Summer R

Session 4: Computing variables, joining tables, and working with missing data

Dominic Bordelon, Research Data Librarian, ULS

Agenda

1. Working with missing data
2. Computing new variables
3. Data viz: faceting
4. Joining tables

More dplyr

We have used **{dplyr}** to…

- select variables/columns
- sort rows
- filter
- group
- summarize

Today with dplyr we will also:

- create new variables in a data frame
- combine tables

We'll also use **{naniar}** to help us visualize missing data.

```r
library(tidyverse)
library(medicaldata)

# install.packages("naniar")
library(naniar)
```

Homework review, questions

Working with missing data

Representations of missingness in R

These are the ways that R represents missing or unknown information:

- `NA`: for a missing value ("not available")
  - logical constant, length 1
  - can be coerced into vectors of other types (numeric, character, etc.)
- `NULL`: an *object* for an undefined result from a function
- `NaN`: for non-real numbers ("not a number")
- `Inf` and `-Inf`: for ∞ and $-\infty$

We will concern ourselves only with `NA`, but it's good to know they all exist.

NA: the missing value

NA is the value for a missing datum. You can think of it like an empty Excel cell.

What NA is *not*:

- NA is not character data (NA and "NA" are different things)
- NA is not zero, because zero has a defined position on the number line
- NA is not necessarily a problem, depending on your situation (though sometimes it is)

Conceptual categories of missingness (1)

When examining a data set, you'll want to see how much data are missing and assess the type of missingness:

- Missing completely at random (MCAR): independent of observable variables, and of parameters of interest
- Missing at random (MAR): missingness depends on observed data, not the missing value
  - Examples: uniform non-response within classes; multiple (varying number) attempts in a process; subject removal due to experimental protocol
  - Impossible to verify statistically—an assumption which needs to be made by the analyst using all real-world information available (i.e., data provenance)

Conceptual categories of missingness (2)

- Missing, not at random (MNAR): depends on the *missing value*
  - "Nonignorable nonresponse": a subject does not respond *because* of their abnormal level
  - Treating these observations as representative (i.e., purely random) will introduce a bias into our analysis!

Which type of missingness you have determines whether a bias is introduced, and how you need to account for it statistically.

Vector functions for working with NA

Many functions whose calculations would be tainted by NA—for example, `mean()`—accept an `na.rm=TRUE` argument.

These functions are useful with NA:

- **`is.na()`** (and `!is.na()`): vectorized check, is the value NA?
  - to get a count of NA in a vector, wrap with `sum()`: `sum(is.na(polyps$age))`
- **`anyNA()`**: return single T/F, are there any NA in the vector?
- `my_vector[!is.na(my_vector)]` strips NA from `my_vector`
  - The square brackets `[ ]` are a base R syntax for filtering vectors
- To replace NA in `x` → a value `y`: `tidyr::replace_na(x, y)`
  - Or `dplyr::coalesce(x, y)` for a more powerful version
- To replace value `y` in `x` → NA: `na_if(x, y)`

Data frame functions for working with NA

- `tidyr::drop_na()` drops (removes) rows with `NA` in any column
  - `drop_na(x, y)` drops rows with `NA` in columns `x`, `y`, or both
- `dplyr::filter()` and `mutate()` can use vector-based functions such as `is.na()`

Visualizing NA with naniar

- `naniar::vis_miss()` shows the `NA` in a data frame
- `gg_miss_upset()`: patterns among variables
- `gg_miss_fct()`: patterns among factor levels (categorical levels)

There are several more, but the three above are most useful. You can browse the naniar gallery or try these out with any data frame.

