1     0.999   0.911 Shapiro-Wilk normality test
```

# Statistical tests

Now let's look at some real data. The variable dataframe of the `atlas1006` dataset in the `microbiome` library has a column on `diversity`.

We might want to test if diversity is different by gender, nationality, or other factors. To pick which statistical tests to use to check those questions, though, it will help to know if this variable is normally distributed.

# Statistical tests

The `atlas1006` data is stored in a `phyloseq` object (think of it as a fancy type of list). To extract a dataframe with characteristics of the samples, you'll need to use `get_variable` (which we can pipe into if we want):

```
library(microbiome)
data(atlas1006)
atlas1006 %>%
  get_variable() %>%
  slice(1:3)
```

```
##   age    sex nationality DNA_extraction_method project divers
## 1  28   male          US                  <NA>       1      5
## 2  24 female          US                  <NA>       1      6
## 3  52   male          US                  <NA>       1      5
##     bmi_group subject time   sample
## 1 severeobese       1    0 Sample-1
## 2       obese       2    0 Sample-2
## 3        lean       3    0 Sample-3
```

# Statistical tests

There are a few people that they measure several times, so there are more rows than the number of people they measure:

```
atlas1006 %>%
  get_variable() %>%
  nrow()
```

```
## [1] 1151
```

# Statistical tests

We probably just want to work with the first measurement from each person, so let's use `filter` to filter to samples with a "time" value of 0 (first measurement):

```
atlas1006 %>%
  get_variable() %>%
  filter(time == 0) %>%
  nrow()
```

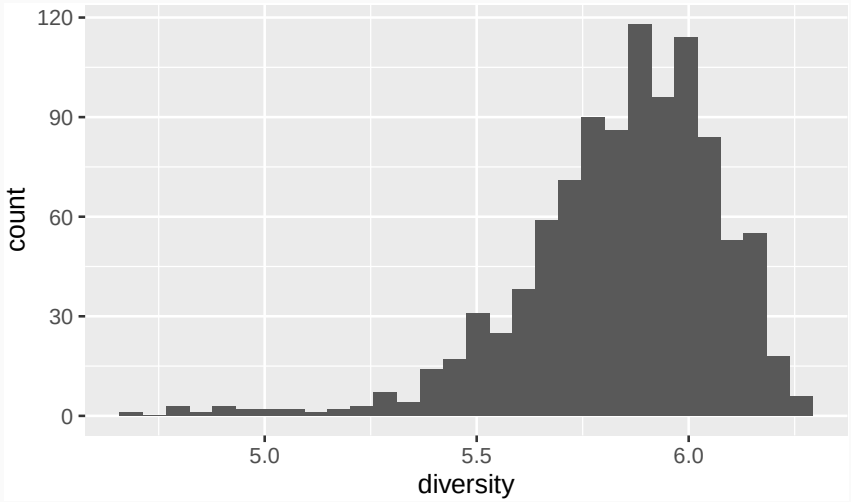
```
## [1] 1006
```

This looks right.

# Statistical tests

We can use a histogram to visually check the normality:

```
atlas1006 %>%
  get_variable() %>%
  filter(time == 0) %>%
  ggplot(aes(x = diversity)) +
  geom_histogram()
```



# Statistical tests

To extract the column on `diversity` as a vector, we can use `pluck`:

```
atlas1006 %>%
  get_variable() %>%
  filter(time == 0) %>%
  pluck("diversity") %>%
  head()
```

```
## [1] 5.76 6.06 5.50 5.87 5.89 5.53
```

# Statistical tests

Now add on the Shapiro test function:

```
atlas1006 %>%
  get_variable() %>%
  filter(time == 0) %>%
  pluck("diversity") %>%
  shapiro.test()
```

```
##
##  Shapiro-Wilk normality test
##
## data:  .
## W = 0.93439, p-value < 2.2e-16
```

# Statistical tests

And finally add on the `tidy` function:

```
atlas1006 %>%
  get_variable() %>%
  filter(time == 0) %>%
  pluck("diversity") %>%
  shapiro.test() %>%
  tidy()
```

```
## # A tibble: 1 x 3
##   statistic  p.value method
##       <dbl>    <dbl> <chr>
## 1     0.934 1.29e-20 Shapiro-Wilk normality test
```

# Find out more about statistical tests in R

I won't be teaching in this course how to find the correct statistical test. That's something you'll hopefully learn in a statistics course.

There are also a variety of books that can help you with this, including some that you can access free online through CSU's library. One servicable introduction is "Statistical Analysis with R for Dummies".

# In-course exercise

We'll take a break now to do sections 2 and 3 of the In-course Exercise for Chapter 7.

# Regression models

# World Cup example

In the World Cup data, we may wonder if the number of tackles is associated with the time the player played. Let's start by grabbing just the variables we care about (we'll be using `Position` later, so we'll include that):

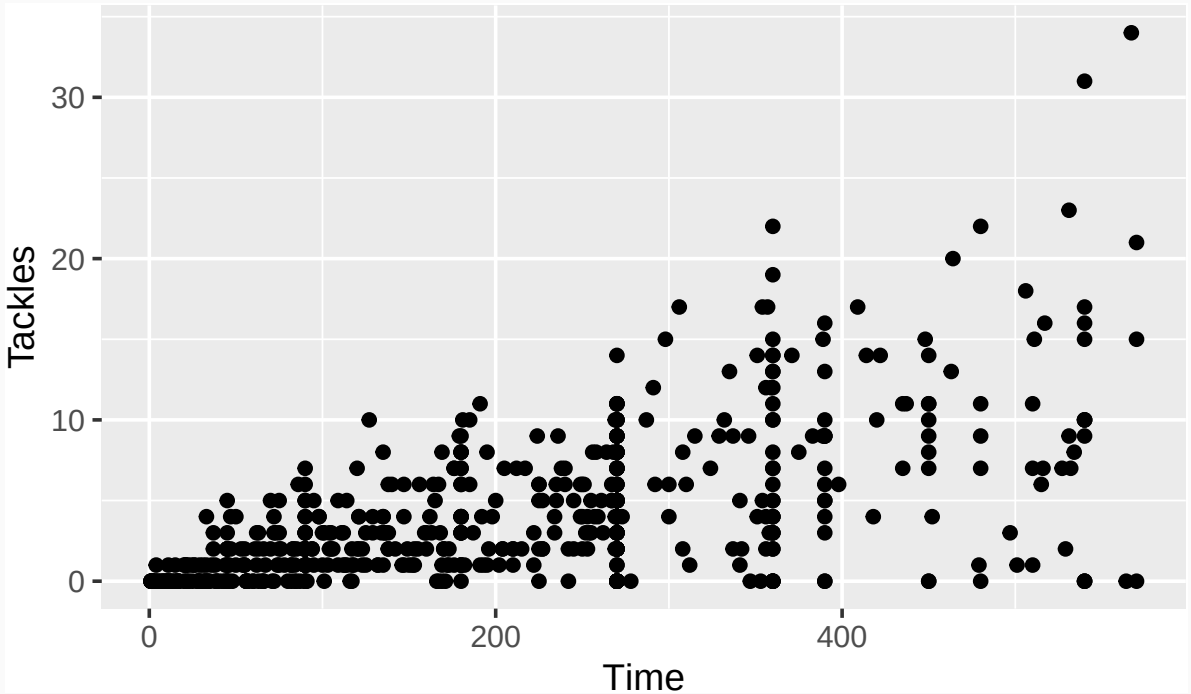
```
library(faraway)
data(worldcup)
worldcup <- worldcup %>%
  select(Time, Tackles, Position)
worldcup %>% slice(1:3)
```

```
##   Time Tackles   Position
## 1   16       0 Midfielder
## 2  351      14 Midfielder
## 3  180       6   Defender
```

# World Cup example

We can start by plotting the relationship between the time a player played and the number of tackles they had:

```
library(ggplot2)
ggplot(worldcup, aes(Time, Tackles)) +
  geom_point()
```



# World Cup example

There does indeed seem to be an association. Next, we might want to test this using some kind of statistical model or test.

Let's start by fitting a linear regression model, to see if there's evidence that tackles tend to change (increase or decrease) as the player's time played increases.

(In a bit, we'll figure out that a linear model might not be the best way to model this, since the number of tackles is a count, rather than a variable with a normal distribution, but bear with me. . . )

# Formula structure

*Regression models* can be used to estimate how the expected value of a *dependent variable* changes as *independent variables* change.

In R, regression formulas take this structure:

```
## Generic code
[response variable] ~ [indep. var. 1] +  [indep. var. 2] + ...
```

Notice that ~ used to separate the independent and dependent variables and the + used to join independent variables. This format mimics the statistical notation:

$$Y_i \sim X_1 + X_2 + \cdots + \epsilon_i$$

You will use this type of structure in R for a lot of different function calls, including those for linear models (`lm`) and generalized linear models (`glm`).

# Linear models

To fit a linear model, you can use the function `lm()`. Use the `data` option to specify the dataframe from which to get the vectors. You can save the model as an object.

```
tackle_model <- lm(Tackles ~ Time, data = worldcup)
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where:

- $Y_i$ : Number of tackles for player $i$ (dependent variable)
- $X_{1,i}$ : Minutes played by player $i$ (independent variable)

# Linear models

A few things to point out:

- By default, an intercept is fit to the model.
- If you specify a dataframe using `data` in the `lm` call, you can write the model formula using just the column names for the independent variable(s) and dependent variable you want, without quotation marks around those names.
- You can save the output of fitting the model to an R object (if you don't, a summary of the fit model will be print out at the console).

# Model objects

The output from fitting a model using `lm` is a list object:

```
class(tackle_model)
```

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

This list object has a lot of different information from the model, including overall model summaries, estimated coefficients, fitted values, residuals, etc.

```
names(tackle_model)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

# Model objects and broom

This list object is not in a "tidy" format. However, you can use functions from broom to pull "tidy" dataframes from this model object.

For example, you can use the glance function to pull out a one-row tidy dataframe with model summaries.

