Advanced Tabular Data Manipulation

Alejandro Schuler, adapted from Steve Bagley and based on R for Data Science by Hadley Wickham
July 2021

- compute cumulative, offset, and sliding-window transformations
- simultaneously transform or summarize multiple columns
- transform between long and wide data formats
- combine multiple data frames using joins on one or more columns

Window transformations

Offsets

- `lead()` and `lag()` return either forward- or backward-shifted versions of their input vectors

```
lead(c(1,2,3))
[1]  2  3 NA
lag(c(1,2,3))
[1] NA  1  2
```

- This is most useful to compute offsets, which we often do when looking at time series.

Offsets Example

- Let's look at some time series data about when GTEx samples were collected, and calculate the change over time

```
gtex_samples_time_link = "https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/af6056f9a8d999a80fd787f89aa3483157d43681/data/gtex_metadata/gtex_samples_time.csv"

gtex_samples_by_month = read_csv(file = gtex_samples_time_link, col_types =
cols())

gtex_samples_by_month %>%
  head(2L)
# A tibble: 2 x 3
  month  year num_samples
  <dbl> <dbl>       <dbl>
1     5  2011          20
2     6  2011          44
```

```
gtex_samples_by_month %>%
  mutate(increase_in_samples = total_num_samples - lag(total_num_samples)) %>%
  head(3L)
Error: Problem with `mutate()` input `increase_in_samples`.
x object 'total_num_samples' not found
i Input `increase_in_samples` is `total_num_samples - lag(total_num_samples)`.
```

Exercise: multiple offsets

```
gtex_samples_by_month %>%
  filter(month %in% c(3, 6, 9, 12)) %>%
  head(n = 6L)
# A tibble: 6 x 3
  month  year num_samples
  <dbl> <dbl>       <dbl>
1     6  2011          44
2     9  2011         100
3    12  2011         111
4     3  2012         141
5     6  2012          63
6     9  2012         115
```

Let's say you want to compute the change in number of samples within months from one year to the next. We've shortened the data to only include four months each year (March, June, September, and December) to simplify this. (i.e. comparing March 2015 to March 2016 and June 2015 to June 2016)

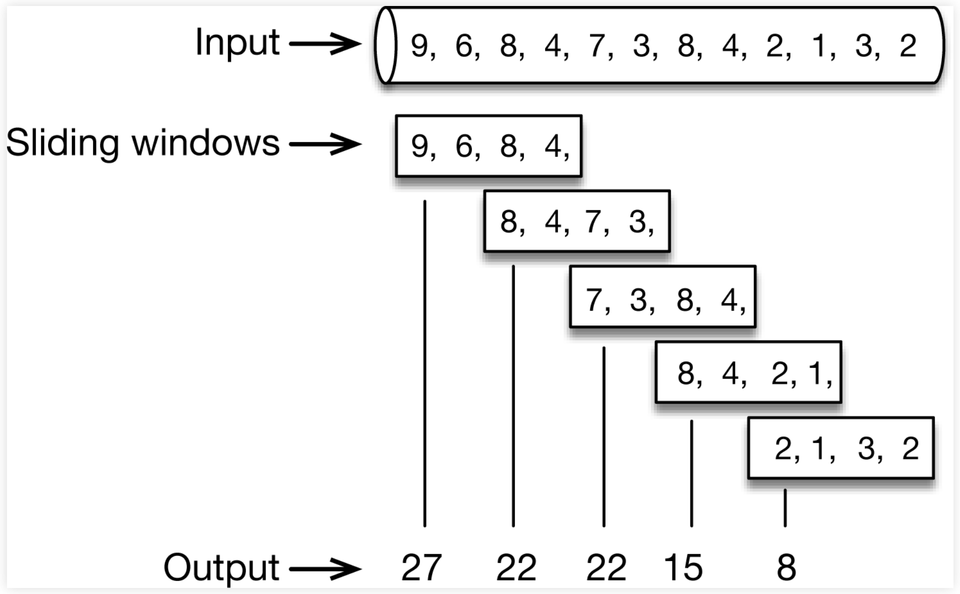
1. Figure out a way to do this using `lead()` or `lag()` in a single `mutate()` statement (hint: check the documentation).
2. Figure out a different way to do this with `group_by()` instead. Which seems more natural or robust to you? Why?
3. In both solutions you end up with some `NA`s since the March 2011 counts are unknown. If we wanted to assume that the March 2011 would be the same as the June 2011, how might we modify our code to reflect that in order to make sure we don't get `NA`s in the result?

Rolling functions

- `slide_vec` applies a function using a sliding window across a vector (sometimes called a "rolling" function)

```
library("slider")
numbers = c(9, 6, 8, 4, 7, 3, 8, 4, 2,
1, 3, 2)
slide_vec(numbers, sum, .after = 3,
.step = 2L)
 [1] 27 NA 22 NA 22 NA 15 NA  8 NA  5
NA
```

- the `.after` argument specifies how many elements after the "index" element are included in the rolling window
- `.step` specifies how to move from one index element to the next
- `.before` is the backward-looking equivalent of `.after`



```
gtex_samples_by_month %>%
  mutate(avg_samples_2_month =
slide_vec(total_num_samples, mean,
.before = 1L)) %>%
  select(-year) %>%
  head(2L)
Error: Problem with `mutate()` input
`avg_samples_2_month`.
x object 'total_num_samples' not found
i Input `avg_samples_2_month` is
`slide_vec(total_num_samples, mean,
.before = 1L)`.
```

Cumulative functions

- A cumulative function is like a rolling window function except that the window expands with each iteration instead of shifting over
- For example, `cumsum` takes the cumulative sum of a vector. See `?cumsum` for similar functions

```
cumsum(c(1, 2, 3))
[1] 1 3 6
```

```
gtex_samples_by_month %>%
  mutate(num_samples_to_date = cumsum(total_num_samples))
Error: Problem with `mutate()` input `num_samples_to_date`.
x object 'total_num_samples' not found
i Input `num_samples_to_date` is `cumsum(total_num_samples)`.
```

Turning any function into a cumulative function

- you can use `slider::slide_vec()` to turn any function that accepts a vector and returns a number into a cumulative function
- Use `.before=Inf` to achieve this

```
library(slider) # imports slide_vec() function

gtex_samples_by_month %>%
  mutate(samples_to_date = slide_vec(total_num_samples, sum, .before = Inf))
Error: Problem with `mutate()` input `samples_to_date`.
x object 'total_num_samples' not found
i Input `samples_to_date` is `slide_vec(total_num_samples, sum, .before = Inf)`.
```

- it is usually better (computationally faster) to use a built-in cumulative function (e.g. `cumsum()`), but if none exists this is a great solution