ICA 4.1: NA, missing data

Computing new variables

`mutate()`: add a new column

- `dplyr::mutate(.data, ...)` creates new columns, usually from existing ones.
- The `...` argument(s) are expression(s) evaluated for each row, forming values for the new column.
- Use a named argument to assign a name to the column
- OK to use an existing name—but it will overwrite that column, of course

```
1  # create a BMI variable in the medicaldata::smartpill data set:
2  names(smartpill)[1:6]
```

```
[1] "Group"  "Gender" "Race"   "Height" "Weight" "Age"
```

```
1  smartpill %>%
2    mutate(BMI = Weight / ((Height*0.01)^2),
3           .before = Height) %>%
4    select(1:7) %>%
5    head(6)
```

```
  Group Gender Race      BMI Height    Weight Age
1     0      1   NA 30.51515 182.88 102.05820  25
2     0      1   NA 31.38078 180.34 102.05820  39
3     0      1   NA 20.92052 180.34  68.03880  44
4     0      1   NA 22.74157 175.26  69.85317  53
5     0      0   NA 19.33440 152.40  44.90561  57
6     0      1   NA 27.57392 185.42  94.80073  43
```

Convert an existing column's data type

Example: like many plaintext-sourced data sets, `covid_testing` columns are auto-encoded as numeric or character upon import. Now, we want `gender` to be represented properly as a categorical variable or *factor*.

```
1  covid_testing %>%
2    select(subject_id, gender) %>%
3    summary()
```

```
  subject_id       gender
Min.   :    1   Length:15524
1st Qu.: 2330   Class :character
Median : 5268   Mode  :character
Mean   : 5571
3rd Qu.: 8636
Max.   :12346
```

```
1  covid_testing %>%
2    mutate(gender = as_factor(gender)) %>%
3    select(subject_id, gender) %>%
4    summary()
```

```
  subject_id       gender
Min.   :    1   female:7832
1st Qu.: 2330   male  :7692
Median : 5268
Mean   : 5571
3rd Qu.: 8636
Max.   :12346
```

`if_else()`: apply decision logic to values

- When we want to dictate decision-making in the column's value, we need to use Boolean programming logic
- Use `if_else(condition, true, false)` within the `mutate()` call.

Scenario starting from `covid_testing`: `subject_id` varies from 1 to 15524. Suppose that ID's below 5000 were assigned to the placebo group, and 5000 and above were assigned to the treatment group.

```
covid_testing %>%
  select(1, 4, age) %>%   # ignore most columns
  mutate(treatment_group = if_else(subject_id < 5000,
                                   "placebo",
                                   "treatment"),
         .after = subject_id) %>%
  slice_sample(n=6)        # randomly sample 6 rows
```

```
# A tibble: 6 × 4
  subject_id treatment_group gender   age
       <dbl> <chr>           <chr>  <dbl>
1       7753 treatment       male     1
2        501 placebo         female   2
3         88 placebo         female  30
4       1745 placebo         male    38
5       3542 placebo         female   0.1
6       6392 treatment       female   8
```

`case_when()`: >2 decision cases

- For if-else logic beyond two choices, `case_when()` is recommended
- Case expressions are evaluated in order until one of them returns `TRUE`

Scenario with `covid_testing` ID's: suppose instead of two groups, we have four, with boundaries of 1–2300, 2301–5200, 5201–8600, and 8601–15524.

```r
# note the syntax; ~ reads like "then"

covid_testing %>%
  select(1, 4, age) %>%
  mutate(treatment_group = case_when(
    subject_id <= 2300 ~ "placebo",
    subject_id <= 5200 ~ "group A",
    subject_id <= 8600 ~ "group B",
    .default = "group C"      # captures everything above 8600
  ),
         .after = subject_id) %>%
  slice_sample(n=8)
```

```
# A tibble: 8 × 4
  subject_id treatment_group gender   age
       <dbl> <chr>           <chr>  <dbl>
1      10261 group C         female  26
2       3485 group A         male    15
3       1960 placebo         male     0
4       5125 group A         male     5
5       1045 placebo         female  46
6       1242 placebo         female   0.7
7       8410 group B         female   2
8       1977 placebo         female  18
```