```
glance(tackle_model)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <int>  <dbl> <dbl> <dbl>
## 1     0.373         0.372  3.69      353. 4.31e-62     2 -1620. 3246. 3259.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

# Model objects and `broom`

If you want to get the estimated model coefficients (and some related summaries) instead, you can use the `tidy` function to do that:

```
tidy(tackle_model)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   0.110    0.265        0.415 6.78e- 1
## 2 Time          0.0195   0.00104     18.8   4.31e-62
```

This output includes, for each model term, the **estimated coefficient** (`estimate`), its **standard error** (`std.error`), the **test statistic** (for `lm` output, the statistic for a test with the null hypothesis that the model coefficient is zero), and the associated **p-value** for that test (`p.value`).

# Model objects and broom

Some of the model output have a value for each original observation (e.g., fitted values, residuals). You can use the augment function to add those elements to the original data used to fit the model:

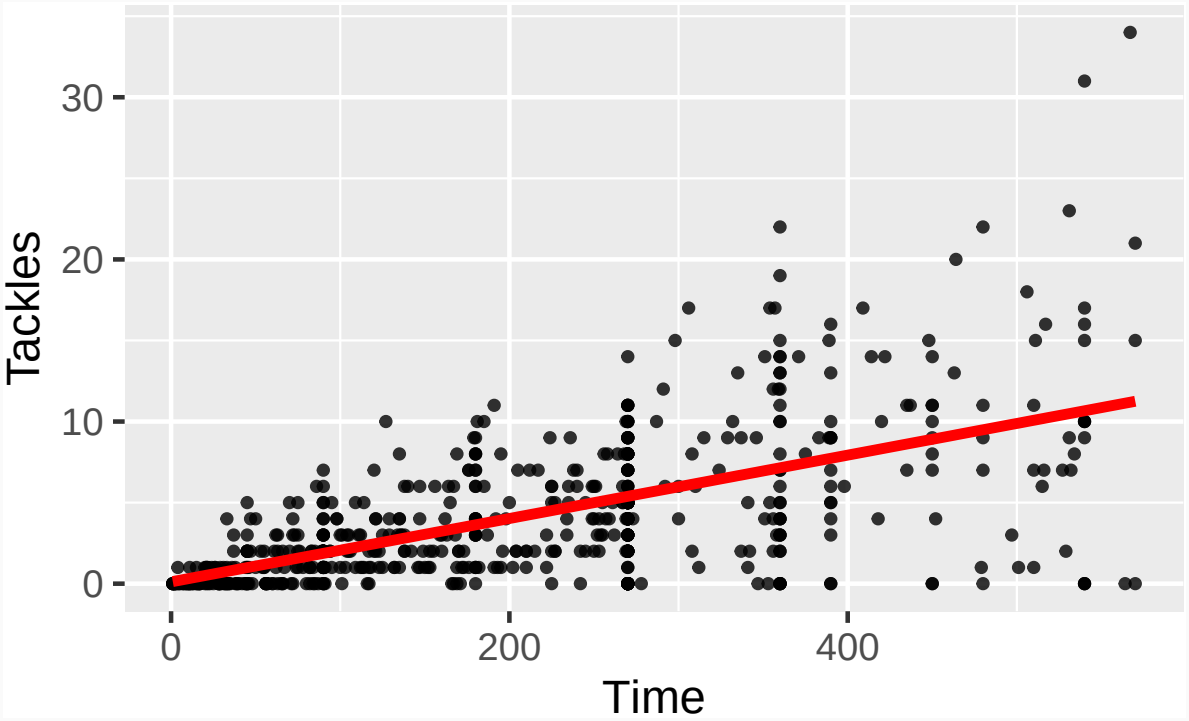
```
augment(tackle_model) %>%
  slice(1:2)
```

```
## # A tibble: 2 x 10
##   .rownames Tackles  Time .fitted .se.fit .resid    .hat .sigma .cooksd
##   <chr>       <int> <int>   <dbl>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>
## 1 Abdoun          0    16   0.423   0.251 -0.423 0.00464   3.69 3.07e-5
## 2 Abe            14   351   6.97    0.212  7.03  0.00329   3.68 6.01e-3
## # ... with 1 more variable: .std.resid <dbl>
```

# Model objects and `broom`

One important use of this `augment` output is to create a plot with both the original data and a line showing the fit model (via the predictions):

```
augment(tackle_model) %>%
  ggplot(aes(x = Time, y = Tackles)) +
  geom_point(size = 0.8, alpha = 0.8) +
  geom_line(aes(y = .fitted), color = "red", size = 1.2)
```



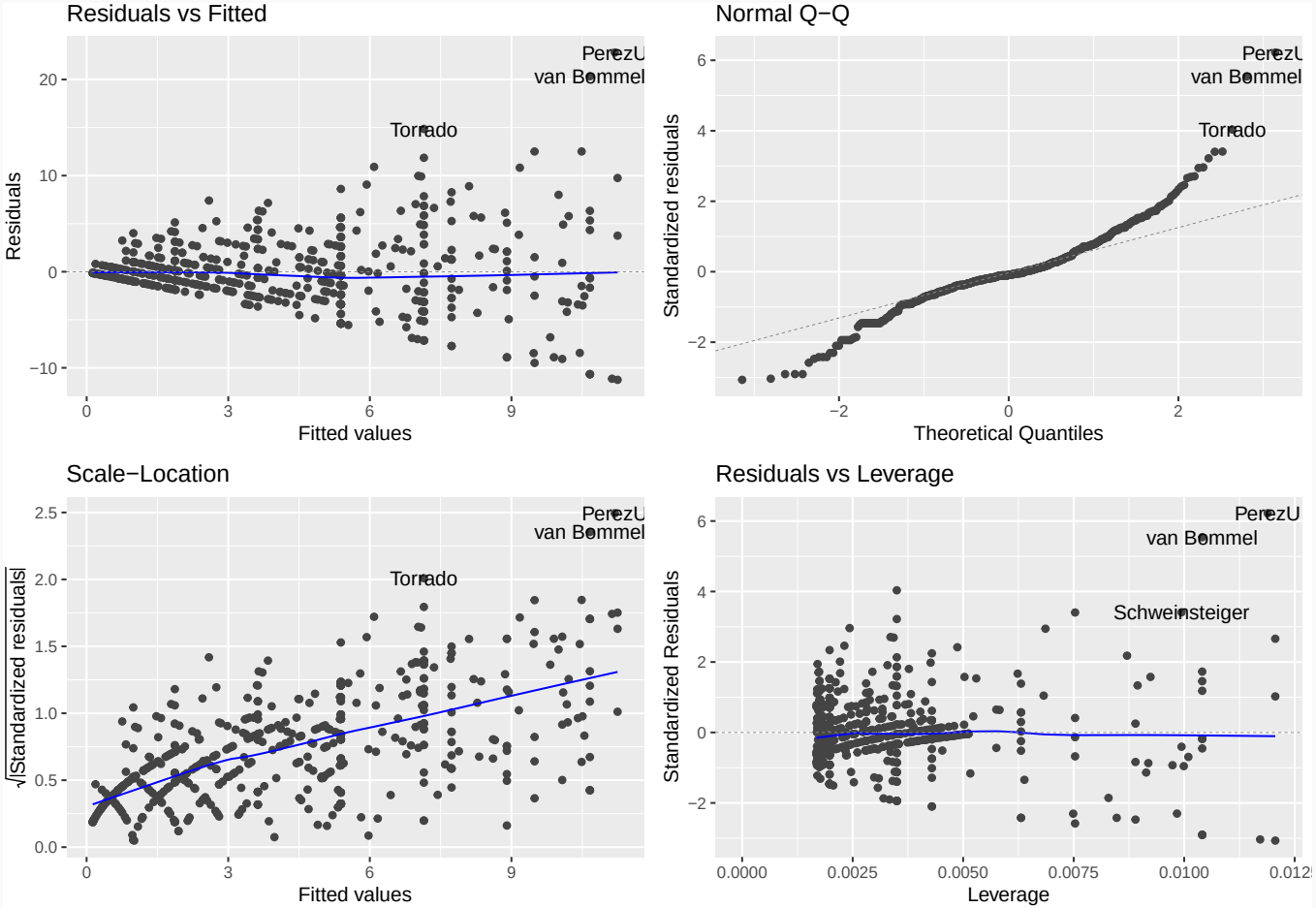
# Model objects and `autoplot`

There is a function called `autoplot` in the `ggplot2` package that will check the class of an object and then create a certain default plot for that class. Although the generic `autoplot` function is in the `ggplot2` package, for `lm` and `glm` objects, you must have the `ggfortify` package installed and loaded to be able to access the methods of `autoplot` specifically for these object types.

If you have the package that includes an `autoplot` method for a specific object type, you can just run `autoplot` on the objects name and get a plot that is considered a useful default for that object type. For `lm` objects, `autoplot` gives small graphics with model diagnostic plots.

