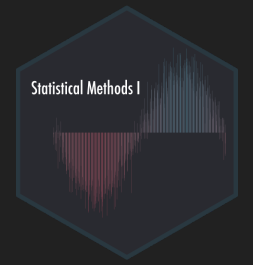
Pivoting and Measuring Confidence

Week 8

Packages needed and a Note about Icons



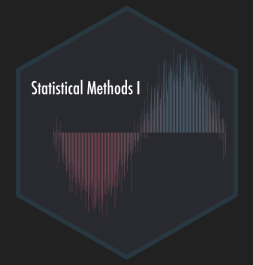
Please load up the following packages. Remember to first install the ones you don't have.

```
library(tidyverse)
library(mosaic)
library(ggplot2movies)
```

You may come across the following icons. The table below lists what each means.

Icon	Description
	Indicates that an example continues on the following slide.
	Indicates that a section using common syntax has ended.
	Indicates that there is an active hyperlink on the slide.
	Indicates that a section covering a concept has ended.

What is a confidence interval?

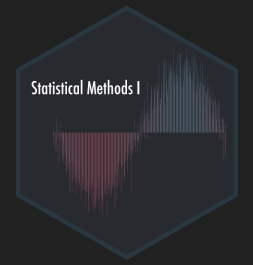


A *confidence interval* (CI) gives a range of possible values for a parameter. It depends on a specified *confidence level* with

- higher confidence levels corresponding to wider confidence intervals
- lower confidence levels corresponding to narrower confidence intervals.

The most common confidence levels include 90%, 95%, and 99%.

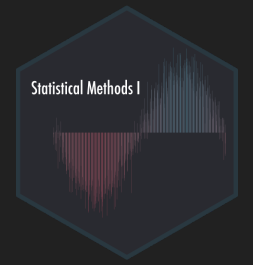
Problems with how confidence intervals are taught



You were just taught about the confidence interval in an bad way!

- Finding confidence intervals for some mean is to first assume a normal curve for a population and then magic
- But assuming normality is a BIG assumption!

Bootstrapping



Hypothesis testing

> - We simply want to if our H_0 or H_1 is correct.
> - First step in being able to generalize

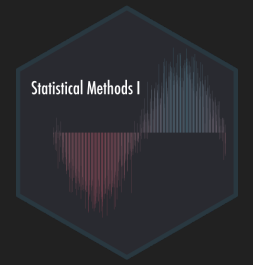
Typically we have a sample of a population's data so we can

1. take repeated samples from a sample data of size whatever
2. calculate the mean for each of these samples
3. created a new distribution of these means
4. estimate the population distribution

aka *bootstrapping*

5. calculate the confidence interval (CI)

ggplot2movies



We'll look at CIs, but first let's look at the `ggplot2movies` data set...

```
head(movies)
```

```
## # A tibble: 6 × 24
##   title       year length budget rating votes    r1    r2    r3
##   <chr>      <int>  <int>  <int>  <dbl> <int> <dbl> <dbl> <dbl>
## 1 $           1971    121     NA    6.4   348   4.5   4.5   4.5
## 2 $1000 a T…  1939     71     NA    6      20   0    14.5   4.5
## 3 $21 a Day…  1941      7     NA    8.2     5   0     0     0
## 4 $40,000     1996     70     NA    8.2     6  14.5   0     0
## 5 $50,000 C…  1975     71     NA    3.4    17  24.5   4.5   0
## 6 $pent       2000     91     NA    4.3    45   4.5   4.5   4.5
## # … with 15 more variables: r4 <dbl>, r5 <dbl>, r6 <dbl>,
## #   r7 <dbl>, r8 <dbl>, r9 <dbl>, r10 <dbl>, mpaa <chr>,
## #   Action <int>, Animation <int>, Comedy <int>, Drama <int>,
## #   Documentary <int>, Romance <int>, Short <int>
```

...its size...


```
dim(movies)
```

```
## [1] 58788    24
```

That's 58,788 rows by 24 columns!

... and the names of its columns.


```
names(movies)
```

```
##  [1] "title"       "year"        "length"      "budget"
##  [5] "rating"      "votes"       "r1"          "r2"
##  [9] "r3"          "r4"          "r5"          "r6"
## [13] "r7"          "r8"          "r9"          "r10"
## [17] "mpaa"        "Action"      "Animation"   "Comedy"
## [21] "Drama"       "Documentary" "Romance"     "Short"
```

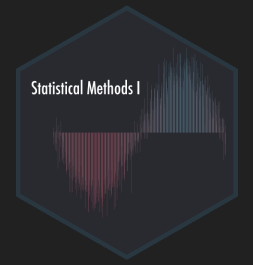
You can see more about the functionality by looking at its documentation. For now, here's what the variables mean:

- **title**. Title of the movie.
- **year**. Year of release.
- **budget**. Total budget (if known) in US dollars
- **length**. Length in minutes.
- **rating**. Average IMDB user rating.
- **votes**. Number of IMDB users who rated this movie.
- **r1-10**. Multiplying by ten gives percentile (to nearest 10%) of users who rated this movie a 1.
- **mpaa**. MPAA rating.
- **Action**, **Animation**, **Comedy**, **Drama**, **Documentary**, **Romance**, **Short**. Binary variables representing if movie was classified as belonging to that genre.

```r
select_movies <- movies %>%
  select(Action, Animation, Comedy, D
  pivot_longer(
    everything(),
    names_to = "genre"
  )

select_movies
```

```
## # A tibble: 411,516 × 2
##    genre       value
##    <chr>       <int>
##  1 Action          0
##  2 Animation       0
##  3 Comedy          1
##  4 Drama           1
##  5 Documentary     0
##  6 Romance         0
##  7 Short           0
##  8 Action          0
##  9 Animation       0
## 10 Comedy          1
## # … with 411,506 more rows
```

