Data visualization

ggplot2 is the most popular data visualisation R package. Its `ggplot()` function is at the core of this package, and this whole approach is colloquially known as *"ggplot"* with the resulting figures sometimes affectionately called "ggplots". The "gg" in these names reflects the "**g**rammar of **g**raphics" used to construct the figures. **ggplot2** benefits from a wide variety of supplementary R packages that further enhance its functionality.

The data visualization with ggplot cheatsheet from the RStudio website is a great reference to have on-hand when creating pltos. If you want inspiration for ways to creatively visualise your data, we suggest reviewing websites like the R graph gallery and Data-to-viz.

1

Data Preparation

Import data

We import the dataset of cases from a simulated Ebola epidemic. If you want to follow along, click to download the "clean" linelist (as .rds file)..

The first 50 rows of the linelist are displayed below. We will focus on the continuous variables `age`, `wt_kg` (weight in kilos), `ct_blood` (CT values), and `days_onset_hosp` (difference between onset date and hospitalisation).

Show 5 ∨ entries   Search: [   ]

case_id ⬍	generation ⬍	date_infection ⬍	date_onset ⬍	date_hospitalisation ⬍	dat
[All]	[All]	[All]	[All]	[All]	[All
5fe599	4	2014-05-08	2014-05-13	2014-05-15	
8689b7	4		2014-05-13	2014-05-14	201
11f8ea	2		2014-05-16	2014-05-18	201
b8812a	3	2014-05-04	2014-05-18	2014-05-20	
893f25	3	2014-05-18	2014-05-21	2014-05-22	201

Showing 1 to 5 of 50 entries

Previous   [1]   2   3   4   5   …   10   Next

General cleaning

Some simple ways we can prepare our data to make it better for plotting can include making the contents of the data better for display - which does not necessarily equate to better for data manipulation. For example:

- Replace `NA` values in a character column with the character string "Unknown"

- Consider converting column to class *factor* so their values have prescribed ordinal levels

- Clean some columns so that their "data friendly" values with underscores etc are changed to normal text or title case.

Here are some examples of this in action:

```
# make display version of columns with more friendly names
linelist <- linelist %>%
  mutate(
    gender_disp = case_when(gender == "m" ~ "Male",         # m to Male
                            gender == "f" ~ "Female",       # f to Female,
                            is.na(gender) ~ "Unknown"),     # NA to Unknown

    outcome_disp = replace_na(outcome, "Unknown")           # replace NA outcome with "un
  )
```

Pivoting longer

As a matter of data structure, for **ggplot2** we often also want to pivot our data into *longer* formats. We will learn more about pivoting later; for now it's enough to be aware that these two data formats exist.

country	1999	2000	2001	2002
Angola	800	750	925	1020
India	20100	25650	26800	27255
Mongolia	450	512	510	586

Pivot data <u>longer</u>

```
data %>%
  pivot_longer(
    cols = 1999:2002,
    names_to = "year",
    values_to = "cases"
  )
```

country	year	cases
Angola	1999	800
Angola	2000	750
Angola	2001	925
Angola	2002	1020
India	1999	20100
India	2000	25650
India	2001	26800
India	2002	27255
Mongolia	1999	450
Mongolia	2000	512
Mongolia	2001	510
Mongolia	2002	586

For example, say that we want to plot data that are in a "wide" format, such as for each case in the `linelist` and their symptoms. Below we create a mini-linelist called `symptoms_data` that contains only the `case_id` and symptoms columns.

```
symptoms_data <- linelist %>%
  select(c(case_id, fever, chills, cough, aches, vomit))
```