Turning any function into a cumulative function

- If the function you want to transform takes additional arguments, you can give those to `slide_vec` and it will pass them through for you

```
gtex_samples_by_month %>%
  mutate(
    avg_samples_by_month = slide_vec(total_num_samples, mean, .before = Inf, na.rm
= TRUE)
  )
Error: Problem with `mutate()` input `avg_samples_by_month`.
x object 'total_num_samples' not found
i Input `avg_samples_by_month` is `slide_vec(total_num_samples, mean, .before =
Inf, na.rm = TRUE)`.
```

Exercise: total number of samples in the last twelve months

```r
library(lubridate) # this helps us with dates
```

```
gtex_samples_by_month
# A tibble: 66 x 3
   month  year num_samples
   <dbl> <dbl>       <dbl>
 1     5  2011          20
 2     6  2011          44
 3     7  2011          90
 4     8  2011         132
 5     9  2011         100
 6    10  2011         110
 7    11  2011         203
 8    12  2011         111
 9     1  2012         208
10     2  2012          95
# … with 56 more rows
```

Starting with this `gtex_samples_by_month` dataframe you, add a column that has the total number of samples in the last twelve months relative to the row we are on.

Column-wise operations

Repeating operations on columns

```
df = tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  g = rbinom(10, 1, 0.5)
)
```

- Let's say we have these data and we want to take the mean of each column `a`, `b`, and `c` within the groups `g`.
- One way to do it is with a normal summarize:

```
df %>%
  group_by(g) %>%
  summarize(
    mean_a = mean(a),
    mean_b = mean(b),
    mean_b = mean(c)
  )
# A tibble: 2 x 3
      g mean_a  mean_b
  <int>  <dbl>   <dbl>
1     0 -0.249 -0.0148
2     1  1.06  -0.569
```

- Copy-pasting code like this frequently creates errors and bugs that are hard to see
- It's even worse if you want to do multiple summaries

```
df %>%
  group_by(g) %>%
  summarize(
    mean_a = mean(a),
    mean_b = mean(b),
    mean_b = mean(c),
    median_a = median(a),
    median_b = median(b),
    median_c = median(c)
  )
# A tibble: 2 x 6
      g mean_a  mean_b median_a
median_b median_c
  <int>  <dbl>   <dbl>    <dbl>
<dbl>    <dbl>
1     0 -0.249 -0.0148   -0.686
-0.218  -0.0994
2     1  1.06  -0.569     1.15
0.570  -0.824
```

Columnwise operations

- The solution is to use `across()` in your summarize:

```
df %>%
  group_by(g) %>%
  summarize(across(.cols = c(a,b,c),
.fns = mean))
# A tibble: 2 x 4
      g      a      b       c
  <int>  <dbl>  <dbl>   <dbl>
1     0 -0.249 -0.236 -0.0148
2     1  1.06   0.788 -0.569
```

- The **first argument** to `across()` is a selection of columns. You can use anything that would work in a `select()` here
- We've explicitly included the argument names for `.cols` and `.fns` here, but in R code out-in-the-wild they're usually omitted.

- The **second argument** is the function you'd like to apply to each column. You can provide multiple functions by wrapping them in a "`list()`". Lists are like vectors but their elements can be of different types and each element has a name (more on that later)

```
fns = list(
  avg = mean,
  max = max
)

df %>%
  group_by(g) %>%
  summarize(across(c(a,b), fns))
# A tibble: 2 x 5
      g  a_avg a_max  b_avg b_max
  <int>  <dbl> <dbl>  <dbl> <dbl>
1     0 -0.249  2.05 -0.236  1.79
2     1  1.06   1.93  0.788  2.61
```

- see `?across()` to find out how to control how these columns get named in the output

Columnwise operations with where()

```r
df = tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = as.character(rnorm(10)),
  g = rbinom(10, 1, 0.5)
)
```

- Sometimes its nice to apply a transformation to all columns of a given type or all columns that match some condition
- `where()` is a handy function for that

```r
df %>%
  group_by(g) %>%
  summarize(across(where(is.numeric), mean))
# A tibble: 2 x 3
      g      a      b
  <int>  <dbl>  <dbl>
1     0 0.0873  0.113
2     1 0.185  -1.20
```

Columnwise mutate

```
df = tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = as.character(rnorm(10)),
  g = rbinom(10, 1, 0.5)
)
```

- `across()` works with any `dplyr` "verb", including `mutate()`:

```
df %>%
  mutate(across(where(is.character), as.numeric))
# A tibble: 10 x 4
        a      b       c     g
    <dbl>  <dbl>   <dbl> <int>
 1 -2.22   1.43   1.49       0
 2  1.52   0.757  1.51       0
 3  0.935  0.972  0.606      1
 4  0.710  1.34   0.205      1
 5 -1.46   0.327 -1.75       1
 6  0.385  0.764  0.635      0
 7 -0.444 -1.38   2.47       1
 8  1.82   0.609  0.0912     1
 9  0.228 -0.675 -0.214      1
10 -0.139  0.978  0.383      1
```

Columnwise mutate

- Most often you will need to write your own mini function to do what you want. To do that you put ~ before your expression and use `.` where you would put the name of the column

```
df %>%
  mutate(across(
    a:b,                         # columns to mutate
    ~ . - lag(.),              # function to mutate them with
    .names = '{col}_offset'    # how to name the outputs
  ))
# A tibble: 10 x 6
        a      b c                                g a_offset b_offset
    <dbl>  <dbl> <chr>                        <int>    <dbl>    <dbl>
 1 -2.22   1.43  1.48582091419658                 0   NA       NA    
 2  1.52   0.757 1.5111112894678                  0    3.74    -0.677
 3  0.935  0.972 0.605762865798173                1   -0.586    0.215
 4  0.710  1.34  0.204771748793352                1   -0.225    0.369
 5 -1.46   0.327 -1.74629427588112                1   -2.17    -1.01 
 6  0.385  0.764 0.634975230582194                0    1.85     0.437
 7 -0.444 -1.38  2.46701997718975                 1   -0.829   -2.14 
 8  1.82   0.609 0.0912452808035654               1    2.27     1.98 
 9  0.228 -0.675 -0.214257824130902               1   -1.59    -1.28 
10 -0.139  0.978 0.382695567242115                1   -0.367    1.65 
```

- Note that I've also used the `.names` argument to control how the output columns get named

Exercise: Filtering out or replacing NAs

Let's go back to the GTEx expression data we've been looking at:

```
gtex_link =
  'https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/9e4fb21ccf93a83e2b6004b9aa467426806f8589/data/gtex.tissue.zscores.advance2020.txt'


gtex_data = read_tsv(file = gtex_link, col_types = cols())

head(gtex_data, 4L)
# A tibble: 4 x 7
  Gene  Ind        Blood Heart  Lung Liver NTissues
  <chr> <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl>
1 A2ML1 GTEX-11DXZ -0.14 -1.08 NA    -0.66        3
2 A2ML1 GTEX-11GSP -0.5   0.53  0.76 -0.1         4
3 A2ML1 GTEX-11NUK -0.08 -0.4  -0.26 -0.13        4
4 A2ML1 GTEX-11NV4 -0.37  0.11 -0.42 -0.61        4
```

1. Replacing `NA`s with some other value is a very common operation so it gets its own function: `replace_na()`. Use this function to replace all `NA`s present in any numeric column with `0`s
2. Instead of replacing these values, we may want to filter them all out instead. Starting with the original data, use `filter()` and `across()` to remove all rows from the data that have any `NA`s in any column. Recall that `is.na()` checks which elements in a vector are `NA`.