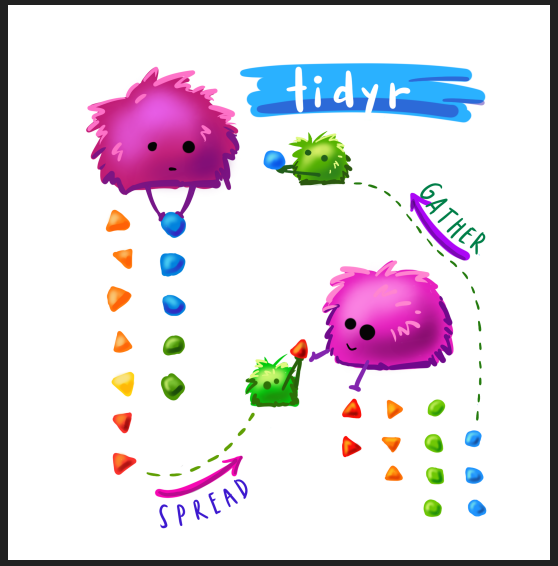


Pivot Wider

In instances where we have to go from a long to wide data set, we'd use a command called `pivot_wider`.

What Just Happened?

The previous iteration of `pivot_wider` and `pivot_longer` were given by the commands `gather` and `spread`. To see what these did, let's start out with a cartoon by the very talented Allison Horst out of UC Santa Barbara.



So the commands basically took data frames from wide to long with `gather` and back again with `spread`. The `pivot_` commands do much the same but with much more flexibility. However they can be equally as confusing so we'll go over a few basics here. If you're interested in more, take a look at this fantastic overview courtesy of R-Ladies Sydney. For an advanced walkthrough, the Data Wrangling site over at Stanford is a great resource.

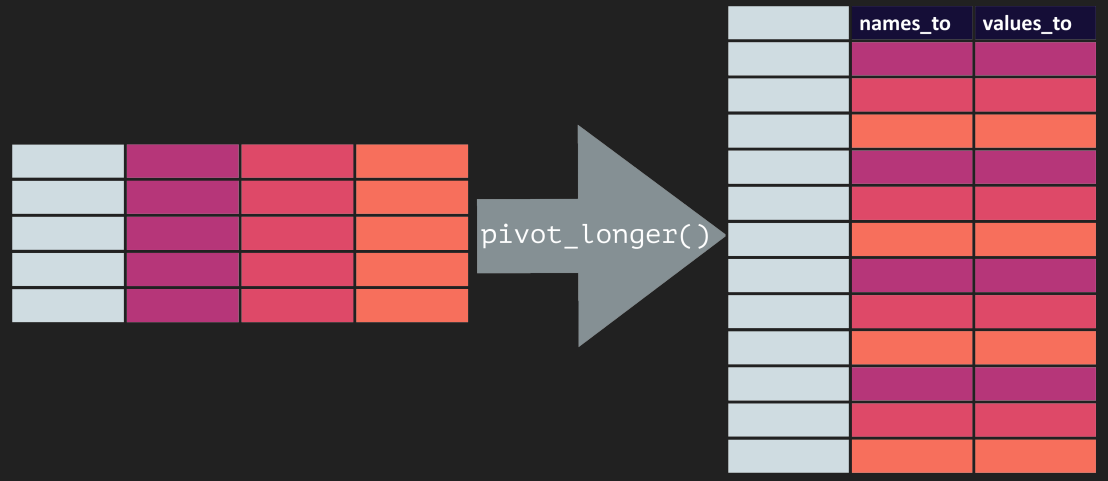
pivot_longer

It is pretty rare that at this stage in your academic development that you need to go from long to wide so we'll be concentrating on the converse with `pivot_longer`. First note that the original graphics in this part of the walkthrough were not created by me, but by RStudio's Allison Hill. I did however amend them for aesthetic purposes.

Ok let's begin!

An overview of pivot_longer

We'll concentrate one two options in `pivot_longer`: `names_to` and `values_to`.



Remember you can always run `?` in front of any command in the Console to get more information about it. For `pivot_longer`, we would simply type in

```
?pivot_longer
```

to see other options.

If you want to follow along with the fake data set we'll be using, run the following command to build the tibble


```
juniors_multiple <- tribble(
  ~ "baker", ~"cinnamon_1", ~"cardamom_2", ~"nutmeg_3",
  "Emma", 1L,   0L, 1L,
  "Harry", 1L,   1L, 1L,
  "Ruby", 1L,   0L, 1L,
  "Zainab", 0L, NA, 0L
)
```