Here is how the first 50 rows of this mini-linelist look - see how they are formatted "wide" with each symptom as a column:

```
   case_id fever chills cough aches vomit
1   5fe599    no     no   yes    no   yes
2   8689b7  <NA>   <NA>  <NA>  <NA>  <NA>
3   11f8ea  <NA>   <NA>  <NA>  <NA>  <NA>
4   b8812a    no     no    no    no    no
5   893f25    no     no   yes    no   yes
6   be99c8    no     no   yes    no   yes
7   07e3e8  <NA>   <NA>  <NA>  <NA>  <NA>
8   369449    no     no   yes    no   yes
9   f393b4    no     no   yes    no   yes
10  1389ca    no     no   yes    no    no
11  2978ac    no     no   yes    no   yes
12  57a565    no     no   yes    no    no
13  fc15ef    no     no   yes    no    no
14  2eaa9a    no     no   yes    no    no
15  bbfa93    no     no   yes    no   yes
```

```
16  c97dd9    no     no    yes   yes     no
17  f50e8a    no    yes    yes    no     no
18  3a7673    no     no    yes    no     no
19  7f5a01  <NA>   <NA>   <NA>  <NA>   <NA>
20  ddddee    no     no    yes    no     no
21  99e8fa    no     no    yes    no    yes
22  567136    no     no    yes    no     no
23  9371a9    no    yes    yes    no     no
24  bc2adf    no     no    yes    no     no
25  403057  <NA>   <NA>   <NA>  <NA>   <NA>
26  8bd1e8    no     no    yes    no     no
27  f327be    no     no    yes    no     no
28  42e1a9    no    yes    yes    no     no
29  90e5fe  <NA>   <NA>   <NA>  <NA>   <NA>
30  959170  <NA>   <NA>   <NA>  <NA>   <NA>
31  8ebf6e    no     no    yes    no     no
32  e56412    no     no    yes    no    yes
33  6d788e  <NA>   <NA>   <NA>  <NA>   <NA>
34  a47529    no     no    yes    no    yes
35  67be4e    no     no    yes    no    yes
36  da8ecb  <NA>   <NA>   <NA>  <NA>   <NA>
37  148f18  <NA>   <NA>   <NA>  <NA>   <NA>
38  2cb9a5    no     no    yes   yes    yes
39  f5c142    no     no    yes   yes    yes
40  70a9fe  <NA>   <NA>   <NA>  <NA>   <NA>
41  3ad520    no     no    yes    no    yes
42  062638    no     no    yes    no    yes
43  c76676  <NA>   <NA>   <NA>  <NA>   <NA>
44  baacc1    no     no    yes    no    yes
45  497372    no    yes    yes    no    yes
46  23e499    no     no    yes    no     no
47  38cc4a    no     no    yes    no    yes
48  3789ee    no     no    yes    no    yes
49  c71dcd    no     no     no    no    yes
50  6b70f0  <NA>   <NA>   <NA>  <NA>   <NA>
```

If we wanted to plot the number of cases with specific symptoms, we are limited by the fact that each symptom is a specific column. However, we can *pivot* the symptoms columns to a longer format like this:

```
symptoms_data_long <- symptoms_data %>%    # begin with "mini" linelist called symptoms_

  pivot_longer(
    cols = -case_id,                       # pivot all columns except case_id (all the s
    names_to = "symptom_name",             # assign name for new column that holds the s
    values_to = "symptom_is_present") %>%  # assign name for new column that holds the v

  mutate(symptom_is_present = replace_na(symptom_is_present, "unknown")) # convert NA to
```

Here are the first 50 rows. Note that case has 5 rows - one for each possible symptom. The
new columns `symptom_name` and `symptom_is_present` are the result of the pivot. Note
that this format may not be very useful for other operations, but is useful for plotting.

```
# A tibble: 50 x 3
   case_id symptom_name symptom_is_present
   <chr>   <chr>        <chr>
 1 5fe599  fever        no
 2 5fe599  chills       no
 3 5fe599  cough        yes
 4 5fe599  aches        no
 5 5fe599  vomit        yes
 6 8689b7  fever        unknown
 7 8689b7  chills       unknown
 8 8689b7  cough        unknown
 9 8689b7  aches        unknown
10 8689b7  vomit        unknown
# i 40 more rows
```

Basics of ggplot

"Grammar of graphics" - ggplot2

Plotting with **ggplot2** is based on "adding" plot layers and design elements on top of one
another, with each command added to the previous ones with a plus symbol (**+**). The result
is a multi-layer plot object that can be saved, modified, printed, exported, etc.