Tidy data: rearranging a data frame

Messy data

- Sometimes data are organized in a way that makes it difficult to compute in a vector-oriented way. For example, look at this dataset:

```
gtex_time_link =
  "https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-
2021/data/gtex_metadata/gtex_time_tissue.csv"

gtex_time_tissue_data = read_csv(file = gtex_time_link, col_types = cols())

head(gtex_time_tissue_data, 3L)
# A tibble: 3 x 8
  tissue          `2011` `2012` `2013` `2014` `2015` `2016` `2017`
  <chr>            <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Adipose Tissue      56    107    243    206     84    134      2
2 Adrenal Gland       28     41     84     65     20     31      0
3 Bladder              2     18      0      1      0      0      0
```

- the values in the table represent how many samples of that tissue were collected during that year.
- How could I use ggplot to make this plot? It's hard!

Messy data

```
head(gtex_time_tissue_data, 3L)
# A tibble: 3 x 8
  tissue          `2011` `2012` `2013` `2014` `2015` `2016` `2017`
  <chr>            <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Adipose Tissue      56    107    243    206     84    134      2
2 Adrenal Gland       28     41     84     65     20     31      0
3 Bladder              2     18      0      1      0      0      0
```

- One of the problems with the way these data are formatted is that the year collected, which is a property of the samples, is stuck into the names of the columns.
- Because of this, it's also not obvious what the numbers in the table mean (although we know they are counts)

Tidy data

- Here's a better way to organize the data:

```
# A tibble: 6 x 3
  tissue         year  count
  <chr>          <chr> <dbl>
1 Adipose Tissue 2011     56
2 Adipose Tissue 2012    107
3 Adipose Tissue 2013    243
4 Adipose Tissue 2014    206
5 Adipose Tissue 2015     84
6 Adipose Tissue 2016    134
```

This data is *tidy*. Tidy data follows three precepts:

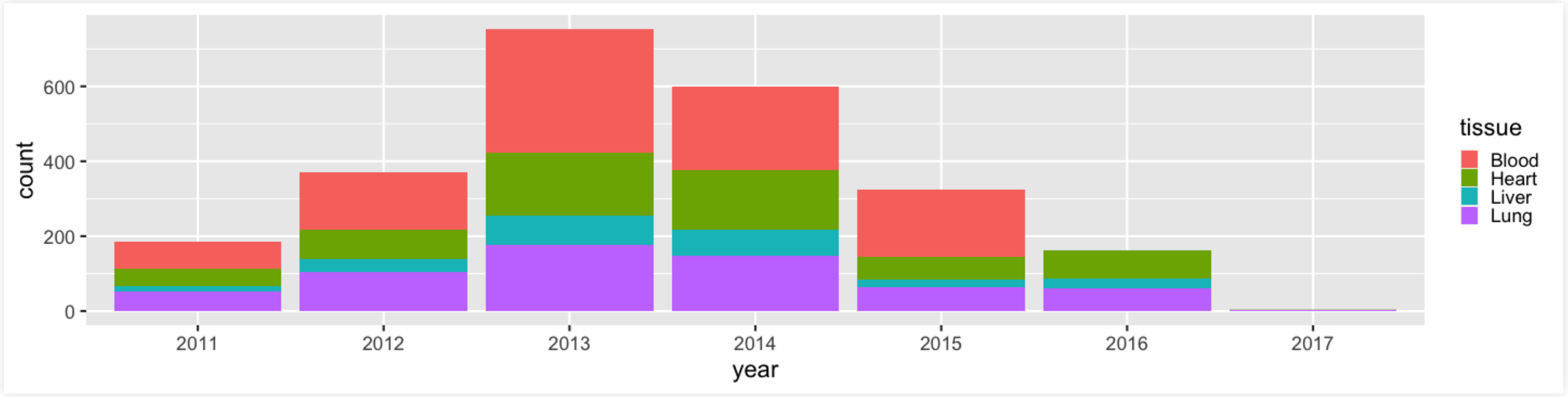
1. each "variable" has its own dedicated column
2. each "observation" has its own row
3. each type of observational unit has its own data frame

In our example, each of the **observations** are different **groups of samples**, each of which has an associated *tissue*, *year*, and *count*. These are the *variables* that are associated with the groups of samples.

Tidy data

Tidy data is easy to work with.

```
tidy %>%
  filter(tissue %in% c("Blood", "Heart", "Liver", "Lung")) %>%
  ggplot() +
  geom_bar(aes(x = year, y = count, fill = tissue), stat = 'identity')
```