and check it just to make sure

```
juniors_multiple
```

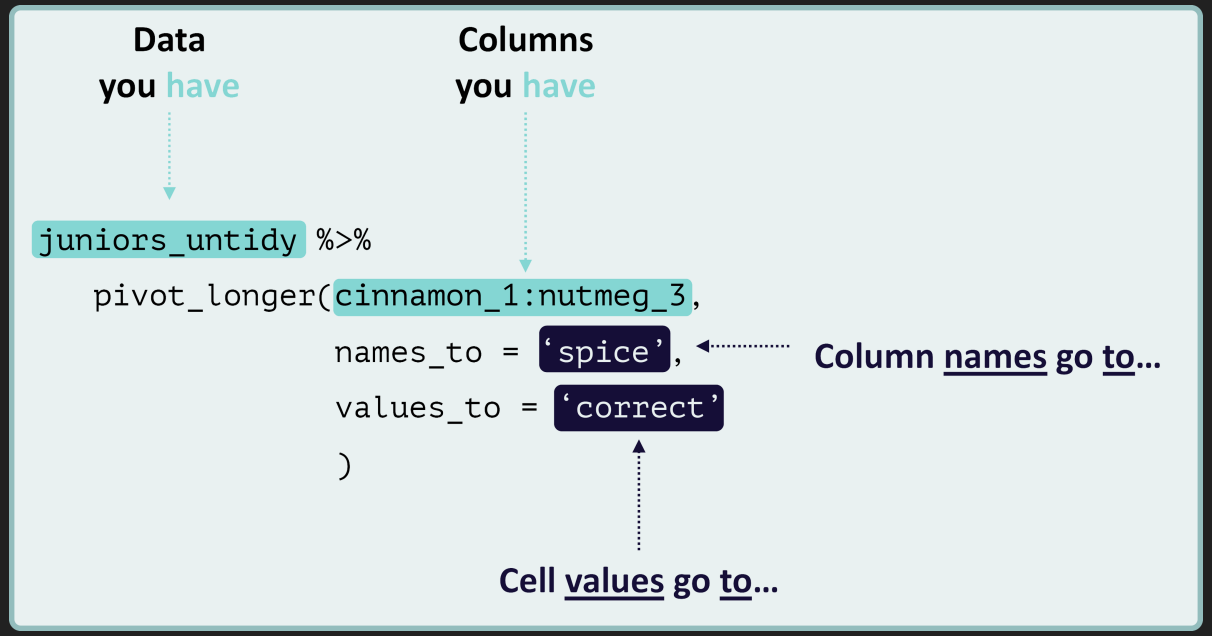
```
## # A tibble: 4 × 4
##   baker  cinnamon_1 cardamom_2 nutmeg_3
##   <chr>       <int>      <int>    <int>
## 1 Emma            1          0        1
## 2 Harry           1          1        1
## 3 Ruby            1          0        1
## 4 Zainab          0         NA        0
```

Looks good! Let's convert this!

To remind you of what the `juniors_multiple` data frame looks like, we have

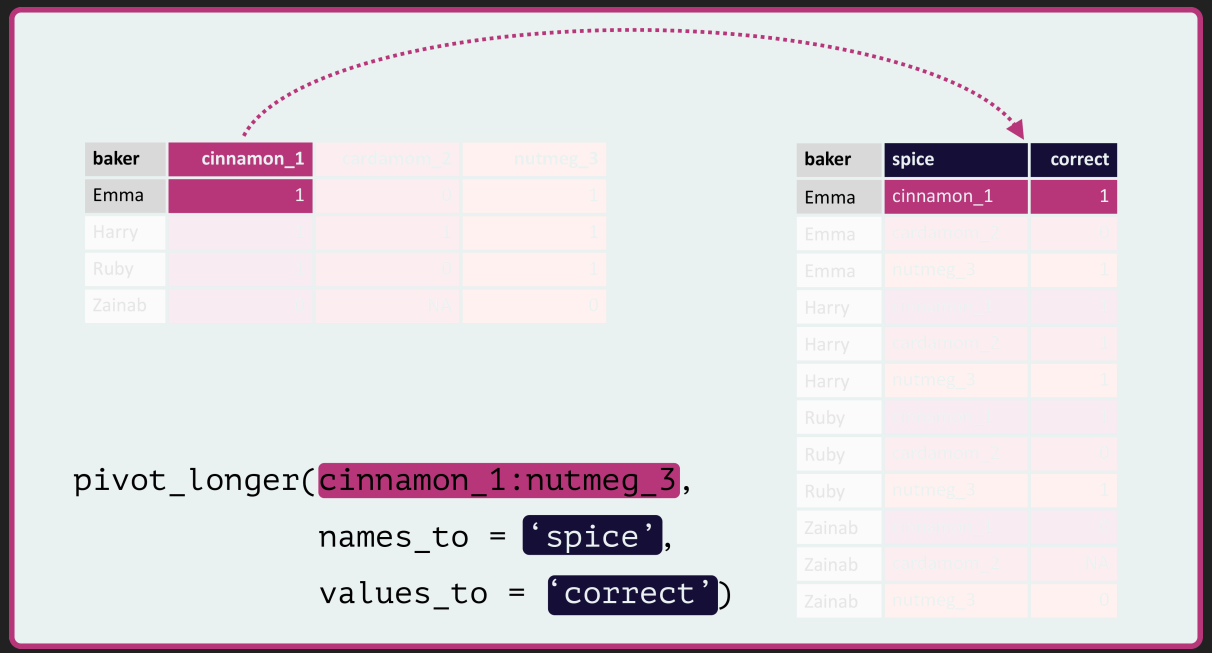
baker	cinnamon_1	cardamom_2	nutmeg_3
Emma	1	0	1
Harry	1	1	1
Ruby	1	0	1
Zainab	0	NA	0

We can assign names to the eventual columns using `names_to` and `values_to`.