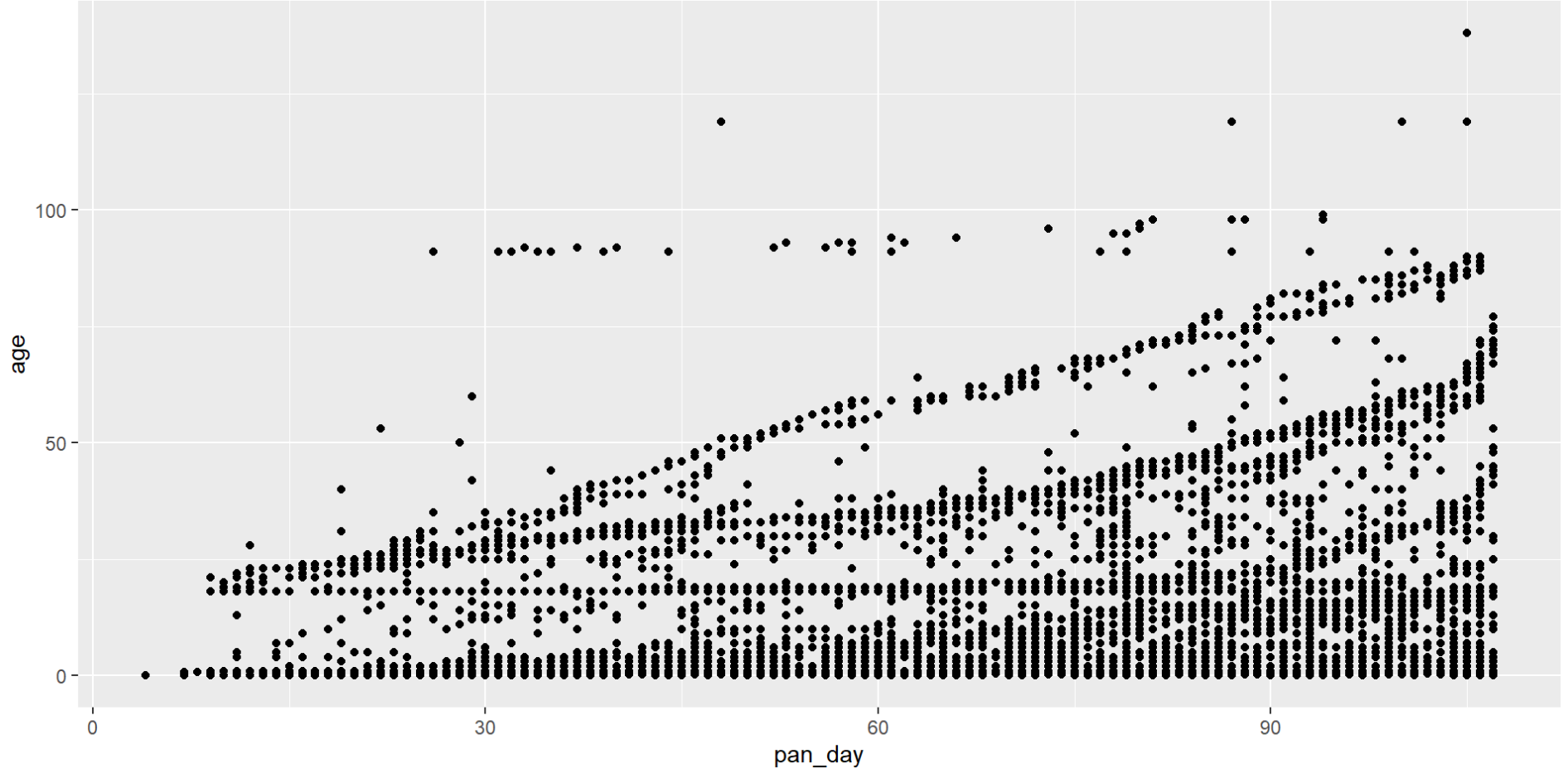
Data viz: faceting

`facet_grid()` and `facet_wrap()`: add faceting to a ggplot

- Faceting of a discrete or categorical variable, AKA small multiples, means breaking out mini-plots for each of the present values/levels.
- Layouts: columns, rows, a grid, or rectangles that stack/flow
- Awkward gotcha: wrap variable/column names in `vars()` (see examples on following slides)
- ⚠ Important rule of facets: axes must always match especially in scale! (The `facet_` functions take care of this for us, but be careful if you ever make your own manually or in a different setting.)