> The idea behind the Grammar of Graphics it is that you can build every graph
> from the same 3 components: (1) a data set, (2) a coordinate system, and (3)
> geoms — i.e. visual marks that represent data points [source]

ggplot objects can be highly complex, but the basic order of layers will usually look like this:

1. Begin with the baseline `ggplot()` command - this "opens" the ggplot and allow subsequent functions to be added with `+`. Typically the dataset is also specified in this command

2. Add "geom" layers - these functions visualize the data as *geometries* (*shapes*), e.g. as a bar graph, line plot, scatter plot, histogram (or a combination!). These functions all start with `geom_` as a prefix.

3. Add design elements to the plot such as axis labels, title, fonts, sizes, color schemes, legends, or axes rotation

In code this amounts to the basic template:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +  <GEOM_FUNCTION>()
```

We can further expand this template to include aspects of the visualization such as theme and labels:

```
# plot data from my_data columns as red points
ggplot(data = my_data)+                  # use the dataset "my_data"
  geom_point(                            # add a layer of points (dots)
    mapping = aes(x = col1, y = col2),   # "map" data column to axes
    color = "red")+                      # other specification for the geom
  labs()+                                # here you add titles, axes labels, etc.
  theme()                                # here you adjust color, font, size etc of non-d
```

We will explain each component in the sections below.

ggplot()

The opening command of any ggplot2 plot is `ggplot()`. This command simply creates a blank canvas upon which to add layers. It "opens" the way for further layers to be added with a `+` symbol.

Typically, the command `ggplot()` includes the `data =` argument for the plot. This sets the default dataset to be used for subsequent layers of the plot.

This command will end with a `+` after its closing parentheses. This leaves the command "open". The ggplot will only execute/appear when the full command includes a final layer *without* a `+` at the end.

```
# This will create plot that is a blank canvas
ggplot(data = linelist)
```

Geoms

A blank canvas is certainly not sufficient - we need to create geometries (shapes) from our data (e.g. bar plots, histograms, scatter plots, box plots).

This is done by adding layers "geoms" to the initial `ggplot()` command. There are many **ggplot2** functions that create "geoms". Each of these functions begins with "geom_", so we will refer to them generically as `geom_XXXX()`. There are over 40 geoms in **ggplot2** and many others created by fans. View them at the ggplot2 gallery. Some common geoms are listed below:

- Histograms - `geom_histogram()`

- Bar charts - `geom_bar()` or `geom_col()` (see "Bar plot" section)

- Box plots - `geom_boxplot()`

- Points (e.g. scatter plots) - `geom_point()`

- Line graphs - `geom_line()` or `geom_path()`

- Trend lines - `geom_smooth()`

In one plot you can display one or multiple geoms. Each is added to previous **ggplot2** commands with a `+`, and they are plotted sequentially such that later geoms are plotted on top of previous ones.

Mapping data to the plot

Most geom functions must be told *what to use* to create their shapes - so you must tell them how they should *map (assign) columns in your data* to components of the plot like the axes, shape colors, shape sizes, etc. For most geoms, the *essential* components that must be mapped to columns in the data are the x-axis, and (if necessary) the y-axis.

This "mapping" occurs with the `mapping =` argument. The mappings you provide to `mapping` must be wrapped in the `aes()` function, so you would write something like `mapping = aes(x = col1, y = col2)`, as shown below.

Below, in the `ggplot()` command the data are set as the case `linelist`. In the `mapping = aes()` argument the column `age` is mapped to the x-axis, and the column `wt_kg` is mapped to the y-axis.

After a `+`, the plotting commands continue. A shape is created with the "geom" function `geom_point()`. This geom *inherits* the mappings from the `ggplot()` command above - it

knows the axis-column assignments and proceeds to visualize those relationships as *points* on the canvas.