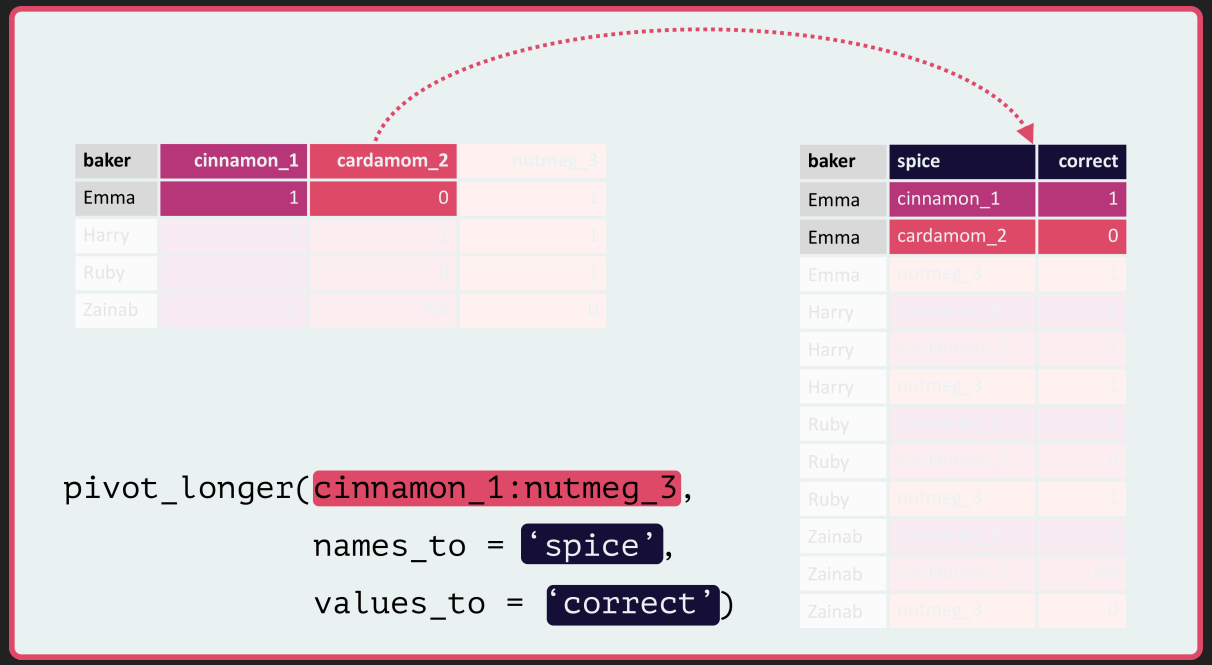
baker	cinnamon_1	cardamom_2	nutmeg_3
Emma	1	0	1
Harry	1	1	1
Ruby	1	0	1
Zainab	0	NA	0

We can assign names to the eventual columns using `names_to` and `values_to`.

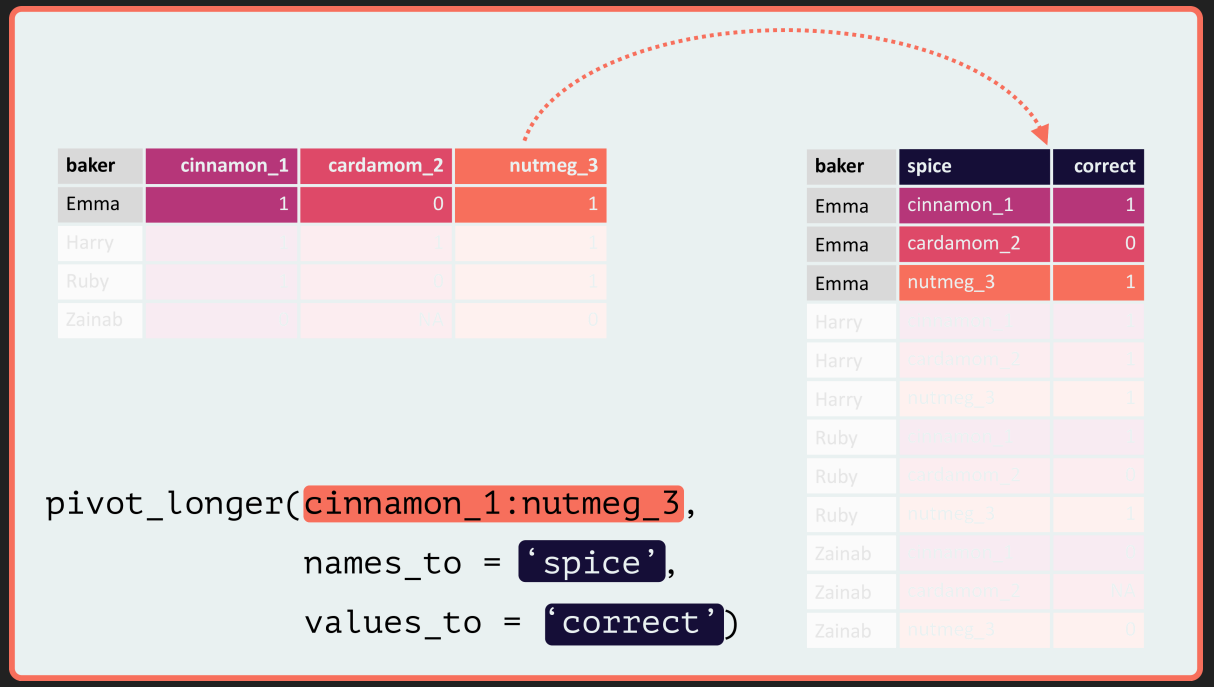


baker	cinnamon_1	cardamom_2	nutmeg_3
Emma	1	0	1
Harry	1	1	1
Ruby	1	0	1
Zainab	0	NA	0

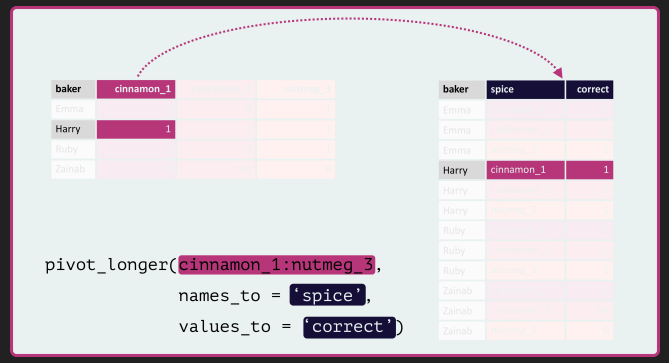
Here you can see the first column `cinnamon_1` and its value `1` associated with the first row `Emma` becomes our first two values under the two columns `spice` and `correct` for our pivoted data frame.