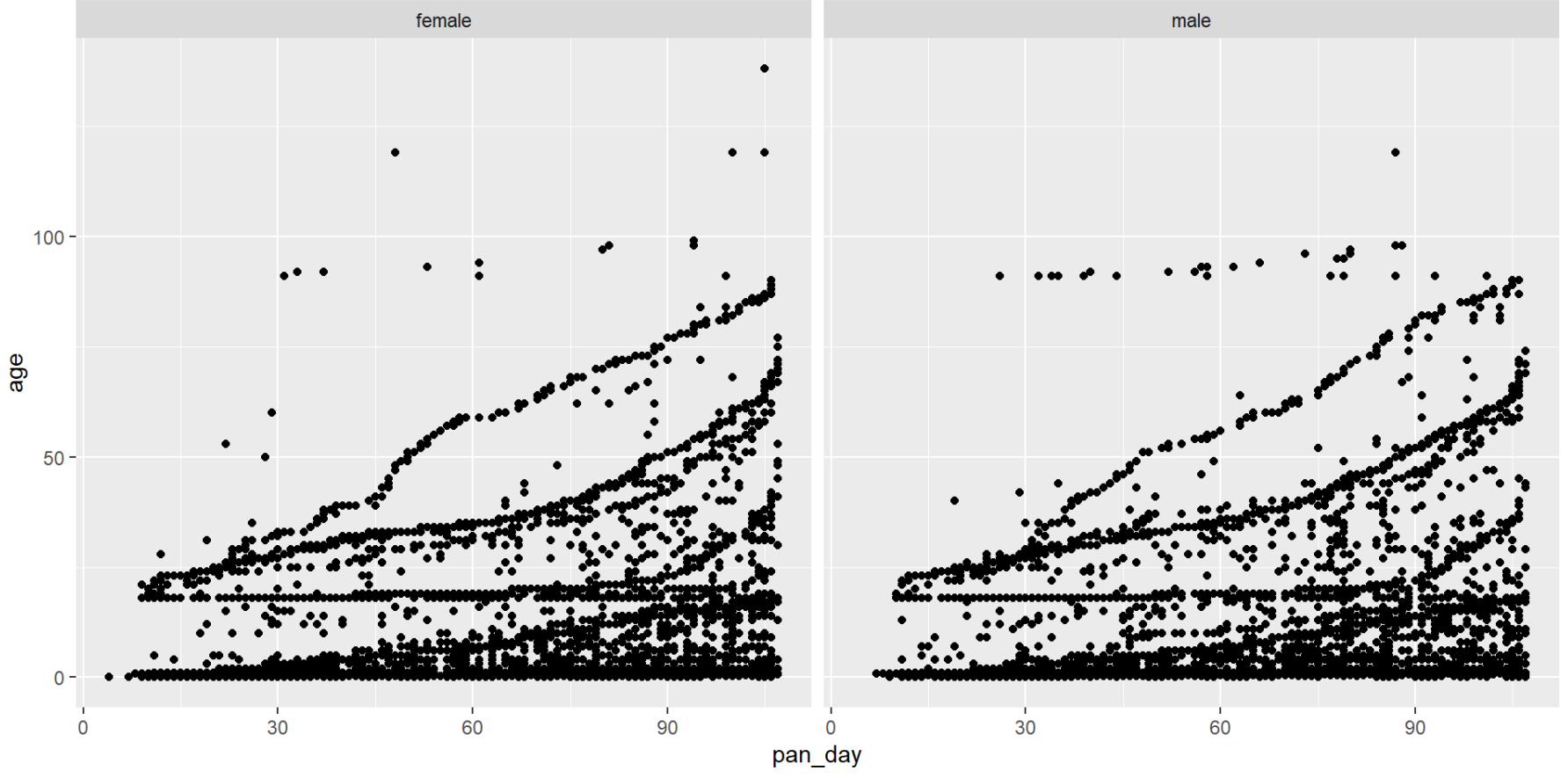
Pre-faceting, scatter plot of pandemic day and patient age:

```
1  covid_testing %>%
2    ggplot() +
3    geom_point(aes(pan_day, age))
```