# Model objects and `autoplot`

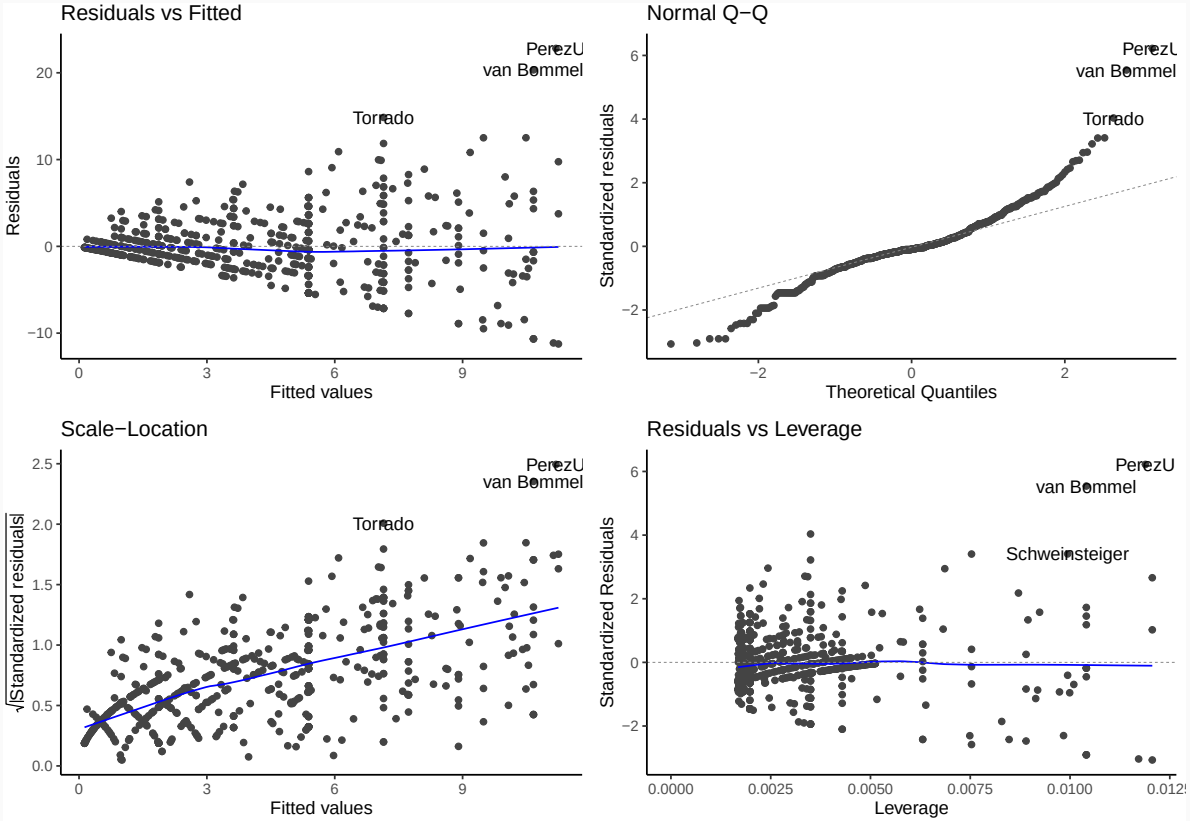
```
library(ggfortify)
autoplot(tackle_model)
```



# Model objects and `autoplot`

The output from `autoplot` is a `ggplot` object, so you can add elements to it as you would with other `ggplot` objects:

```
autoplot(tackle_model) +
  theme_classic()
```

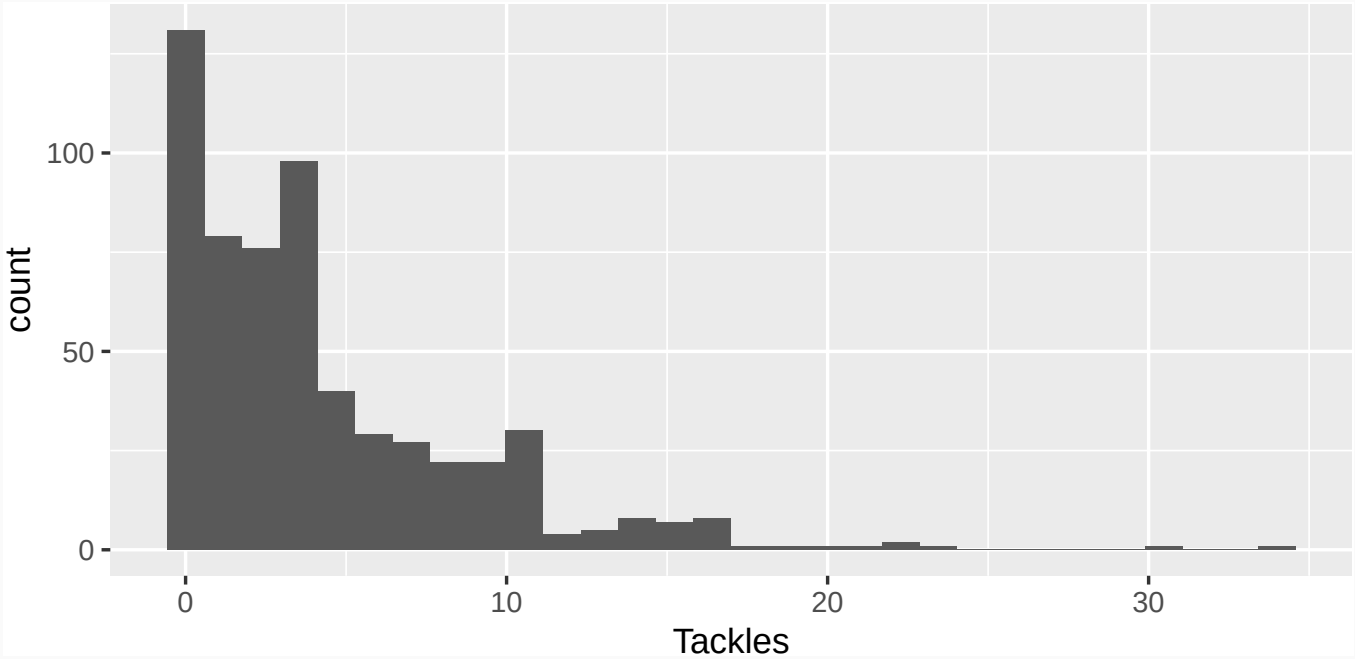


# Regression models

In this case, these diagnostics clearly show that there are some problems with using a linear regression model to fit this data.

Many of these issues arise because the outcome (dependent) variable doesn't follow a normal distribution.

```
ggplot(worldcup, aes(x = Tackles)) +
  geom_histogram()
```



# Regression models

A better model, therefore, might be one where we assume that `Tackles` follows a Poisson distribution, rather than a normal distribution. (For variables that represent counts, this will often be the case.)

In the a little bit, we'll look at **generalized linear models**, which let us extend the idea of a linear model to situations where the dependent variable follows a distribution other than the normal distribution.

# In-course exercise

We'll take a break now to do section 4 of the In-Course Exercise for Chapter 7.

# Fitting a model with a factor

You can also use binary variables or factors (i.e., categorical variables) as independent variables in regression models:

```
tackles_model_2 <- lm(Tackles ~ Position, data = worldcup)
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where $X_{1,i}$ : Position of player $i$

# Fitting a model with a factor

If there are more than one levels to the factor, then the model will fit a separate value for each level of the factor above the first level (which will serve as a baseline):

```
levels(worldcup$Position)
```

```
## [1] "Defender"   "Forward"    "Goalkeeper" "Midfielder"
```

```
tidy(tackles_model_2)
```

```
## # A tibble: 4 x 5
##   term               estimate std.error statistic  p.value
##   <chr>                 <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)           5.46      0.315    17.3   1.14e-54
## 2 PositionForward      -3.44      0.480    -7.17  2.20e-12
## 3 PositionGoalkeeper   -5.43      0.787    -6.91  1.27e-11
## 4 PositionMidfielder   -0.300     0.426    -0.705 4.81e- 1
```

# Fitting a model with a factor

The intercept is the expected (average) value of the outcome (`Tackles`) for the first level of the factor. Each other estimate gives the expected difference between the value of the outcome for this first level of `Position` and one of the other levels of the factor.

```
levels(worldcup$Position)
```

```
## [1] "Defender"   "Forward"    "Goalkeeper" "Midfielder"
```

```
tidy(tackles_model_2)
```

```
## # A tibble: 4 x 5
##   term               estimate std.error statistic  p.value
##   <chr>                 <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)           5.46      0.315    17.3   1.14e-54
## 2 PositionForward      -3.44      0.480    -7.17  2.20e-12
## 3 PositionGoalkeeper   -5.43      0.787    -6.91  1.27e-11
## 4 PositionMidfielder   -0.300     0.426    -0.705 4.81e- 1
```

# Linear models versus GLMs

You can fit a variety of models, including linear models, logistic models, and Poisson models, using generalized linear models (GLMs).

For linear models, the only difference between `lm` and `glm` is how they're fitting the model (least squares versus maximum likelihood). You should get the same results regardless of which you pick.

# Linear models versus GLMs

For example:

```
glm(Tackles ~ Time, data = worldcup) %>%
  tidy()
```

```
## # A tibble: 2 x 5
##   term         estimate std.error statistic  p.value
##   <chr>           <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   0.110    0.265        0.415 6.78e- 1
## 2 Time          0.0195   0.00104     18.8   4.31e-62
```