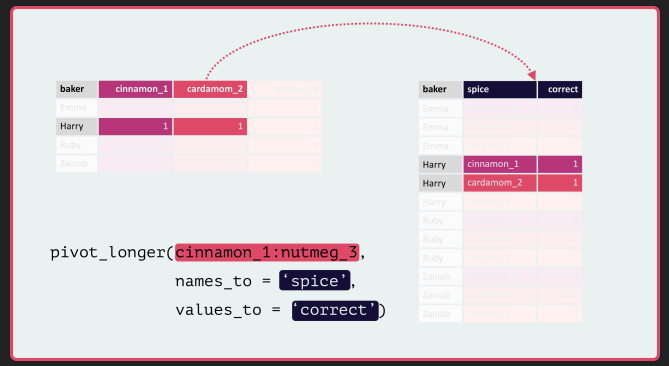


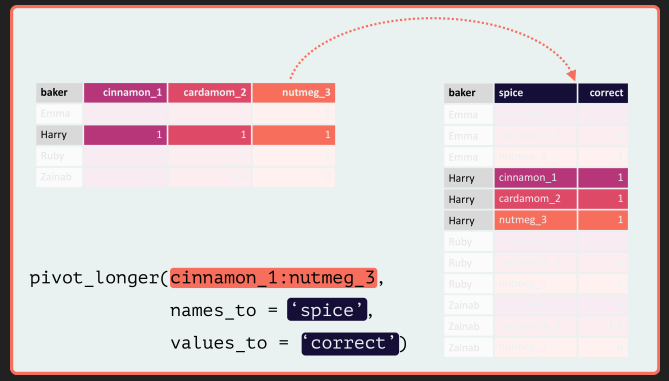
This pattern continues until a whole row is used up.



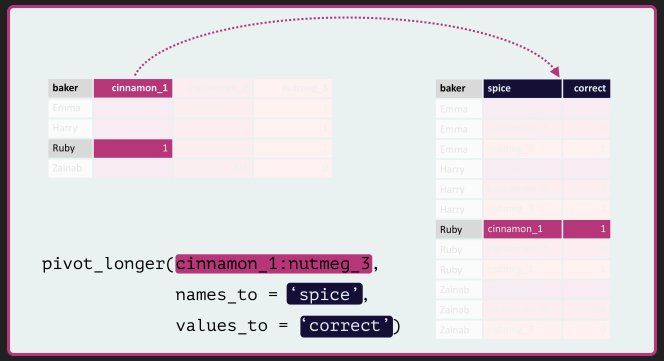
Then it repeats for the next row of values...

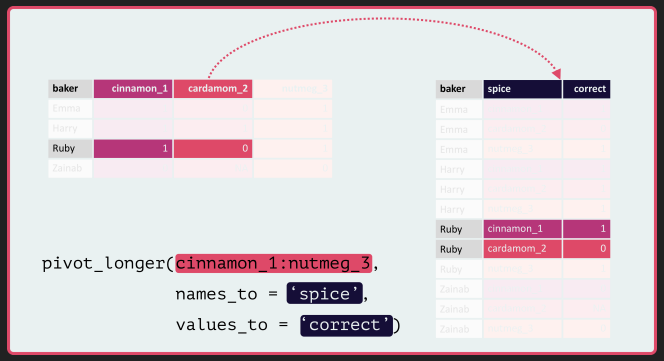


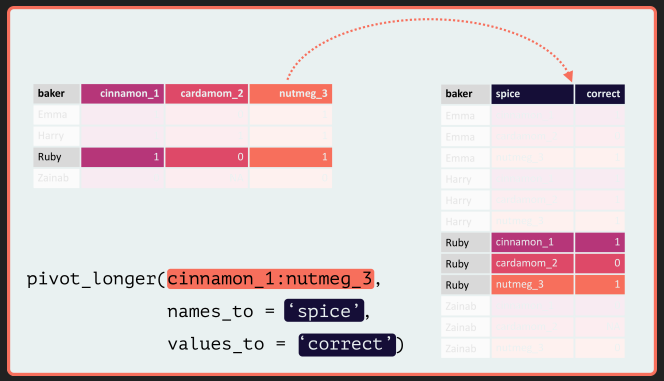




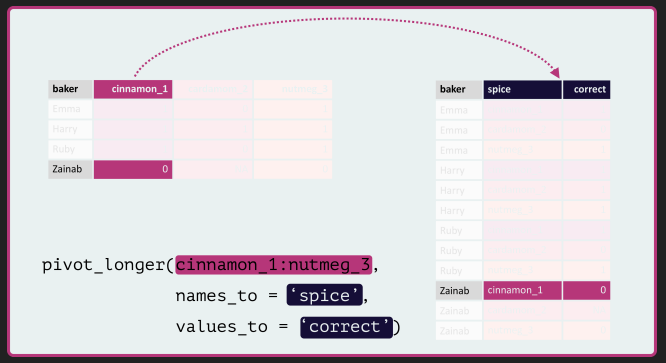
...and so forth...