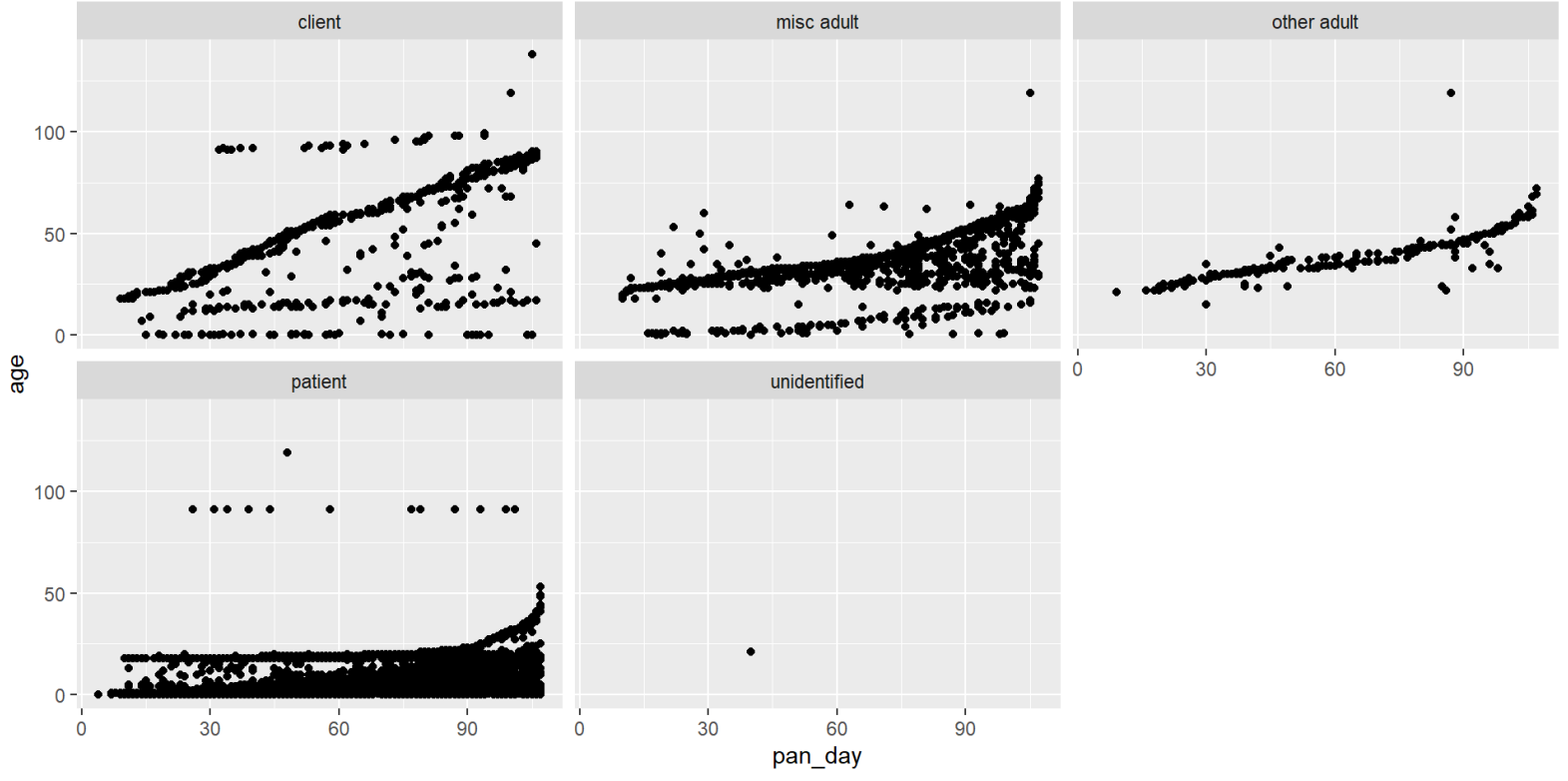
Gender faceting into columns:

```
1  covid_testing %>%
2    ggplot() +
3    geom_point(aes(pan_day, age)) +
4    facet_grid(cols=vars(gender))
```