```
lm(Tackles ~ Time, data = worldcup) %>%
  tidy()
```

```
## # A tibble: 2 x 5
##   term         estimate std.error statistic  p.value
##   <chr>           <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   0.110    0.265        0.415 6.78e- 1
## 2 Time          0.0195   0.00104     18.8   4.31e-62
```

# GLMs

You can fit other model types with `glm()` using the `family` option:

| Model type | `family` option |
|---|---|
| Linear | `family = gaussian(link = 'identity')` |
| Logistic | `family = binomial(link = 'logit')` |
| Poisson | `family = poisson(link = 'log')` |

# GLM example

For example, say we wanted to fit a GLM, but specifying a Poisson distribution for the outcome (and a log link) since we think that `Tackles` might be distributed with a Poisson distribution:

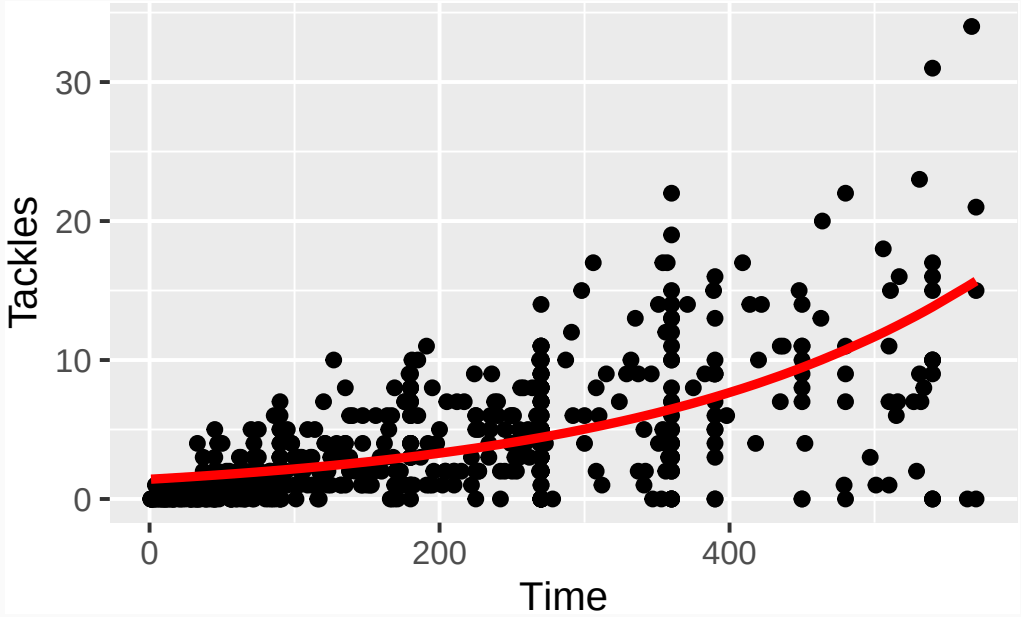
```
tackle_model_3 <- glm(Tackles ~ Time, data = worldcup,
                      family = poisson(link = "log"))
tackle_model_3 %>%
  tidy()
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)  0.350    0.0443        7.90 2.85e- 15
## 2 Time         0.00422  0.000129     32.8  1.81e-235
```

# GLM example

Here are the predicted values from this model (red line):

```
tackle_model_3 %>%
  augment() %>%
  mutate(.fitted = exp(.fitted)) %>%
  ggplot(aes(x = Time, y = Tackles)) +
  geom_point() +
  geom_line(aes(y = .fitted), color = "red", size = 1.2)
```



# Formula structure

There are some conventions that can be used in R formulas. Common ones include:

| Convention | Meaning |
|---|---|
| `I()` | calculate the value inside before fitting (e.g., `I(x1 + x2)`) |
| `:` | fit the interaction between two variables (e.g., `x1:x2`) |
| `*` | fit the main effects and interaction for both variables (e.g., `x1*x2` equals `x1 + x2 + x1:x2`) |
| `.` | fit all variables other than the response (e.g., `y ~ .`) |
| `-` | do not include a variable (e.g., `y ~ . - x1`) |
| `1` | intercept (e.g., `y ~ 1`) |

# To find out more

Great resources to find out more about using R for basic statistics:

- Statistical Analysis with R for Dummies, Joseph Schmuller (free online through our library; Chapter 14 covers regression modeling)
- The R Book, Michael J. Crawley (free online through our library; Chapter 14 covers regression modeling, Chapters 10 and 13 cover linear and generalized linear regression modeling)
- R for Data Science (Section 4)

If you want all the details about fitting linear models and GLMs in R, Faraway's books are fantastic (more at level of Master's in Applied Statistics):

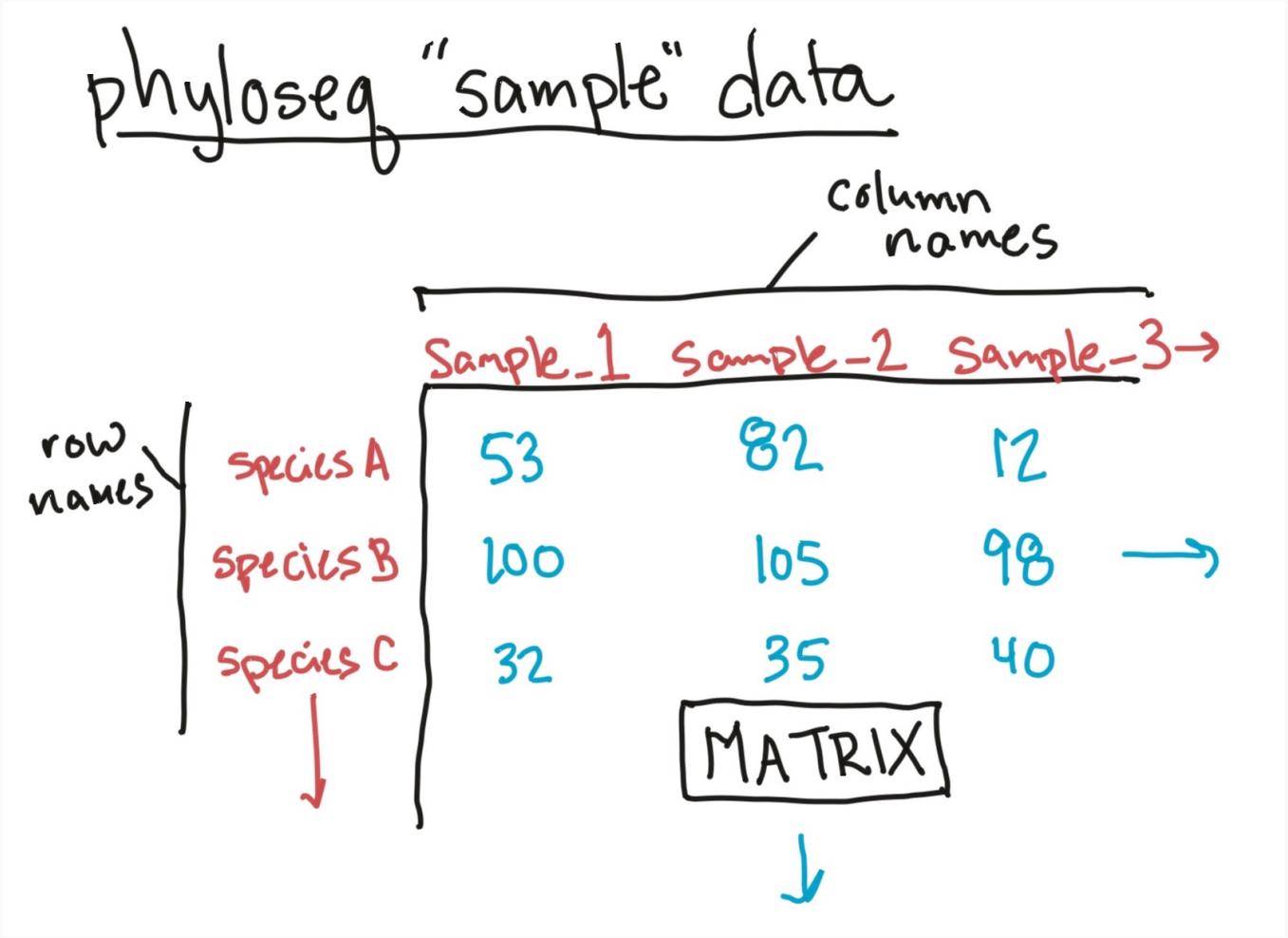
- Linear Models with R, Julian Faraway (also freely available online through our library)
- Extending the Linear Model with R, Julian Faraway (available in hardcopy through our library)

# In-course exercise

We'll take a break now to do section 5 of the In-Course Exercise for Chapter 7.

# Nesting and mapping

# phyloseq sample data




The "phyloseq" object class has a "sample" slot, with a matrix with prevalence for each bacteria in each sample.

# phyloseq sample data

You can use the `get_sample` accessor function to extract this data:

```
library("microbiome")
data("atlas1006")
atlas_sample_data <- atlas1006 %>%
  get_sample()
```

# phyloseq sample data

You can check that this is a matrix, with column names giving sample number and rownames giving bacteria species:

```
atlas_sample_data %>% is.matrix()
```

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

```
atlas_sample_data %>% colnames() %>% head(n = 3)
```

```
## [1] "Sample-1" "Sample-2" "Sample-3"
```

```
atlas_sample_data %>% row.names() %>% head()
```

```
## [1] "Actinomycetaceae"                "Aerococcus"
## [3] "Aeromonas"                       "Akkermansia"
## [5] "Alcaligenes faecalis et rel." "Allistipes et rel."
```

# phyloseq sample data

You can use square bracket indexing to check the top left corner of the sample data:

```
atlas_sample_data[1:6, 1:3]
```

```
##                                 Sample-1 Sample-2 Sample-3
## Actinomycetaceae                       0        0        0
## Aerococcus                             0        0        0
## Aeromonas                              0        0        0
## Akkermansia                           21       36      475
## Alcaligenes faecalis et rel.           1        1        1
## Allistipes et rel.                    72      127       34
```

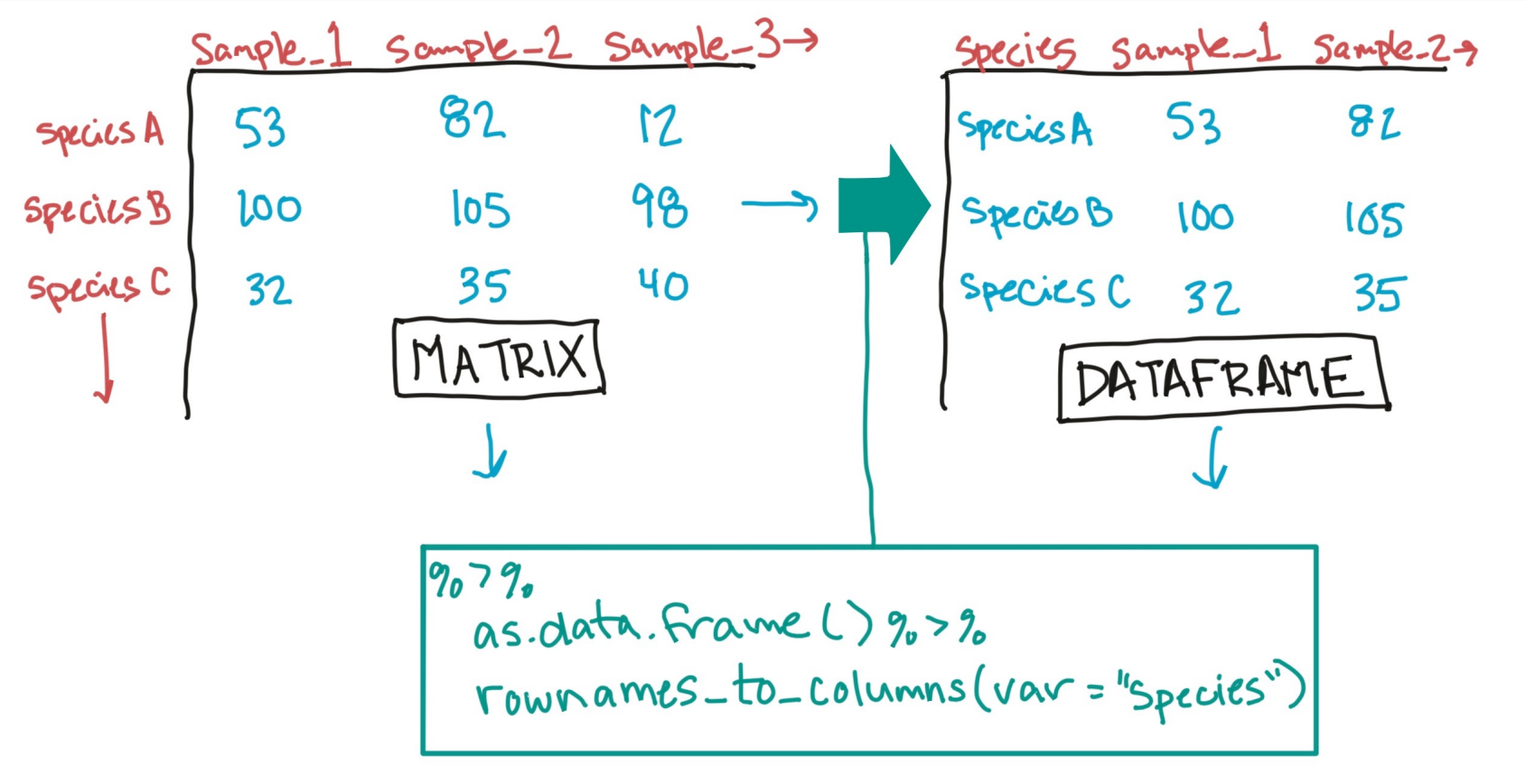
# Tidying phyloseq sample data

To tidy this data, we need to:

1. Change to a data frame
2. Move row names into a column
3. Pivot longer so that column names are in their own column as values

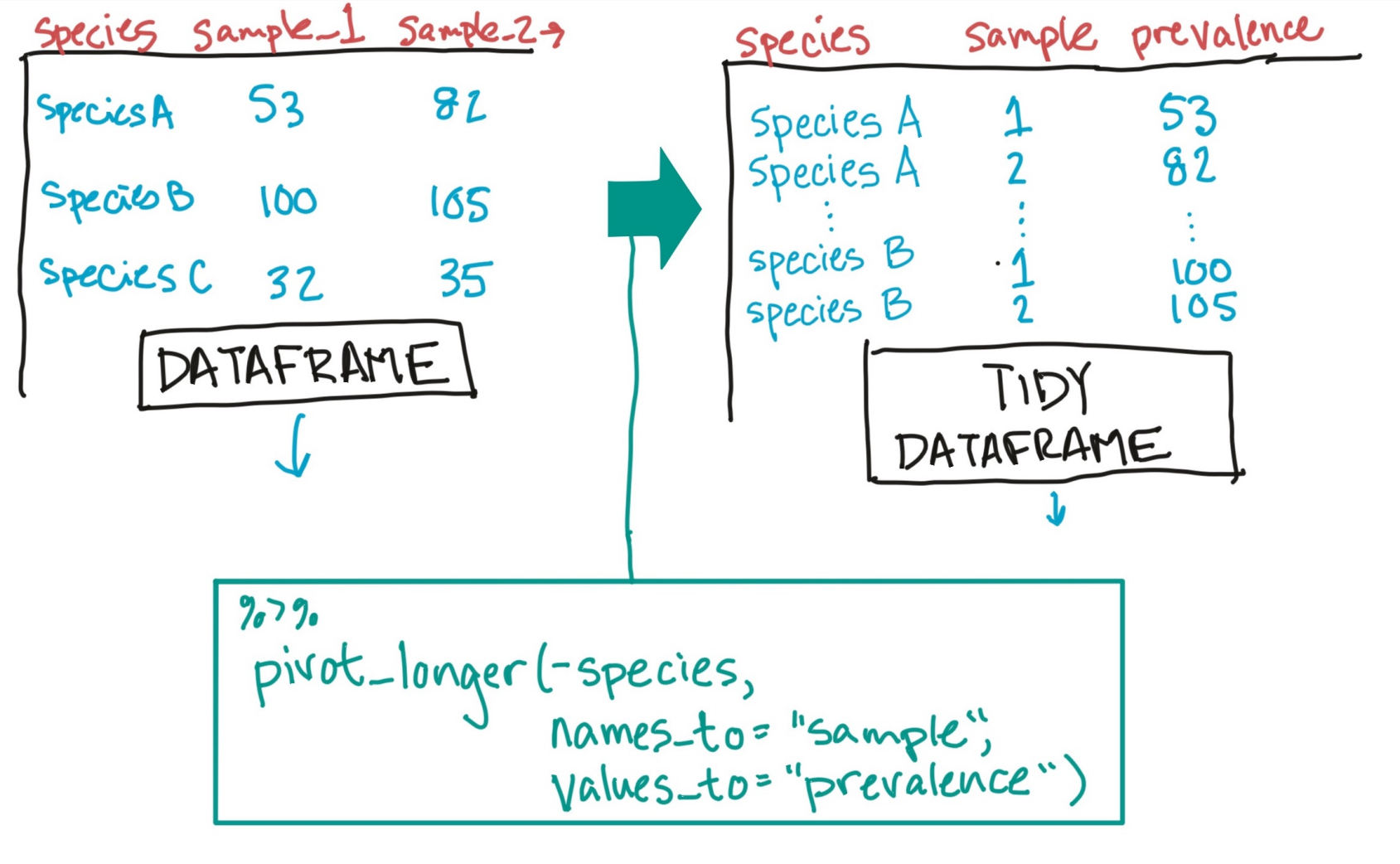
# Tidying phyloseq sample data

Change to a data frame and move row names into a column:



# Tidying phyloseq sample data

Pivot longer so that column names are in their own column as values:



# Tidying phyloseq sample data

Here is everything in code:

```
library(tibble)
library(tidyr)

tidy_samples <- atlas1006 %>%
  get_sample() %>%
  as.data.frame() %>%
  rownames_to_column(var = "species") %>%
  pivot_longer(-species,
               names_to = "sample",
               values_to = "prevalence")
```

# Tidying phyloseq sample data

Here's what the beginning of the tidy data looks like:

```
tidy_samples %>%
  slice(1:5)
```

```
## # A tibble: 5 x 3
##   species          sample   prevalence
##   <chr>            <chr>         <dbl>
## 1 Actinomycetaceae Sample-1          0
## 2 Actinomycetaceae Sample-2          0
## 3 Actinomycetaceae Sample-3          0
## 4 Actinomycetaceae Sample-4          0
## 5 Actinomycetaceae Sample-5          0
```

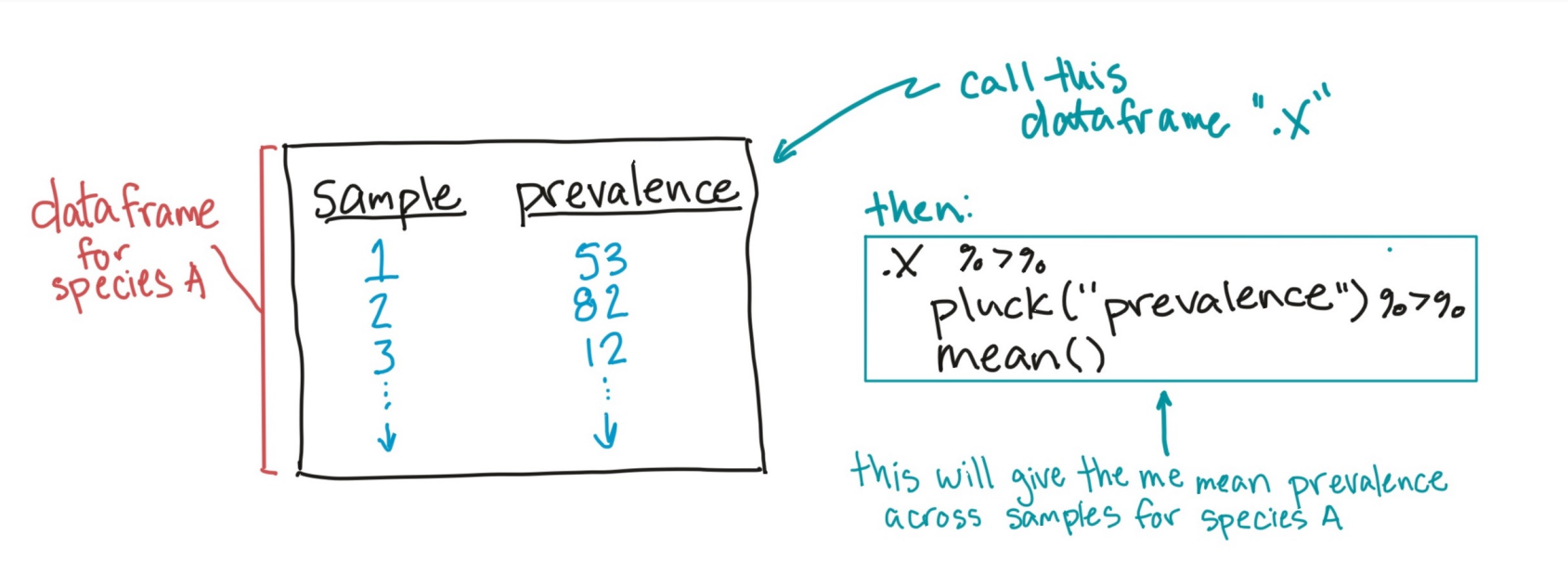
# Nesting and mapping

The first step, with nesting and mapping, is to decide what you'd do to a subsample—the dataframe that you'd get if you filtered to the rows just for one grouping factor (for example, bacteria species).

# Nesting and mapping

As a simple example, say you want to get the mean of each bacteria's prevalence across all samples.

Start by thinking about how you would calculate the mean prevalence for **one** species of bacteria if you had a subset of the dataframe rows for just that species.



# Nesting and mapping

For example, say that you created a subset of the data that only had the rows for the species "Allistipes et rel.":