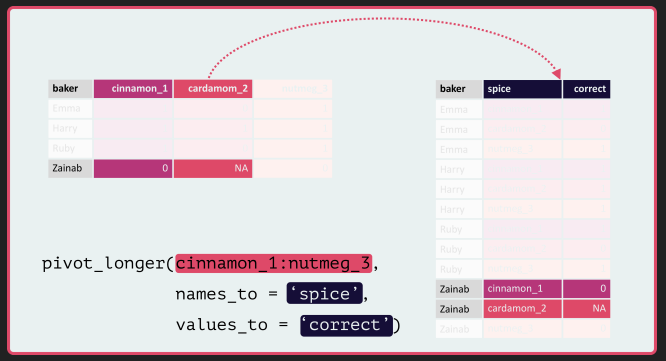

baker cinnamon_1
Ruby 1
baker spice correct
Ruby cinnamon_1 1
pivot_longer(cinnamon_1:nutmeg_3,
names_to = 'spice',
values_to = 'correct')

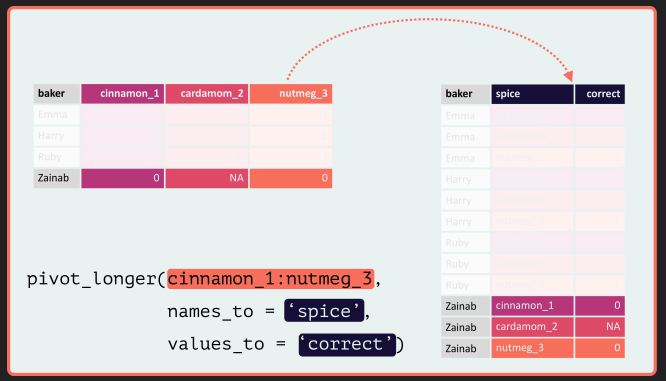

baker cinnamon_1 cardamom_2
Ruby 1 0
baker spice correct
Ruby cinnamon_1 1
Ruby cardamom_2 0
pivot_longer(cinnamon_1:nutmeg_3,
names_to = 'spice',
values_to = 'correct')


baker cinnamon_1 cardamom_2 nutmeg_3
Ruby 1 0 1
baker spice correct
Ruby cinnamon_1 1
Ruby cardamom_2 0
Ruby nutmeg_3 1
pivot_longer(cinnamon_1:nutmeg_3,
names_to = 'spice',
values_to = 'correct')

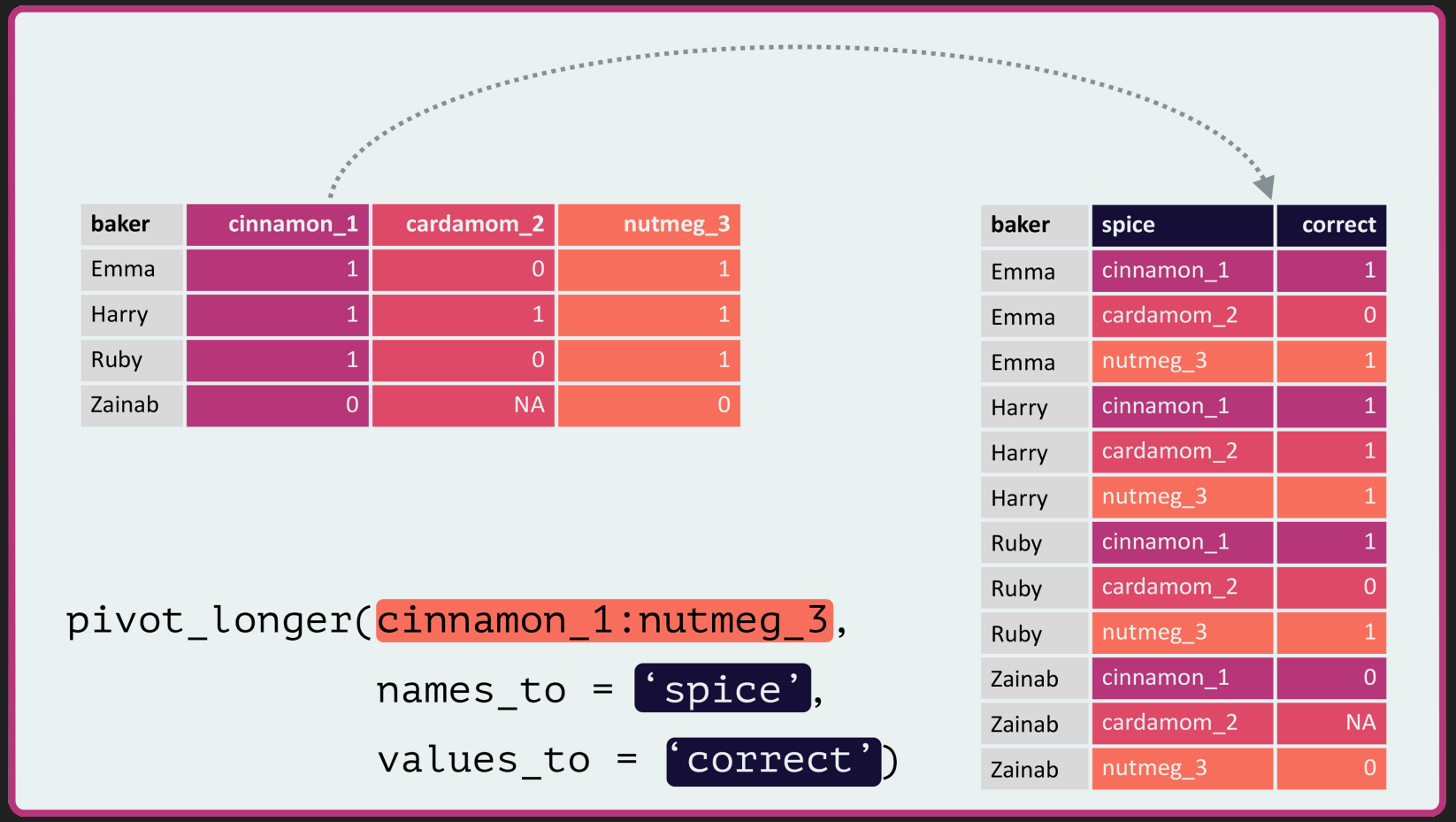
...until we run out of rows...