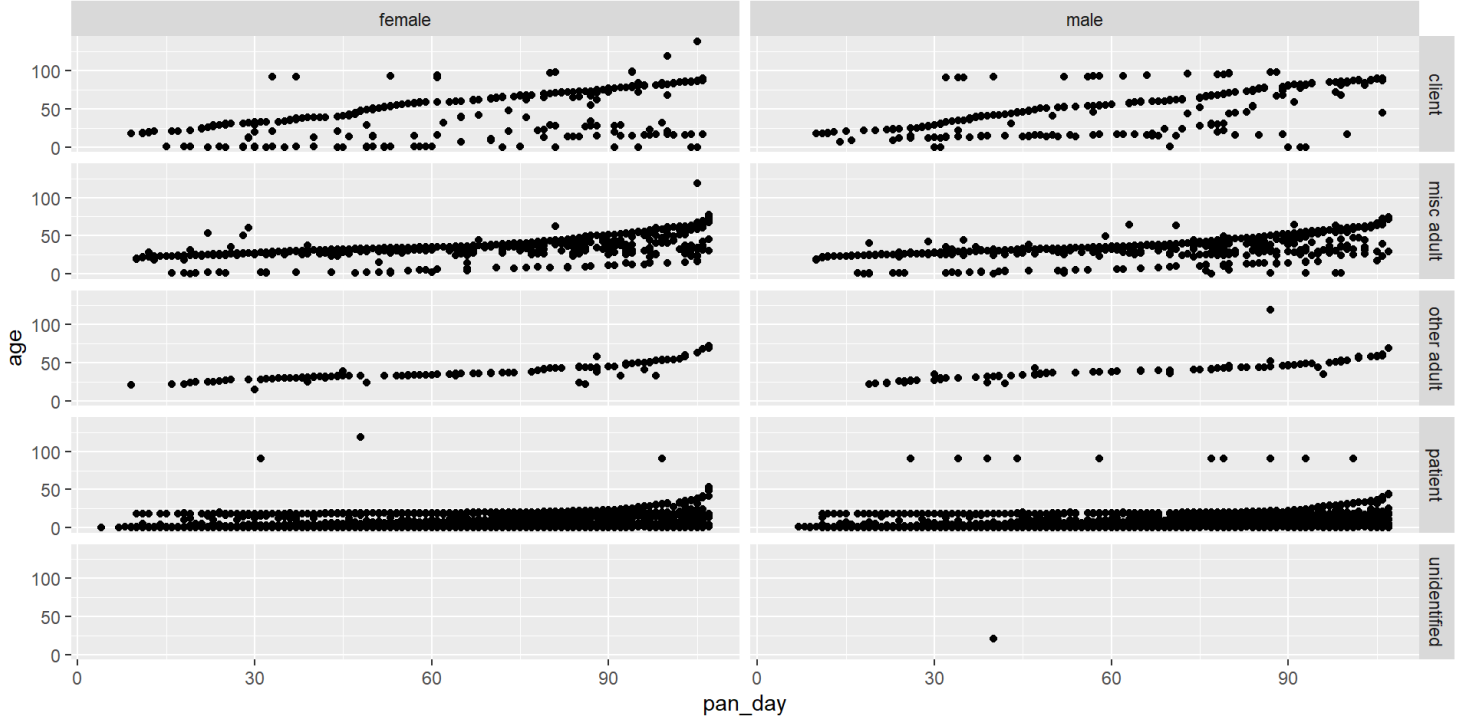
For more than ~3 groups, use `facet_wrap()` to flow the subplots onto multiple rows:

```
1  covid_testing %>%
2    ggplot() +
3    geom_point(aes(pan_day, age)) +
4    facet_wrap(vars(demo_group))
```