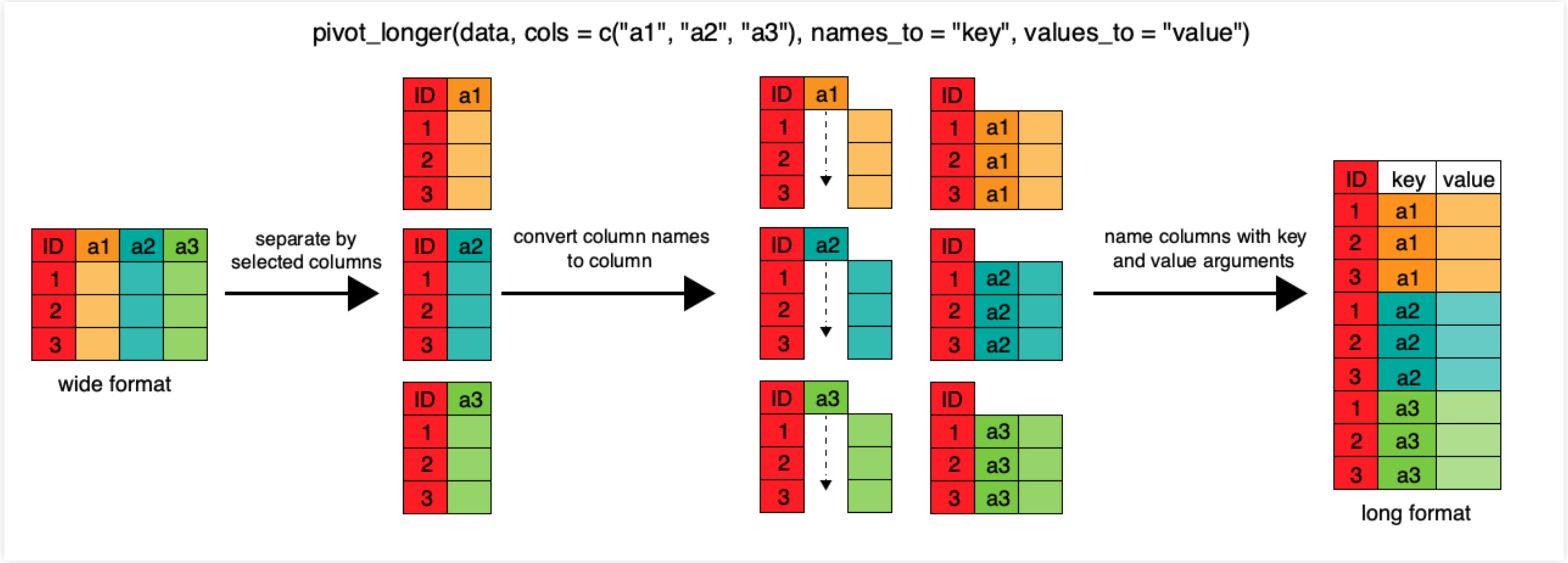


Tidying data with pivot_longer()

- `tidyr::pivot_longer()` is the function you will most often want to use to tidy your data

```
gtex_time_tissue_data %>%
  pivot_longer(-tissue, names_to = "year", values_to = "count") %>%
  head(2L)
# A tibble: 2 x 3
  tissue         year  count
  <chr>          <chr> <dbl>
1 Adipose Tissue 2011     56
2 Adipose Tissue 2012    107
```

- the three important arguments are: a) a selection of columns, b) the name of the new key column, and c) the name of the new value column

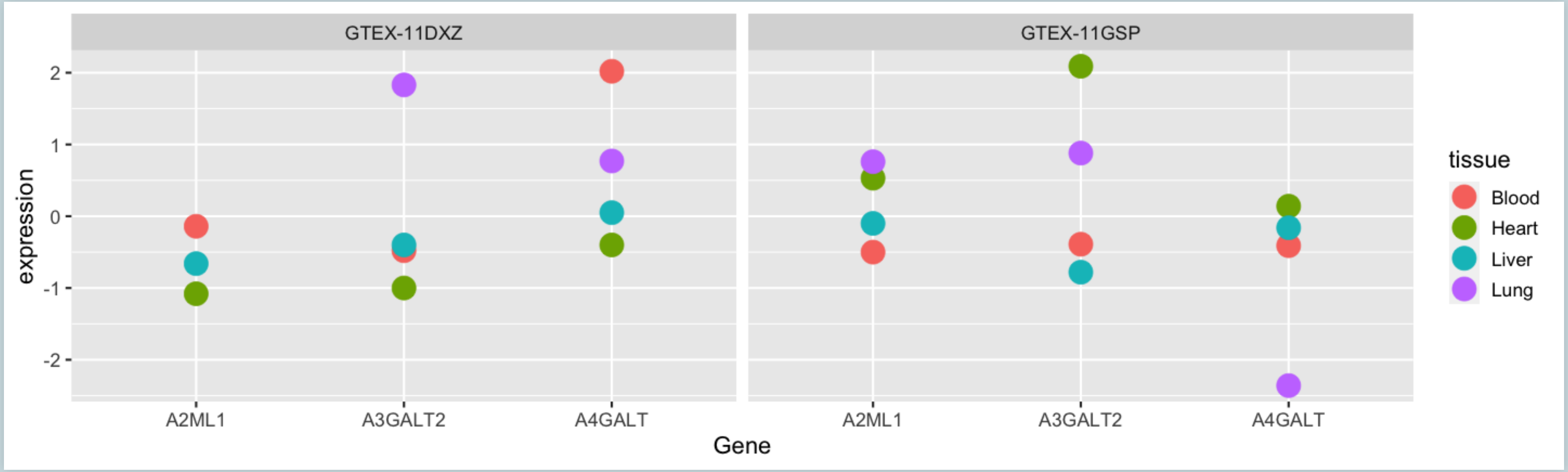


pivot_longer(data, cols = c("a1", "a2", "a3"), names_to = "key", values_to = "value")
ID a1 a2 a3
1
2
3
wide format
separate by selected columns
ID a1
ID a2
ID a3
convert column names to column
ID
a1
a2
a3
name columns with key and value arguments
ID key value
long format

Exercise: cleaning GTEX

```
head(gtex_data, 3L)
# A tibble: 3 x 7
  Gene  Ind         Blood Heart  Lung Liver NTissues
  <chr> <chr>       <dbl> <dbl> <dbl> <dbl>    <dbl>
1 A2ML1 GTEX-11DXZ -0.14 -1.08 NA    -0.66        3
2 A2ML1 GTEX-11GSP -0.5   0.53  0.76 -0.1         4
3 A2ML1 GTEX-11NUK -0.08 -0.4  -0.26 -0.13        4
```

Use the GTEX data to reproduce the following plot:



The individuals and genes of interest are `c('GTEX-11GSP', 'GTEX-11DXZ')` and `c('A2ML1', 'A3GALT2', 'A4GALT')`, respectively.

”Messy” data is relative and not always bad

```
# A tibble: 4 x 3
  mouse weight_before weight_after
  <dbl>         <dbl>        <dbl>
1     1         11.2         13.8
2     2         10.1         12.9
3     3         10.4          9.72
4     4          7.96         8.39
```

```
wide_mice %>%
  mutate(weight_gain = weight_after -
weight_before) %>%
  select(mouse, weight_gain)
# A tibble: 4 x 2
  mouse weight_gain
  <dbl>       <dbl>
1     1       2.58
2     2       2.84
3     3      -0.637
4     4       0.430
```

```
# A tibble: 8 x 3
  mouse time   weight
  <dbl> <chr>   <dbl>
1     1 before  11.2
2     1 after   13.8
3     2 before  10.1
4     2 after   12.9
5     3 before  10.4
6     3 after    9.72
7     4 before   7.96
8     4 after    8.39
```

```
long_mice %>%
  group_by(mouse) %>%
  mutate(weight_gain = weight -
lag(weight)) %>%
  filter(!is.na(weight_gain)) %>%
  select(mouse, weight_gain)
# A tibble: 4 x 2
# Groups:   mouse [4]
  mouse weight_gain
  <dbl>       <dbl>
1     1       2.58
2     2       2.84
3     3      -0.637
4     4       0.430
```