...and get the final table of pivoted values.

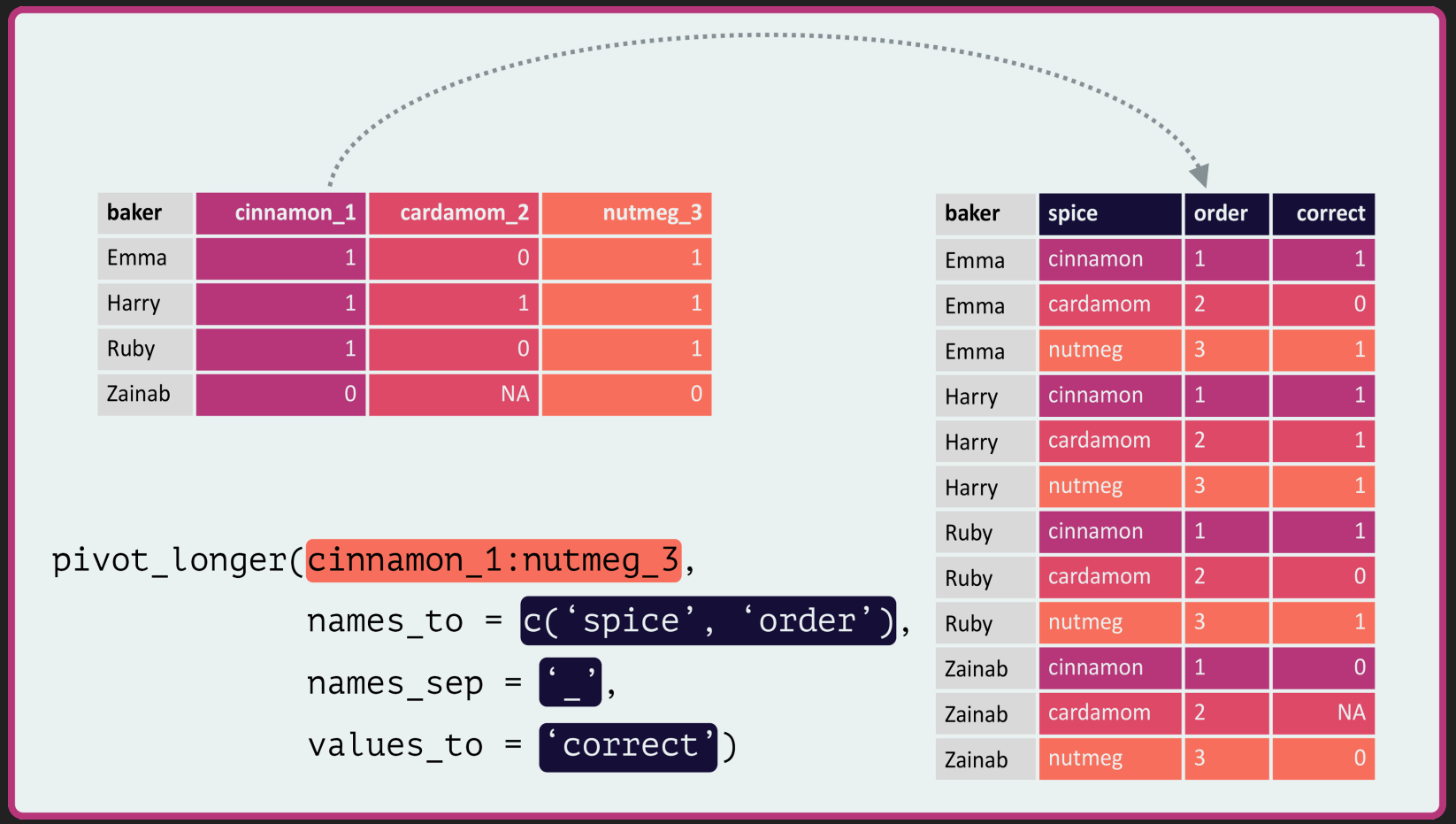


baker	cinnamon_1	cardamom_2	nutmeg_3
Emma	1	0	1
Harry	1	1	1
Ruby	1	0	1
Zainab	0	NA	0

```
pivot_longer(cinnamon_1:nutmeg_3,
             names_to = 'spice',
             values_to = 'correct')
```

baker	spice	correct
Emma	cinnamon_1	1
Emma	cardamom_2	0
Emma	nutmeg_3	1
Harry	cinnamon_1	1
Harry	cardamom_2	1
Harry	nutmeg_3	1
Ruby	cinnamon_1	1
Ruby	cardamom_2	0
Ruby	nutmeg_3	1
Zainab	cinnamon_1	0
Zainab	cardamom_2	NA
Zainab	nutmeg_3	0

We can even amend the current command to include things like `order`!