Lastly to facet by *two* variables, in a grid: (note the limited legibility of narrow plots!)

```
1  covid_testing %>%
2    ggplot() +
3    geom_point(aes(pan_day, age)) +
4    facet_grid(cols=vars(gender),
5               rows=vars(demo_group))
```



ICA 4.2: New variables, faceting

Joining tables

Relational data and table joins

- Much of the world's data lives in *relational database management systems* (RDBMSes), or simply "relational databases"
- Sometimes you might work with one of these directly; other times you might receive CSV files which originated in a RDBMS
- "Relational" == Table A has a relationship to Table B, via column X which appears in both tables ("foreign key relationship")
- Why relational?
  - One big table gets to be unmanageable, but we also don't want to duplicate info in the multiple places that we need it
  - Often-repeated values (e.g., a drug name) can be more efficiently stored/retrieved if they are replaced by a number, which points to the human-readable name.

Joining synthesizes a new a table from two or more parent tables.

Relationship types

- **One-to-one relationships**: one row in Table A corresponds to one row in Table B
- **One-to-many relationships**: “lookup” function; controlled vocabulary; Table B describes categories that are referenced in Table A
- **Many-to-many relationships**: there are many subjects (Table A), and many genetic markers (Table B); a patient may have an unspecified number of markers and vice versa, a relationship described in Table C

It can be helpful to have these relationships in mind when thinking about different ways we might want to join tables together.

Types of joins

Join type asks, “if I have unpaired rows when I join Tables A and B, what do I do with those rows?”

- Full join: keep all rows from both tables, filling NA as needed
- Inner join: keep only rows that appear in *both* tables
- Left join: inner join + keep all of the rows from the *left* table, filling NA as needed
- Right join: inner join + keep all of the rows from the *right* table, filling NA as needed
- Filtering joins: filter rows of Table A based on presence/absence of the row’s ID in Table B

Each of the above joins might produce a different number of rows.

```r
lg_sample <- read_csv("data/lg_sample.csv")
asa_status <- read_csv("data/asa_status.csv")
```

Consider these two tables:

```r
# 6-row sample of licorice_gargle

lg_sample %>%
  select(1:3)
```

```
# A tibble: 6 × 3
  preOp_gender preOp_asa
preOp_calcBMI
         <dbl>     <dbl>
<dbl>
1            0         2
31.3
2            0         1
32.3
3            1         3
30.5
4            0         3
32.2
5            1        NA
36.3
6            0         2
```

```r
# listing of ASA statuses

asa_status
```

```
# A tibble: 4 × 2
  asa_status_id asa_status
          <dbl> <chr>
1             1 a normal healthy
patient
2             2 a patient with mild
systemic disease
3             3 a patient with severe
systemic disease
4             4 UNUSED CATEGORY
```

`full_join()`: combine all data

```
1  lg_sample %>%
2    select(1:4) %>%
3    full_join(y = asa_status, by = join_by(preOp_asa == asa_status_id))
```

```
# A tibble: 7 × 5
  preOp_gender preOp_asa preOp_calcBMI preOp_age asa_status
         <dbl>     <dbl>          <dbl>     <dbl> <chr>
1            0         2           31.3        80 a patient with mild systemic
d…
2            0         1           32.3        58 a normal healthy patient
3            1         3           30.5        65 a patient with severe
systemic…
4            0         3           32.2        54 a patient with severe
systemic…
5            1        NA           36.3        56 <NA>
6            0         2           28.0        68 a patient with mild systemic
d…
7           NA         4           NA          NA UNUSED CATEGORY
```

Note 7 rows: 5 with matches in both tables; one Table A patient with no pre-op ASA status reported; and one Table B unused