Pivoting wider

- As we saw with the mouse example, sometimes our data is actually easier to work with in the “wide” format.
- wide data is also often nice to make tables for presentations, or is (unfortunately) sometimes required as input for other software packages
- To go from long to wide, we use `pivot_wider()`:

```
long_mice
# A tibble: 8 x 3
  mouse time   weight
  <dbl> <chr>   <dbl>
1     1 before  11.2
2     1 after   13.8
3     2 before  10.1
4     2 after   12.9
5     3 before  10.4
6     3 after    9.72
7     4 before   7.96
8     4 after    8.39
```

```
long_mice %>%
  pivot_wider(
    names_from = time,
    values_from = weight
  )
# A tibble: 4 x 3
  mouse before after
  <dbl>  <dbl> <dbl>
1     1  11.2  13.8
2     2  10.1  12.9
3     3  10.4   9.72
4     4   7.96  8.39
```

Names prefix

```
long_mice
# A tibble: 8 x 3
  mouse time   weight
  <dbl> <chr>   <dbl>
1     1 before  11.2
2     1 after   13.8
3     2 before  10.1
4     2 after   12.9
5     3 before  10.4
6     3 after    9.72
7     4 before   7.96
8     4 after    8.39
```

- you can use `names_prefix` to make variables names that are more clear in the result

```
long_mice %>%
  pivot_wider(
    names_from = time,
    values_from = weight,
    names_prefix = "weight_"
  ) %>%
  head(2L)
# A tibble: 2 x 3
  mouse weight_before weight_after
  <dbl>         <dbl>        <dbl>
1     1          11.2         13.8
2     2          10.1         12.9
```

- this can also be used to *remove* a prefix when going from wide to long:

```
wide_mice %>%
  pivot_longer(
    -mouse
    names_to = "time",
    values_to = "weight",
    names_prefix = "weight_"
  )
```

Exercise: creating a table

Use the GTEX data to make the following table:

```
[1] "Number of missing tissues:"
# A tibble: 2 x 4
# Groups:   Ind [2]
  Ind         A2ML1 A3GALT2 A4GALT
  <chr>       <int>   <int>  <int>
1 GTEX-11DXZ      1       0      0
2 GTEX-11GSP      0       0      0
```

The numbers in the table are the number of tissues in each individual for which the gene in question was missing.

Multi-pivoting

Have a look at the following data. How do you think we might want to make it look?

```
gtex_time_chunk_link =
  "https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-
2021/data/gtex_metadata/gtex_samples_tiss_time_chunk.csv"

gtex_samples_time_chunk =
  read_csv(file = gtex_time_chunk_link, col_types = cols())

head(gtex_samples_time_chunk)
# A tibble: 6 x 9
  tissue      `Sept-2015` `Sept-2016` `Oct-2015` `Oct-2016` `Nov-2015` `Nov-2016`
  <chr>             <dbl>       <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
1 Adipose T…            5          36          4         20         15         16
2 Adrenal G…            4           5          1          5          2          6
3 Blood                 2           0          6          0         33          0
4 Blood Ves…            9          24          7         26          9         17
5 Brain                12           2          3          9         17         24
6 Breast                9          19          7         20          5          8
# … with 2 more variables: Dec-2016 <dbl>, Dec-2015 <dbl>
```

The problem here is that the column names contain two pieces of data:

1. the year
2. the month it came from

Our use of `pivot_longer` has so far been to extract a single piece of information from the column name

Multi-pivoting

- Turns out this problem can be tackled too:

```
gtex_samples_time_chunk %>%
  pivot_longer(
    cols=contains("-201"), # selects columns that contain this
    names_pattern = "(\\D+)-(\\d+)", # a "regular expression"- we'll learn about
these later
    names_to = c(".value", "year")
  )
# A tibble: 54 x 6
   tissue          year   Sept   Oct   Nov   Dec
   <chr>           <chr> <dbl> <dbl> <dbl> <dbl>
 1 Adipose Tissue  2015      5     4    15     0
 2 Adipose Tissue  2016     36    20    16    15
 3 Adrenal Gland   2015      4     1     2     0
 4 Adrenal Gland   2016      5     5     6     2
 5 Blood           2015      2     6    33    17
 6 Blood           2016      0     0     0     0
 7 Blood Vessel    2015      9     7     9     0
 8 Blood Vessel    2016     24    26    17    12
 9 Brain           2015     12     3    17     0
10 Brain           2016      2     9    24     9
# … with 44 more rows
```

- We won't dig into this, but you should know that almost any kind of data-tidying problem can be solved with some combination of the functions in the `tidyr` package.
- See the online docs and vignettes for more info

Combining multiple tables with joins

Relational data

- Relational data are interconnected data that is spread across multiple tables, each of which usually has a different unit of observation
- When we get an expression dataset, the data is usually divided into an expression matrix with the expression values of each sample, and table(s) with metadata about the samples themselves.
- For the GTEx dataset, we have information about the samples, subjects, and experiment batches in additional data frames in addition to the expression matrix we've been working with.

```
gtex_metadata_link =
  "https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-
2021/data/gtex_metadata/gtex_sample_metadata.csv"

gtex_sample_data = read_csv(file = gtex_metadata_link, col_types = cols())

head(gtex_sample_data, 2L)
# A tibble: 2 x 6
  subject_id sample_id     batch_id center_id tissue rin_score
  <chr>      <chr>         <chr>    <chr>     <chr>      <dbl>
1 GTEX-11DXZ 0003-SM-58Q7X BP-39216 B1        Blood       NA
2 GTEX-11DXZ 0126-SM-5EGGY BP-44460 B1        Liver        7.9
```

- The sample data has information about the tissue and the subject who contributed the sample, the batch it was processed in, the center the sample was processed at, and the RNA integrity number (RIN score) for the sample.

Relational data

The subject data table contains some subject demographic information. Death refers to circumstances surrounding death.

```
gtex_subject_link =
  "https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/9e4fb21ccf93a83e2b6004b9aa467426806f8589/data/gtex_metadata/gtex_subject_metadata.csv"


gtex_subject_data = read_csv(file = gtex_subject_link, col_types = cols())

head(gtex_subject_data, 2L)
# A tibble: 2 x 4
  subject_id sex    age   death
  <chr>      <chr>  <chr> <chr>
1 GTEX-11DXZ male   50-59 ventilator
2 GTEX-11GSP female 60-69 sudden but natural causes
```

The batch data containts the batch type and the dates the batches were run (we were been working a bit with this date data aggregated into counts of samples earlier).