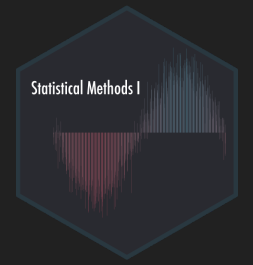
baker	cinnamon_1	cardamom_2	nutmeg_3
Emma	1	0	1
Harry	1	1	1
Ruby	1	0	1
Zainab	0	NA	0

```
pivot_longer(cinnamon_1:nutmeg_3,
             names_to = c('spice', 'order'),
             names_sep = '_',
             values_to = 'correct')
```

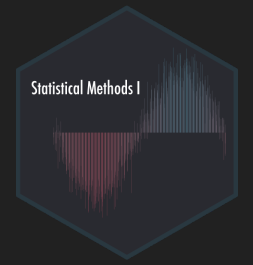
baker	spice	order	correct
Emma	cinnamon	1	1
Emma	cardamom	2	0
Emma	nutmeg	3	1
Harry	cinnamon	1	1
Harry	cardamom	2	1
Harry	nutmeg	3	1
Ruby	cinnamon	1	1
Ruby	cardamom	2	0
Ruby	nutmeg	3	1
Zainab	cinnamon	1	0
Zainab	cardamom	2	NA
Zainab	nutmeg	3	0

```r
juniors_multiple %>%
  pivot_longer(-baker,
               names_to = c('spice',
               names_sep = '_',
               values_to = 'correct'
```

```
## # A tibble: 12 × 4
##    baker  spice    order correct
##    <chr>  <chr>    <chr>   <int>
##  1 Emma   cinnamon 1           1
##  2 Emma   cardamom 2           0
##  3 Emma   nutmeg   3           1
##  4 Harry  cinnamon 1           1
##  5 Harry  cardamom 2           1
##  6 Harry  nutmeg   3           1
##  7 Ruby   cinnamon 1           1
##  8 Ruby   cardamom 2           0
##  9 Ruby   nutmeg   3           1
## 10 Zainab cinnamon 1           0
## 11 Zainab cardamom 2          NA
## 12 Zainab nutmeg   3           0
```



Single column types



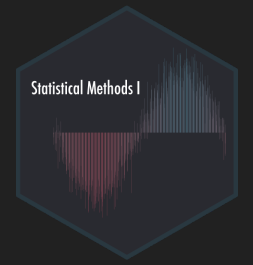
`pivot_wider` is great for columns of the same type. For example, if we run

```
glimpse(juniors_multiple)
```

```
## Rows: 4
## Columns: 4
## $ baker      <chr> "Emma", "Harry", "Ruby", "Zainab"
## $ cinnamon_1 <int> 1, 1, 1, 0
## $ cardamom_2 <int> 0, 1, 0, NA
## $ nutmeg_3   <int> 1, 1, 1, 0
```

all we have are integers...

Multiple column types



... but for the following

```
juniors_multiple_full <- tribble(
  ~ "baker", ~"score_1", ~"score_2", ~"score_3",
  ~ "guess_1", ~"guess_2", ~"guess_3",
    "Emma", 1L,   0L, 1L, "cinnamon", "cloves", "nutmeg",
    "Harry", 1L,   1L, 1L, "cinnamon", "cardamom", "nutmeg",
    "Ruby", 1L,   0L, 1L, "cinnamon", "cumin", "nutmeg",
    "Zainab", 0L, NA, 0L, "cardamom", NA_character_, "cinnamon"
  )
```

```
juniors_multiple_full
```

```
## # A tibble: 4 × 7
##   baker  score_1 score_2 score_3 guess_1  guess_2  guess_3
##   <chr>    <int>   <int>   <int> <chr>    <chr>    <chr>
## 1 Emma         1       0       1 cinnamon cloves   nutmeg
## 2 Harry        1       1       1 cinnamon cardamom nutmeg
## 3 Ruby         1       0       1 cinnamon cumin    nutmeg
## 4 Zainab       0      NA       0 cardamom <NA>     cinnamon
```

```
glimpse(juniors_multiple_full)
```


```
## Rows: 4
## Columns: 7
## $ baker   <chr> "Emma", "Harry", "Ruby", "Zainab"
## $ score_1 <int> 1, 1, 1, 0
## $ score_2 <int> 0, 1, 0, NA
## $ score_3 <int> 1, 1, 1, 0
## $ guess_1 <chr> "cinnamon", "cinnamon", "cinnamon", "cardamom"
## $ guess_2 <chr> "cloves", "cardamom", "cumin", NA
## $ guess_3 <chr> "nutmeg", "nutmeg", "nutmeg", "cinnamon"
```

...we have both character and numeric vectors.

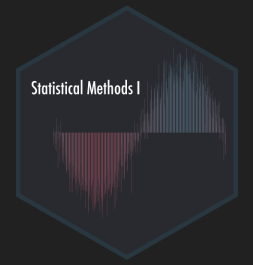
Try running the following


```
juniors_multiple_full %>%
  pivot_longer(score_1:guess_3,
               names_to = c('score', 'guess'),
               names_sep = "_",
               values_to = 'correct')
```

Do you get `Error: Can't combine score_1 <integer> and guess_1 <character>.`? So what can you do?

Well since computers are stupid, you have to tell R what to look for.

Generalizing



We can actually just tell R to treat all values the same.

```
juniors_multiple_full %>%
  # Don't do anything with the baker column
    pivot_longer(-baker,
                 # Treat all columns the same and order them
                 names_to = c(".value", "order"),
                 # Control how the column names are broken up
                 names_sep = "_")
```

```
## # A tibble: 12 × 4
##    baker  order score guess
##    <chr>  <chr> <int> <chr>
##  1 Emma   1         1 cinnamon
##  2 Emma   2         0 cloves
##  3 Emma   3         1 nutmeg
##  4 Harry  1         1 cinnamon
##  5 Harry  2         1 cardamom
##  6 Harry  3         1 nutmeg
##  7 Ruby   1         1 cinnamon
##  8 Ruby   2         0 cumin
##  9 Ruby   3         1 nutmeg
## 10 Zainab 1         0 cardamom
## 11 Zainab 2        NA <NA>
## 12 Zainab 3         0 cinnamon
```

Thats it!