```
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point()
```
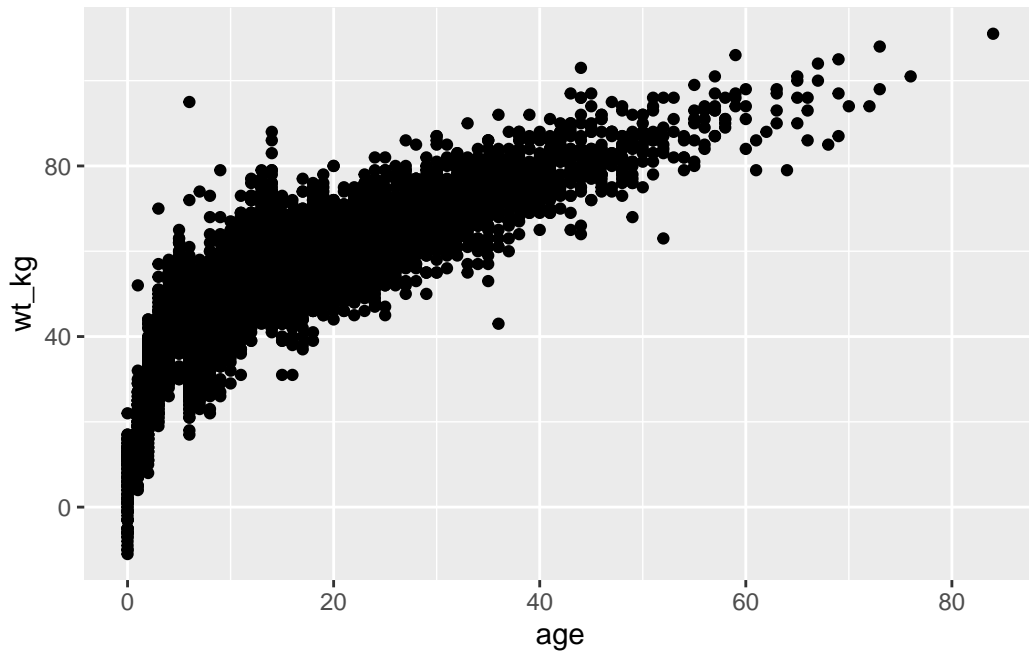

As another example, the following commands utilize the same data, a slightly different mapping, and a different geom. The `geom_histogram()` function only requires a column mapped to the x-axis, as the counts y-axis is generated automatically.