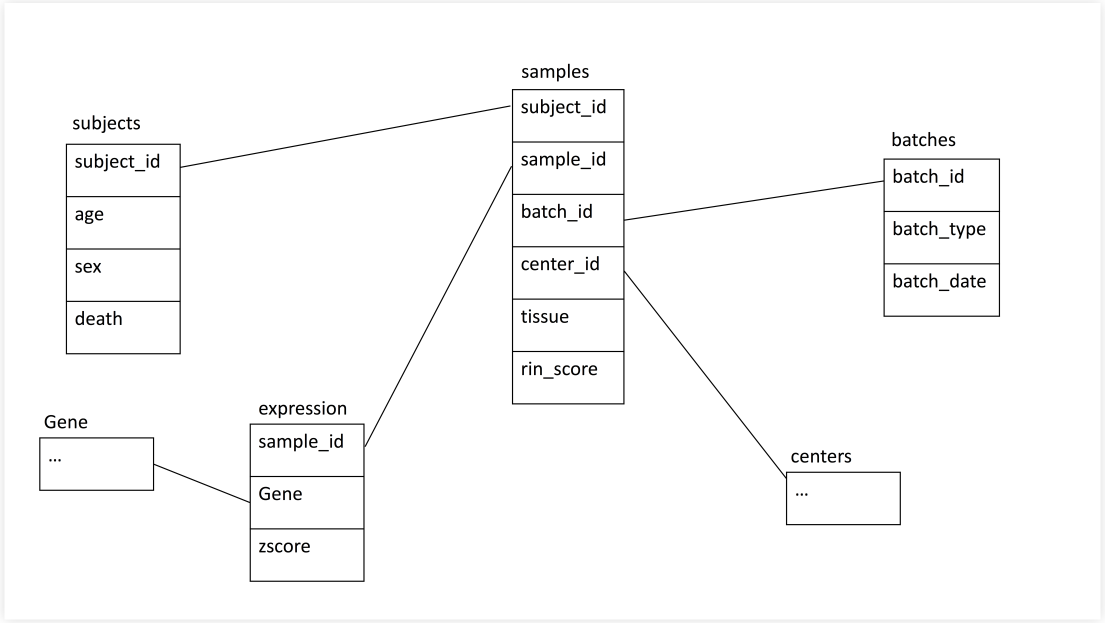
```
gtex_batch_link = "https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/9e4fb21ccf93a83e2b6004b9aa467426806f8589/data/gtex_metadata/gtex_batch_metadata.csv"

gtex_batch_data = read_csv(file = gtex_batch_link, col_types = cols())

head(gtex_batch_data, 2L)
# A tibble: 2 x 3
```

Relational data

- These data are not independent of each other. Subjects described in the `subject` data are referenced in the `sample` data, and the batches referenced in the `sample` data are in the `batch` data. The sample ids from the `sample` data are used for accessing expression data.



- `subject` connects to `sample` via a single variable, `subject_id`.
- `sample` connects to `batch` through the `batch_id` variable.

Relational + tidy data

For the expression data, we have been using the `gtex_data` expression data frame:

```
gtex_data = read_tsv(file = gtex_link, col_types = cols())

gtex_data %>%
  head(2L)
# A tibble: 2 x 7
  Gene  Ind        Blood Heart  Lung Liver NTissues
  <chr> <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl>
1 A2ML1 GTEX-11DXZ -0.14 -1.08 NA    -0.66        3
2 A2ML1 GTEX-11GSP -0.5   0.53  0.76 -0.1         4
```

The expression data on the previous slide is formatted slightly differently:

```
gtex_expression_link = "https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/9e4fb21ccf93a83e2b6004b9aa467426806f8589/data/gtex_metadata/gtex_expression.csv"


gtex_expression = read_csv(file = gtex_expression_link, col_types = cols())

gtex_expression %>%
  head(2L) # note: sample_id replaces Ind + tissue
# A tibble: 2 x 3
  sample_id     Gene  zscore
  <chr>         <chr>  <dbl>
1 0003-SM-58Q7X A2ML1  -0.14
2 0326-SM-5EGH1 A2ML1  -1.08
```

An example join

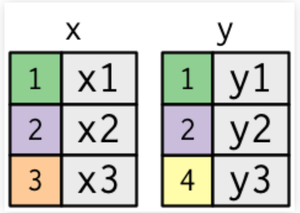
- Imagine we want to add subject information to the sample data
- We can accomplish that with a **join**:

```
gtex_sample_data %>%
  inner_join(gtex_subject_data, by = "subject_id")
# A tibble: 312 x 9
   subject_id sample_id  batch_id center_id tissue rin_score sex   age   death
   <chr>      <chr>      <chr>    <chr>     <chr>      <dbl> <chr> <chr> <chr>
 1 GTEX-11DXZ 0003-SM-5… BP-39216 B1        Blood       NA   male  50-59 ventil…
 2 GTEX-11DXZ 0126-SM-5… BP-44460 B1        Liver        7.9 male  50-59 ventil…
 3 GTEX-11DXZ 0326-SM-5… BP-44460 B1        Heart        8.3 male  50-59 ventil…
 4 GTEX-11DXZ 0726-SM-5… BP-43956 B1        Lung         7.8 male  50-59 ventil…
 5 GTEX-11GSP 0004-SM-5… BP-39412 B1        Blood       NA   fema… 60-69 sudden…
 6 GTEX-11GSP 0626-SM-5… BP-44902 B1        Liver        6.2 fema… 60-69 sudden…
 7 GTEX-11GSP 0726-SM-5… BP-44902 B1        Lung         6.9 fema… 60-69 sudden…
 8 GTEX-11GSP 1226-SM-5… BP-44902 B1        Heart        7.9 fema… 60-69 sudden…
 9 GTEX-11NUK 0004-SM-5… BP-39723 B1        Blood       NA   male  50-59 sudden…
10 GTEX-11NUK 0826-SM-5… BP-43730 B1        Lung         7.4 male  50-59 sudden…
# … with 302 more rows
```

Joins

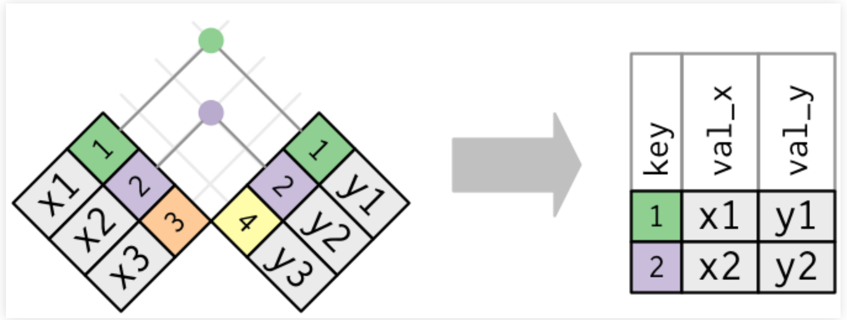
```
x = tibble(
  key = c(1, 2, 3),
  val_x = c("x1", "x2", "x3")
)

y = tibble(
  key = c(1, 2, 4),
  val_y = c("y1", "y2", "y3")
)
```



```
inner_join(x, y, by = "key")
# A tibble: 2 x 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
```