```
allistipes <- tidy_samples %>%
  filter(species == "Allistipes et rel.")

allistipes %>%
  slice(1:5)
```

```
## # A tibble: 5 x 3
##   species              sample   prevalence
##   <chr>                <chr>         <dbl>
## 1 Allistipes et rel. Sample-1         72
## 2 Allistipes et rel. Sample-2        127
## 3 Allistipes et rel. Sample-3         34
## 4 Allistipes et rel. Sample-4        344
## 5 Allistipes et rel. Sample-5         50
```

# Nesting and mapping

You could determine the mean of prevalence by "plucking" the column measuring prevalence and then taking the mean of that vector:

```
allistipes %>%
  pluck("prevalence") %>%
  mean()
```

```
## [1] 199.5248
```

# Nesting and mapping

Once you've figured out this "recipe", you can **nest** the full dataframe by the grouping factor (e.g., bacteria species) and then **map** this recipe across the subsetted dataframe for each value of the grouping factor.

# Nested dataframe

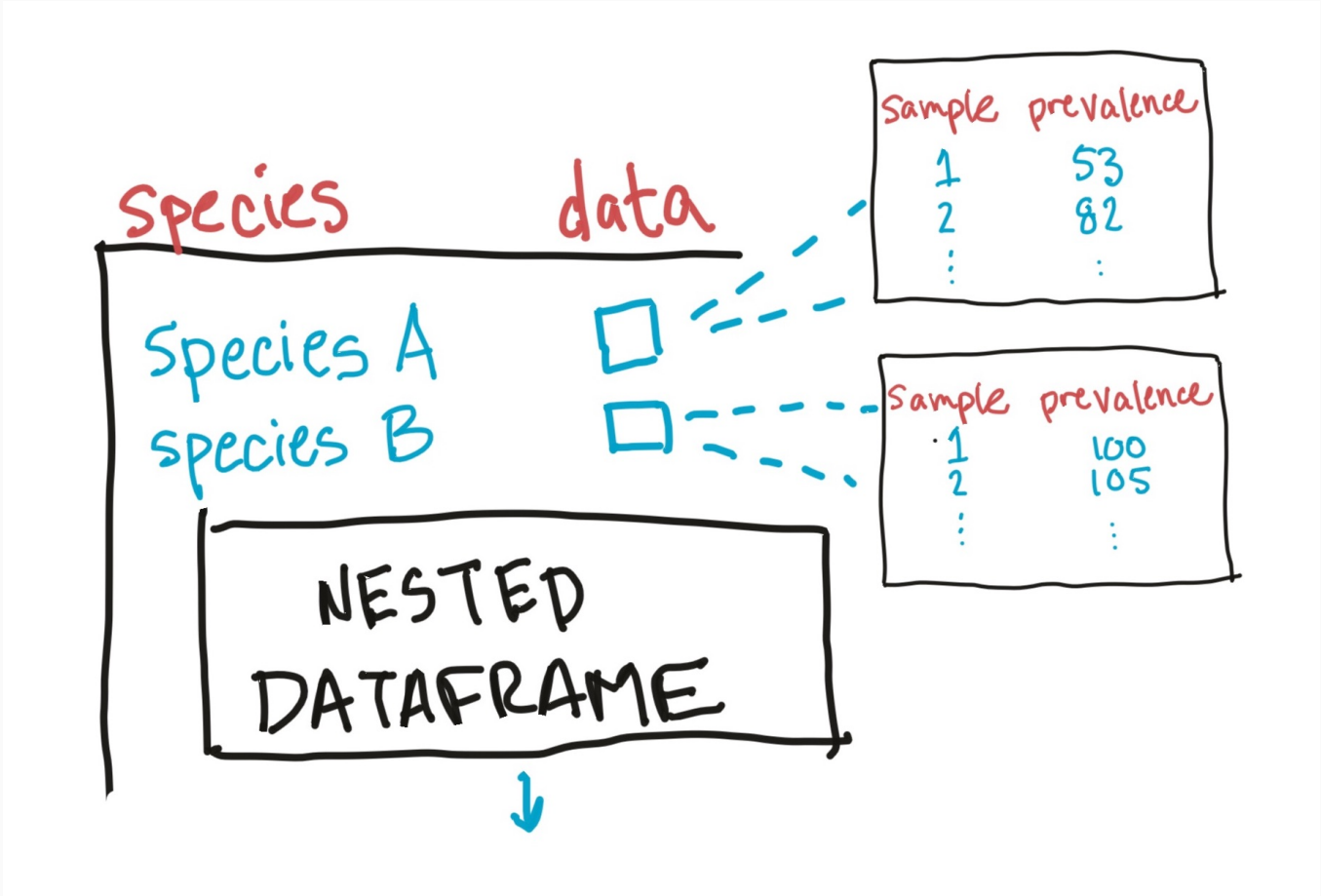
A nested dataframe is a fancy type of tibble.

For classic dataframes, each column must be a **vector**. For a nested dataframe, some of the columns can be **list-columns**, where each element is a more complex object than just a vector.

The elements in one of these list-columns can be a dataframe or a statistical model output object (or any other kind of list).

# Nested dataframe

Here's an example where the list-column ("data") contains a dataframe for each bacterial species, with the prevalence measured for each sample for that bacteria.



# Nested dataframe

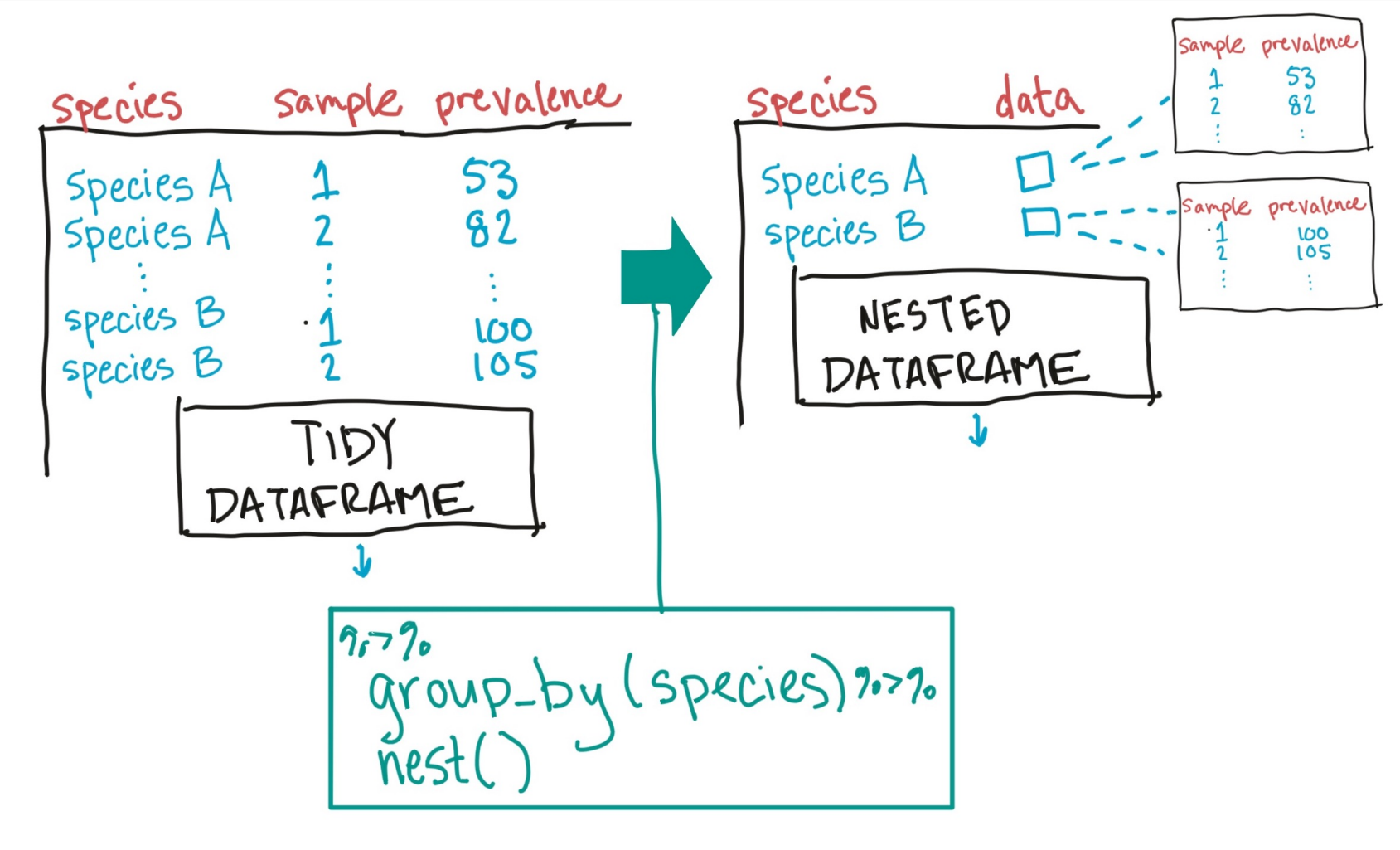
Because a list-column packs in a lot more than a typical column, it will print out a little differently in R. For example, here the "data" column stores a dataframe for each bacteria sample:

```
## # A tibble: 3 x 2
## # Groups:   species [3]
##   species                  data
##   <chr>           <list<df[,2]>>
## 1 Actinomycetaceae    [1,151 x 2]
## 2 Aerococcus          [1,151 x 2]
## 3 Aeromonas           [1,151 x 2]
```

You can see that this element is a dataframe and its dimensions, but not values in it.

# Nested dataframe

To created this type of nested dataframe, you can `group_by` a grouping value (e.g., bacteria species) and then `nest`:



# Nested dataframe

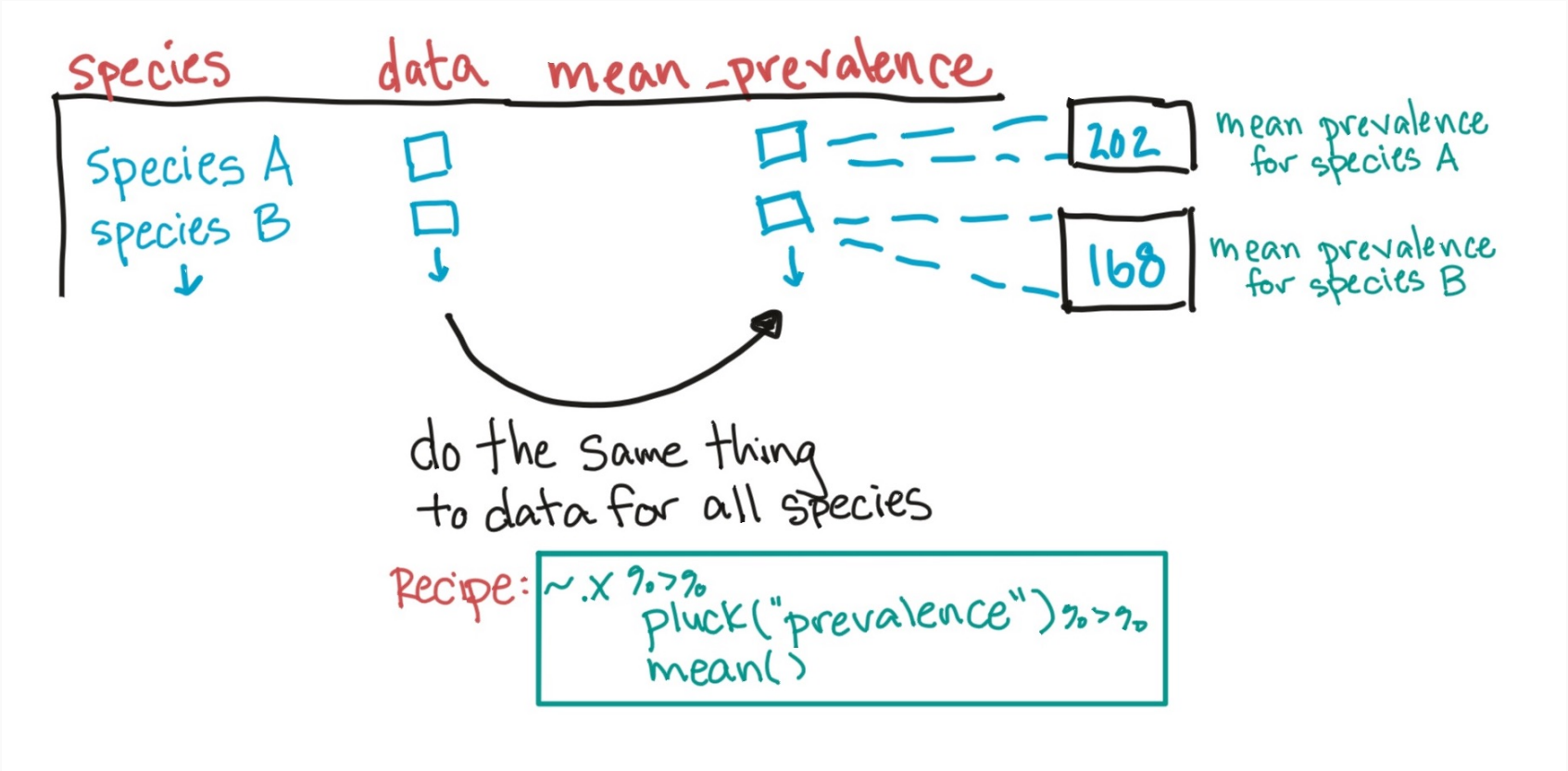
Here's the code to do that:

```
nested_samples <- tidy_samples %>%
  group_by(species) %>%
  nest()

nested_samples %>% head(3)
```

```
## # A tibble: 3 x 2
## # Groups:   species [3]
##   species                      data
##   <chr>              <list<df[,2]>>
## 1 Actinomycetaceae    [1,151 x 2]
## 2 Aerococcus          [1,151 x 2]
## 3 Aeromonas           [1,151 x 2]
```

# Mapping with a nested dataframe

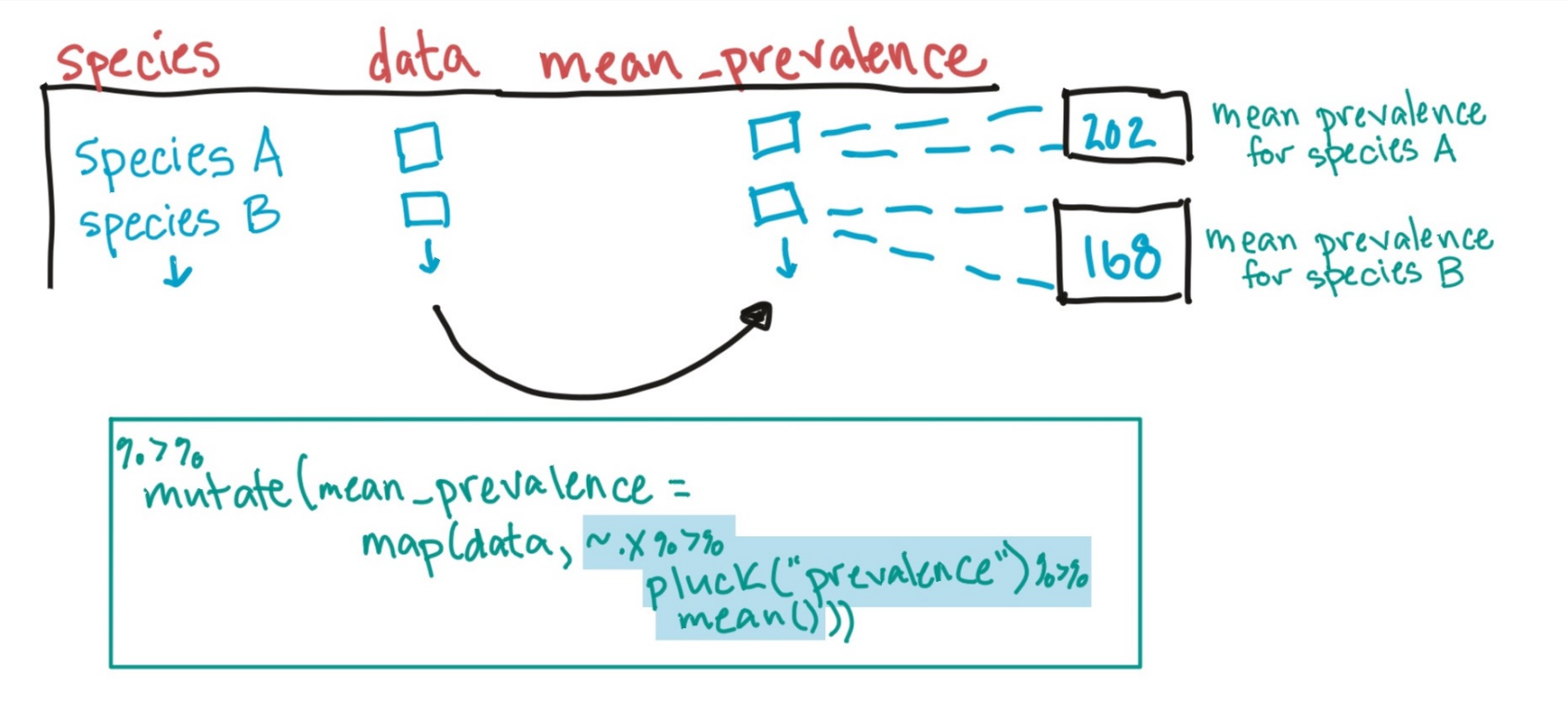
Now, you want to run the "recipe" you figured out on each bacteria species' dataframe.




The goal is to create a new list-column with the results for each species.

# Mapping with a nested dataframe

You can use the `map` function from the `purrr` package within a `mutate` function to run the "recipe" you figured out on each bacteria species' dataframe.



# Mapping with a nested dataframe

Here is what this looks like in code:

```
nested_samples2 <- nested_samples %>%
  mutate(mean_prevalence = map(data, ~ .x %>%
                                 pluck("prevalence") %>%
                                 mean()))


nested_samples2 %>%
  head(3)
```

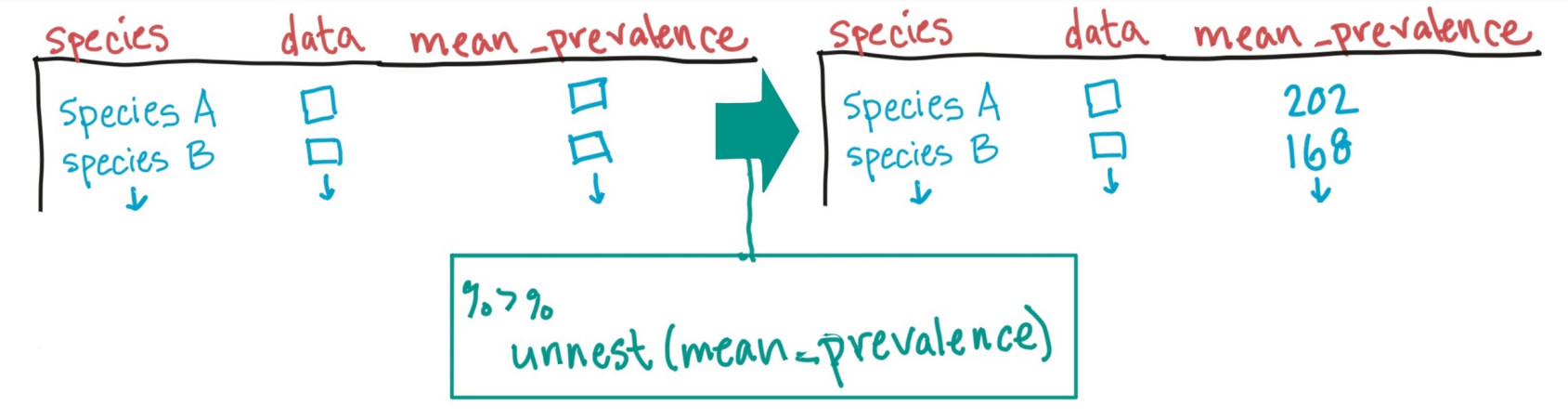
```
## # A tibble: 3 x 3
## # Groups:   species [3]
##   species                 data mean_prevalence
##   <chr>          <list<df[,2]>> <list>
## 1 Actinomycetaceae    [1,151 x 2] <dbl [1]>
## 2 Aerococcus          [1,151 x 2] <dbl [1]>
## 3 Aeromonas           [1,151 x 2] <dbl [1]>
```

# Unnesting a nested dataframe

You'll usually want to unnest the new column so you can use if for creating plots, tables, and other output. You can use `unnest` to convert this column back to a regular column or columns (e.g., a vector or vectors) in a dataframe.

# Unnesting a nested dataframe

You'll need to specify which list-column to "unnest" when you use `unnest`:



# Unnesting a nested dataframe

Here's an example in code:

```
unnested_samples <- nested_samples2 %>%
  unnest(mean_prevalence)

unnested_samples %>%
  head(3)
```

```
## # A tibble: 3 x 3
## # Groups:   species [3]
##   species                      data mean_prevalence
##   <chr>              <list<df[,2]>>           <dbl>
## 1 Actinomycetaceae    [1,151 x 2]          0.368
## 2 Aerococcus          [1,151 x 2]          0.00521
## 3 Aeromonas           [1,151 x 2]          0.00174
```

# Nesting and mapping

Here's what the whole process looks like, if you're doing it in a single piece of code:

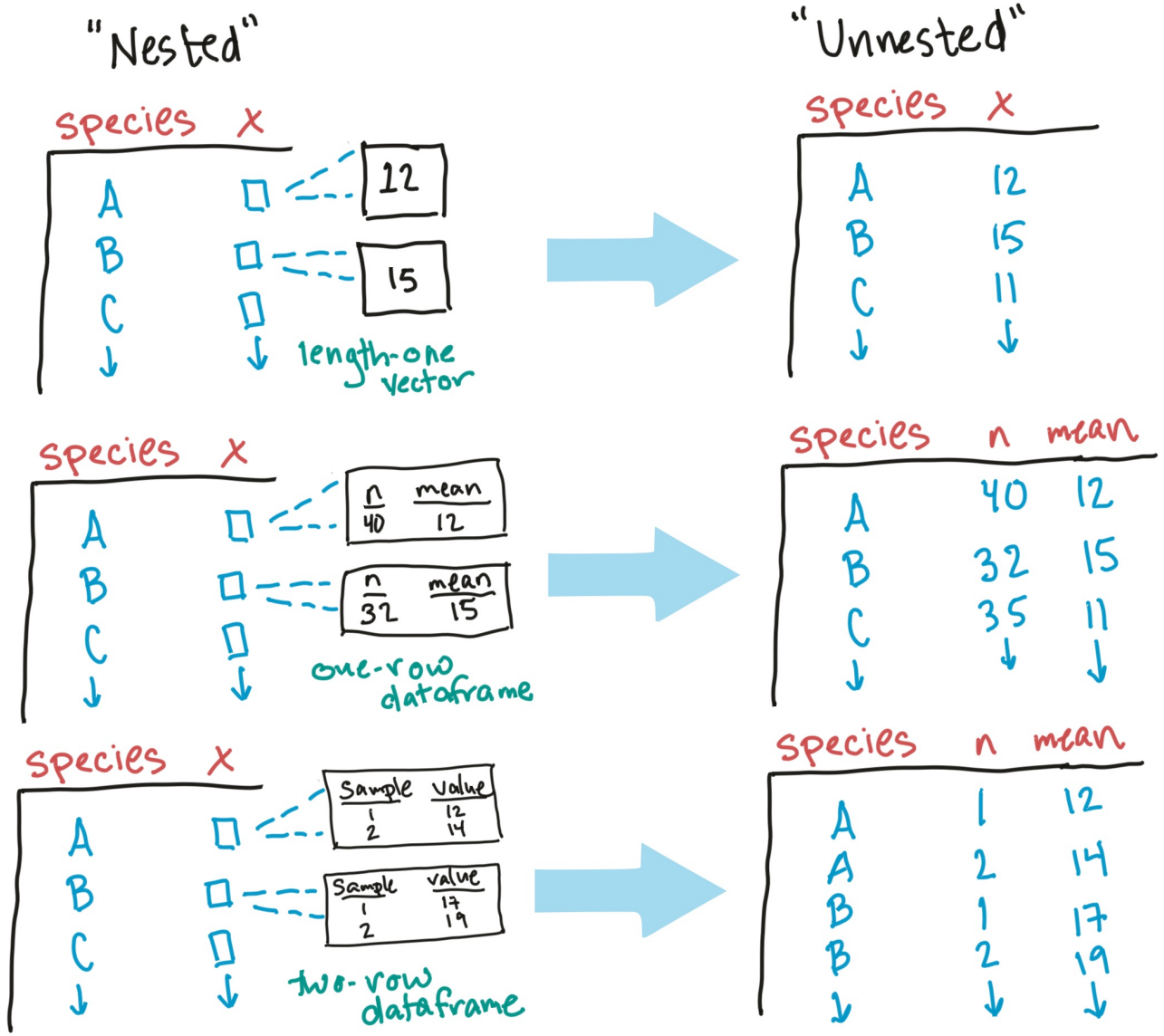
```
atlas1006 %>%
  get_sample() %>%
  as.data.frame() %>%
  tibble::rownames_to_column(var = "species") %>%
  tidyr::pivot_longer(- species,
                      names_to = "sample",
                      values_to = "prevalence") %>%
  group_by(species) %>%
  nest() %>%
  mutate(mean_prevalence = map(data, ~ .x %>%
                                 pluck("prevalence") %>%
                                 mean())) %>%
  unnest(mean_prevalence)
```

# Nesting and mapping

So far, this might not seem too exciting, since everything we just did could have been done more easily with `group_by` and `summarize`.

However, this approach allows you to do more complex things. You can expand because you can unnest much more complicated list-columns than ones with length-one vectors for each grouping value.

# Nesting and mapping



# Nesting and mapping

For example, you can use this approach to run Shapiro-Wilk tests for all bacteria's prevalence samples:

```
sample_norm_test <- atlas1006 %>%
  get_sample() %>%
  as.data.frame() %>%
  tibble::rownames_to_column(var = "species") %>%
  tidyr::pivot_longer(- species,
                      names_to = "sample",
                      values_to = "prevalence") %>%
  group_by(species) %>%
  nest() %>%
  mutate(norm_test = map(data, ~ .x %>%
                           pluck("prevalence") %>%
                           shapiro.test() %>%
                           tidy())) %>%
  unnest(norm_test)
```

# Nesting and mapping

```
sample_norm_test %>%
  head(4)
```

```
## # A tibble: 4 x 5
## # Groups:   species [4]
##   species                    data statistic  p.value method
##   <chr>           <list<df[,2]>>     <dbl>    <dbl> <chr>
## 1 Actinomycetace~    [1,151 x 2]     0.343  2.42e-53 Shapiro-W
## 2 Aerococcus         [1,151 x 2]     0.0362 6.49e-60 Shapiro-W
## 3 Aeromonas          [1,151 x 2]     0.0185 3.09e-60 Shapiro-W
## 4 Akkermansia        [1,151 x 2]     0.510  1.49e-48 Shapiro-W
```

# Nesting and mapping

```
sample_norm_test %>%
  ggplot(aes(x = log10(p.value))) +
  geom_histogram()
```