```
ggplot(data = linelist, mapping = aes(x = age))+
  geom_histogram()
```
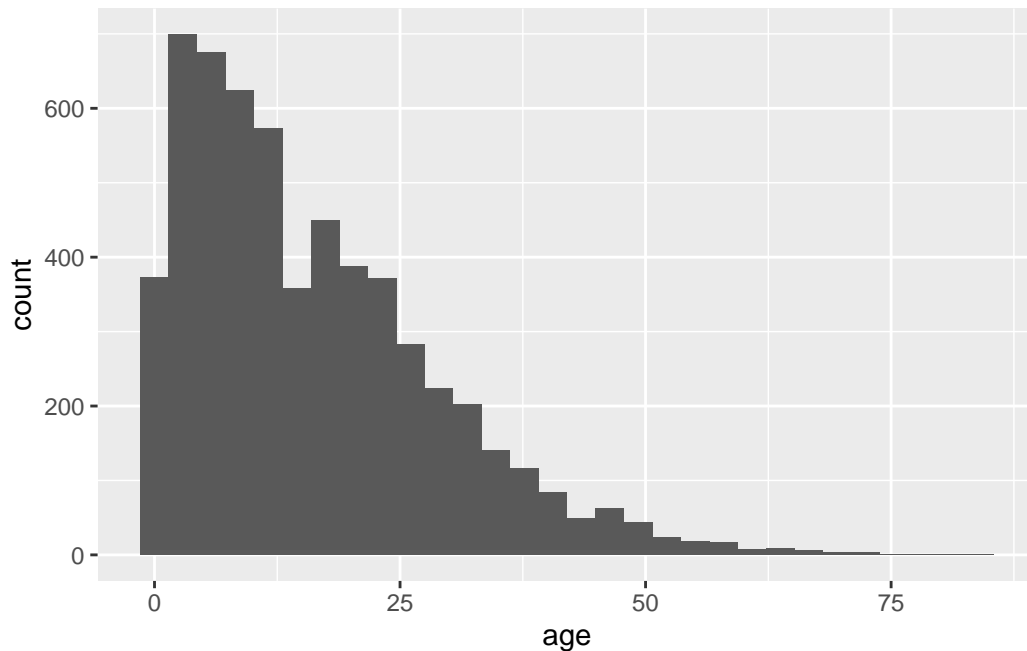
Plot aesthetics

In ggplot terminology a plot "aesthetic" has a specific meaning. It refers to a visual property of *plotted data*. Note that "aesthetic" here refers to the *data being plotted in geoms/shapes* - not the surrounding display such as titles, axis labels, background color, that you might associate with the word "aesthetics" in common English. In ggplot those details are called "themes" and are adjusted within a `theme()` command (see this section).

Therefore, plot object *aesthetics* can be colors, sizes, transparencies, placement, etc. *of the plotted data*. Not all geoms will have the same aesthetic options, but many can be used by most geoms. Here are some examples:

- `shape` = Display a point with `geom_point()` as a dot, star, triangle, or square...

- `fill` = The interior color (e.g. of a bar or boxplot)

- `color` = The exterior line of a bar, boxplot, etc., or the point color if using `geom_point()`

- `size` = Size (e.g. line thickness, point size)

- `alpha` = Transparency (1 = opaque, 0 = invisible)

11

- `binwidth =` Width of histogram bins

- `width =` Width of "bar plot" columns

- `linetype =` Line type (e.g. solid, dashed, dotted)

These plot object aesthetics can be assigned values in two ways:

1) Assigned a static value (e.g. `color = "blue"`) to apply across all plotted observations

2) Assigned to a column of the data (e.g. `color = hospital`) such that display of each observation depends on its value in that column

Set to a static value

If you want the plot object aesthetic to be static, that is - to be the same for every observation in the data, you write its assignment within the geom but *outside* of any `mapping = aes()` statement. These assignments could look like `size = 1` or `color = "blue"`. Here are two examples:

- In the first example, the `mapping = aes()` is in the `ggplot()` command and the axes are mapped to age and weight columns in the data. The plot aesthetics `color =`, `size =`, and `alpha =` (transparency) are assigned to static values. For clarity, this is done in the `geom_point()` function, as you may add other geoms afterward that would take different values for their plot aesthetics.

- In the second example, the histogram requires only the x-axis mapped to a column. The histogram `binwidth =`, `color =`, `fill =` (internal color), and `alpha =` are again set within the geom to static values.