- An inner join matches pairs of observations when their "keys" are equal
- the column that is joined on is specified as a "key" with the argument `by="column"`

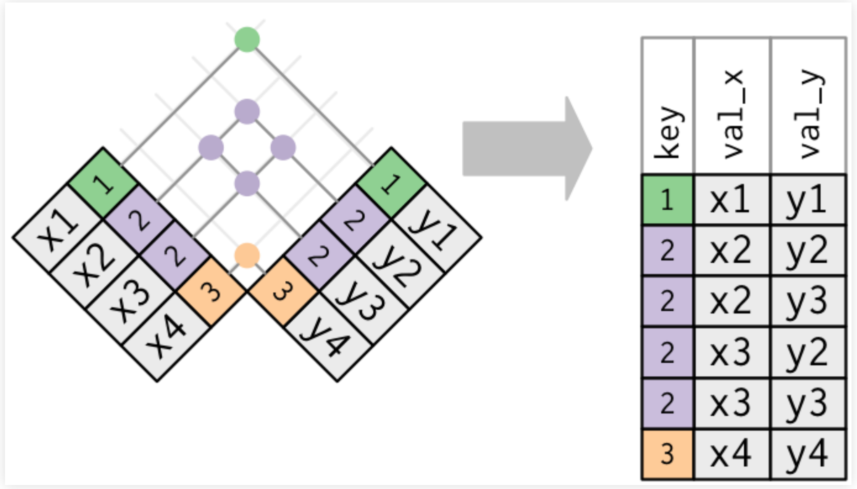


Duplicate keys

```r
x = tibble(
  key = c(1, 2, 2, 3),
  val_x = c("x1", "x2", "x3", "x4")
)

y = tibble(
  key = c(1, 2, 2, 4),
  val_y = c("y1", "y2", "y3", "y4")
)
```

```r
inner_join(x, y, by = "key")
# A tibble: 5 x 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
3     2 x2    y3
4     2 x3    y2
5     2 x3    y3
```



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

When keys are duplicated, multiple rows can match multiple rows, so each possible combination is produced

Specifying the keys

```
gtex_sample_data %>%
  inner_join(gtex_subject_data, by = "center_id")
Error: Join columns must be present in data.
x Problem with `center_id`.
```

- Why does this fail?

Specifying the keys

- When keys have different names in different dataframes, the syntax to join is:

```
head(gtex_data, 2)
# A tibble: 2 x 7
  Gene  Ind         Blood Heart  Lung Liver NTissues
  <chr> <chr>       <dbl> <dbl> <dbl> <dbl>    <dbl>
1 A2ML1 GTEX-11DXZ -0.14 -1.08 NA    -0.66        3
2 A2ML1 GTEX-11GSP -0.5   0.53  0.76 -0.1         4
head(gtex_subject_data, 2)
# A tibble: 2 x 4
  subject_id sex    age   death
  <chr>      <chr>  <chr> <chr>
1 GTEX-11DXZ male   50-59 ventilator
2 GTEX-11GSP female 60-69 sudden but natural causes

gtex_data %>%
  inner_join(gtex_subject_data, by = c("Ind" = "subject_id")) %>%
  head(5L)
# A tibble: 5 x 10
  Gene  Ind        Blood Heart  Lung Liver NTissues sex   age   death
  <chr> <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl> <chr> <chr> <chr>
1 A2ML1 GTEX-11D… -0.14 -1.08 NA    -0.66        3 male  50-59 ventilator
2 A2ML1 GTEX-11G… -0.5   0.53  0.76 -0.1         4 fema… 60-69 sudden but natur…
3 A2ML1 GTEX-11N… -0.08 -0.4  -0.26 -0.13        4 male  50-59 sudden but natur…
4 A2ML1 GTEX-11N… -0.37  0.11 -0.42 -0.61        4 male  60-69 sudden but natur…
5 A2ML1 GTEX-11T…  0.3  -1.11  0.59 -0.12        4 male  20-29 ventilator
```

Exercise: finding expression of specific samples

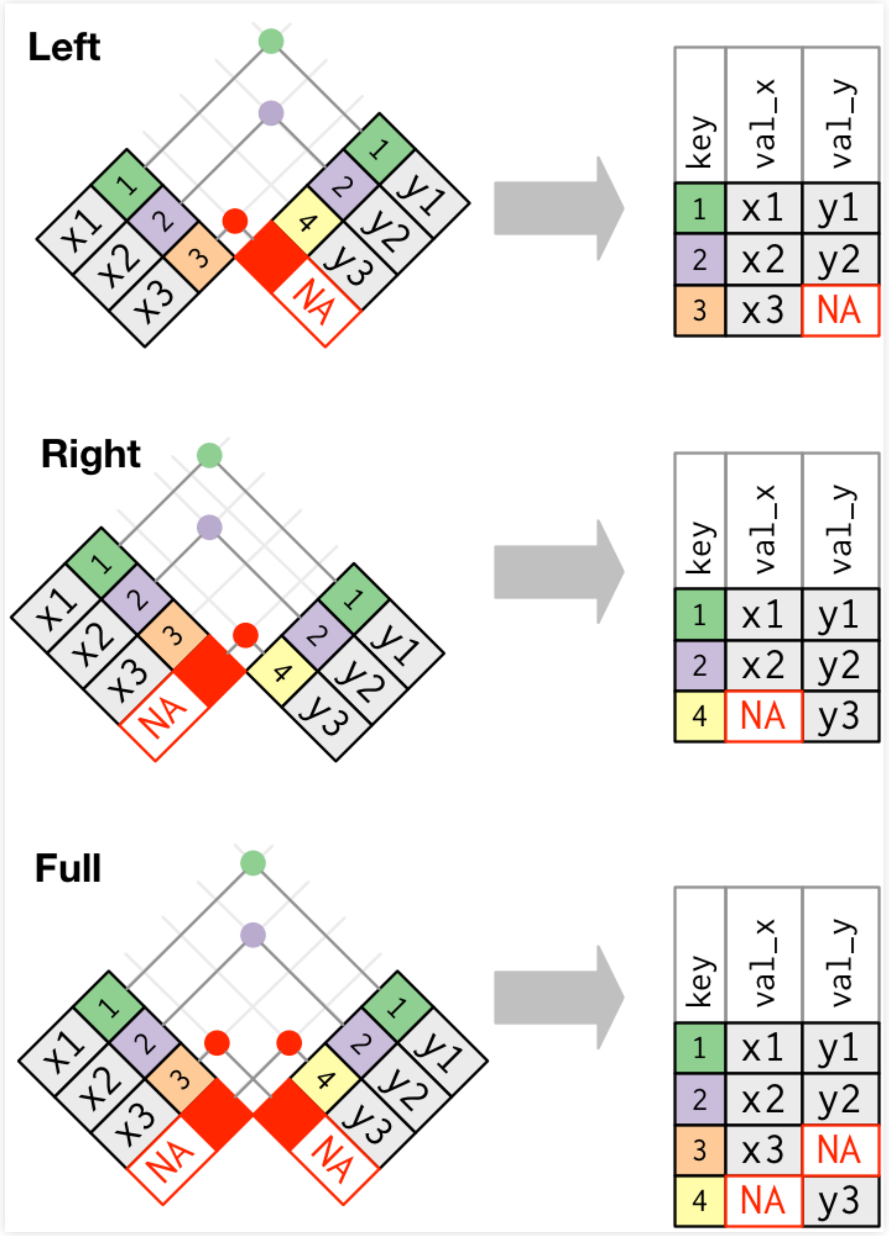
Use joins to find the samples collected in 2015 with high blood expression (Z>3) of "KRT19" in males. Start with the `batch_data_year`; this data has an extra extracted column with the year (we'll go over how this worked in the next lecture).

```
batch_data_year =
  gtex_batch_data %>%
  mutate(
    batch_date = lubridate::mdy(batch_date),
    year = lubridate::year(batch_date)
  )

head(batch_data_year, 2L)
# A tibble: 2 x 4
  batch_id batch_type                                    batch_date  year
  <chr>    <chr>                                         <date>     <dbl>
1 BP-38516 DNA isolation_Whole Blood_QIAGEN Puregene (Manual) 2013-05-02  2013
2 BP-42319 RNA isolation_PAXgene Tissue miRNA            2013-08-14  2013
```