`inner_join()`: keep only rows from both tables

```
lg_sample %>%
  select(1:4) %>%
  inner_join(y = asa_status, by = join_by(preOp_asa == asa_status_id))
```

```
# A tibble: 5 × 5
  preOp_gender preOp_asa preOp_calcBMI preOp_age asa_status
         <dbl>     <dbl>          <dbl>     <dbl> <chr>
1            0         2           31.3        80 a patient with mild systemic
d…
2            0         1           32.3        58 a normal healthy patient
3            1         3           30.5        65 a patient with severe
systemic…
4            0         3           32.2        54 a patient with severe
systemic…
5            0         2           28.0        68 a patient with mild systemic
d…
```

Now we have only the five rows that appear in both tables.

`left_join()`: keep rows from x

```
1  lg_sample %>%
2    select(1:4) %>%
3    left_join(y = asa_status, by = join_by(preOp_asa == asa_status_id))
```

```
# A tibble: 6 × 5
  preOp_gender preOp_asa preOp_calcBMI preOp_age asa_status
         <dbl>     <dbl>          <dbl>     <dbl> <chr>
1            0         2           31.3        80 a patient with mild systemic
d…
2            0         1           32.3        58 a normal healthy patient
3            1         3           30.5        65 a patient with severe
systemic…
4            0         3           32.2        54 a patient with severe
systemic…
5            1        NA           36.3        56 <NA>
6            0         2           28.0        68 a patient with mild systemic
d…
```

Now we have increased to 6 rows, because we kept all of the left table, including the patient with no pre-op ASA status reported.

`right_join()`: keep rows from y

```
lg_sample %>%
  select(1:4) %>%
  right_join(y = asa_status, by = join_by(preOp_asa == asa_status_id))
```

```
# A tibble: 6 × 5
  preOp_gender preOp_asa preOp_calcBMI preOp_age asa_status
         <dbl>     <dbl>          <dbl>     <dbl> <chr>
1            0         2           31.3        80 a patient with mild systemic
d…
2            0         1           32.3        58 a normal healthy patient
3            1         3           30.5        65 a patient with severe
systemic…
4            0         3           32.2        54 a patient with severe
systemic…
5            0         2           28.0        68 a patient with mild systemic
d…
6           NA         4           NA          NA UNUSED CATEGORY
```

6 rows again, but this time we have dropped the nonreporting patient, and we see instead the unused category.

`semi_join()`, `anti_join()`: filter x vs. y

Filtering joins decide which rows to keep from x, based on the presence (or absence) of the value in y

Example: you applied inclusion criteria to get a list of patient ID's to include in analysis, and now you would like to filter just those patients' records into a data frame.

```
inclusion_ids <- tibble(subject_id = c(9134, 663, 5408))

covid_testing %>%
  semi_join(y = inclusion_ids, by = join_by(subject_id))
```

```
# A tibble: 3 × 17
  subject_id fake_…¹ fake_…² gender pan_day test_id clini…³ result demo_…⁴   age
       <dbl> <chr>   <chr>   <chr>    <dbl> <chr>   <chr>   <chr>  <chr>   <dbl>
1       9134 grunt   rivers  male         7 covid   clinic… negat… patient   0.8
2        663 ithoke  targar… male         9 covid   clinic… negat… patient   0.8
3       5408 alia    ryswell female      10 covid   clinic… negat… patient   0.9
# … with 7 more variables: drive_thru_ind <dbl>, ct_result <dbl>,
#   orderset <dbl>, payor_group <chr>, patient_class <chr>, col_rec_tat <dbl>,
#   rec_ver_tat <dbl>, and abbreviated variable names ¹fake_first_name,
#   ²fake_last_name, ³clinic_name, ⁴demo_group
```

ICA 4.3: Joins

Wrap up

Conclusion

We learned about:

- finding and visualizing missing data (NA)
- computing new columns in a data frame
- joining tables to combine data

Next time: working with factors; pivoting data