Note that you'll have to join to other data frames the `sample` data frame to put this together.

Other joins

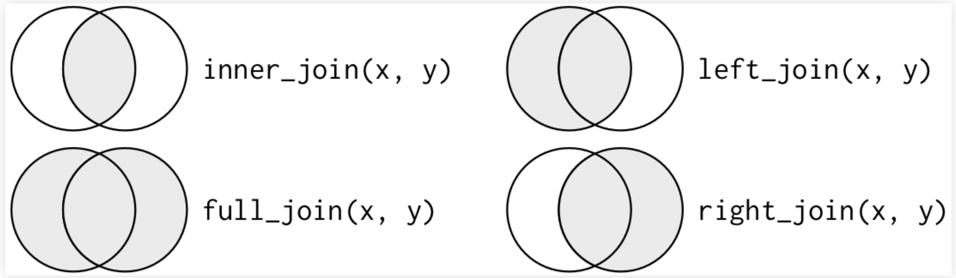


key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

- A left join keeps all observations in `x`.
- A right join keeps all observations in `y`.
- A full join keeps all observations in `x` and `y`.



- Left join should be your default
  - it looks up additional information in other tables
  - preserves all rows in the table you're most interested in

Joining on multiple columns

- It is often desirable to find matches along more than one column, such as month and year in this example. Here we're joining tissue sample counts with total sample counts.

```
gtex_tissue_month_link = "https://raw.githubusercontent.com/alejandroschuler/r4ds-
courses/9e4fb21ccf93a83e2b6004b9aa467426806f8589/data/gtex_metadata/gtex_tissue_month_year.csv"


gtex_tissue_month =
  read_csv(file = gtex_tissue_month_link, col_types = cols()) %>%
  filter(tissue %in% c("Blood", "Heart", "Liver", "Lung"))

head(gtex_tissue_month, 2L)
# A tibble: 2 x 4
  tissue month  year tiss_samples
  <chr>  <dbl> <dbl>        <dbl>
1 Blood      1  2012           25
2 Blood      1  2013           16

gtex_samples_by_month =
  read_csv(file = gtex_samples_time_link, col_types = cols()) %>%
  rename(total_num_samples = num_samples)

head(gtex_samples_by_month, 2L)
# A tibble: 2 x 3
  month  year total_num_samples
  <dbl> <dbl>             <dbl>
1     5  2011                20
2     6  2011                44
```

Joining on multiple columns

This is also possible if the columns have different names:

```
gtex_data_long = gtex_data %>%
  pivot_longer(cols = c("Blood", "Heart", "Lung", "Liver"), names_to = "tissue",
    values_to = "zscore")
head(gtex_data_long, n = 2L)
# A tibble: 2 x 5
  Gene  Ind        NTissues tissue zscore
  <chr> <chr>         <dbl> <chr>   <dbl>
1 A2ML1 GTEX-11DXZ        3 Blood   -0.14
2 A2ML1 GTEX-11DXZ        3 Heart   -1.08
head(gtex_sample_data, n = 2L)
# A tibble: 2 x 6
  subject_id sample_id     batch_id center_id tissue rin_score
  <chr>      <chr>         <chr>    <chr>     <chr>      <dbl>
1 GTEX-11DXZ 0003-SM-58Q7X BP-39216 B1        Blood         NA
2 GTEX-11DXZ 0126-SM-5EGGY BP-44460 B1        Liver        7.9

gtex_data_long %>%
  inner_join(gtex_sample_data, by = c("tissue", "Ind" = "subject_id")) %>%
  head(n = 4L)
# A tibble: 4 x 9
  Gene  Ind        NTissues tissue zscore sample_id    batch_id center_id rin_score
  <chr> <chr>         <dbl> <chr>   <dbl> <chr>        <chr>    <chr>         <dbl>
1 A2ML1 GTEX-11…          3 Blood   -0.14 0003-SM-58… BP-39216 B1               NA
2 A2ML1 GTEX-11…          3 Heart   -1.08 0326-SM-5E… BP-44460 B1              8.3
3 A2ML1 GTEX-11…          3 Lung    NA    0726-SM-5N… BP-43956 B1              7.8
4 A2ML1 GTEX-11…          3 Liver   -0.66 0126-SM-5E… BP-44460 B1              7.9
```

Join problems

- Joins can be a source of subtle errors in your code
- check for `NA`s in variables you are going to join on
- make sure rows aren't being dropped if you don't intend to drop rows
    - checking the number of rows before and after the join is not sufficient. If you have an inner join with duplicate keys in both tables, you might get unlucky as the number of dropped rows might exactly equal the number of duplicated rows
- `anti_join()` and `semi_join()` are useful tools (filtering joins) to diagnose problems
    - `anti_join()` keeps only the rows in `x` that *don't* have a match in `y`
    - `semi_join()` keeps only the rows in `x` that *do* have a match in `y`

Exercise: Looking for variables related to data missingness

It is important to make sure that the missingness in the expression data is not related to variables present in the data. Use the tables `batch_data_year`, `sample_data`, `subject_data`, and the `gtex_data` to look at the relationship between missing gene values and other variables in the data.