```
# scatterplot
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+  # set data and axes mapping
  geom_point(color = "darkgreen", size = 0.5, alpha = 0.2)        # set static point aest

# histogram
ggplot(data = linelist, mapping = aes(x = age))+       # set data and axes
  geom_histogram(                   # display histogram
    binwidth = 7,                    # width of bins
    color = "red",                   # bin line color
    fill = "blue",                   # bin interior color
    alpha = 0.1)                     # bin transparency
```
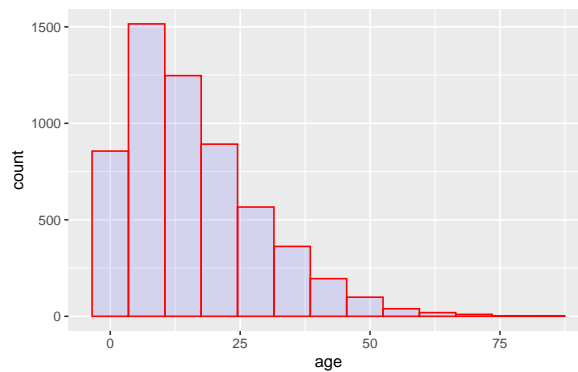
Scaled to column values

The alternative is to scale the plot object aesthetic by the values in a column. In this approach, the display of this aesthetic will depend on that observation's value in that column of the data. If the column values are continuous, the display scale (legend) for that aesthetic will be continuous. If the column values are discrete, the legend will display each value and the plotted data will appear as distinctly "grouped" (read more in the grouping section of this page).

To achieve this, you map that plot aesthetic to a *column name* (not in quotes). This must be done *within a `mapping = aes()` function* (note: there are several places in the code you can make these mapping assignments, as discussed below).

Two examples are below.

- In the first example, the `color =` aesthetic (of each point) is mapped to the column `age` - and a scale has appeared in a legend! For now just note that the scale exists - we will show how to modify it in later sections.

13

- In the second example two new plot aesthetics are also mapped to columns (`color =` and `size =`), while the plot aesthetics `shape =` and `alpha =` are mapped to static values outside of any `mapping = aes()` function.

```
# scatterplot
ggplot(data = linelist,   # set data
       mapping = aes(     # map aesthetics to column values
         x = age,              # map x-axis to age
         y = wt_kg,            # map y-axis to weight
         color = age)
       )+      # map color to age
  geom_point()          # display data as points


# scatterplot
ggplot(data = linelist,   # set data
       mapping = aes(     # map aesthetics to column values
         x = age,              # map x-axis to age
         y = wt_kg,            # map y-axis to weight
         color = age,          # map color to age
         size = age))+         # map size to age
  geom_point(               # display data as points
    shape = "diamond",       # points display as diamonds
    alpha = 0.3)             # point transparency at 30%
```


14
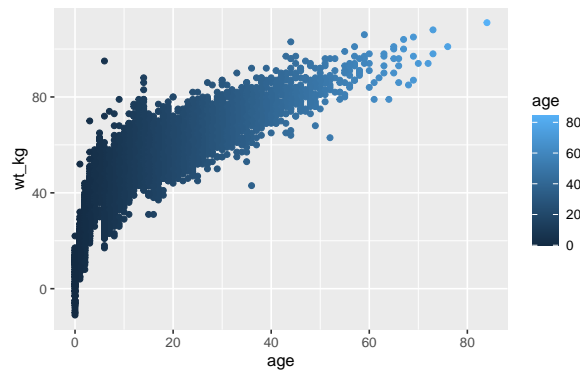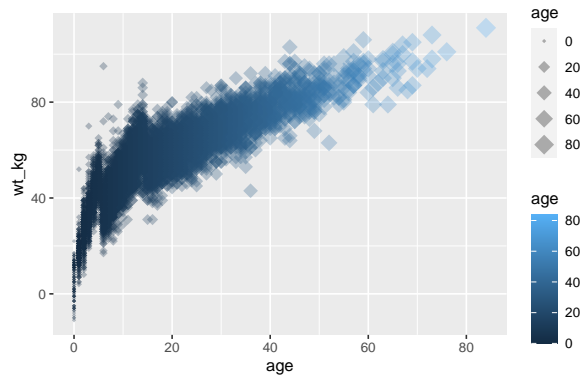
> **i Note**
>
> Axes assignments are always assigned to columns in the data (not to static values), and this is always done within `mapping = aes()`.

It becomes important to keep track of your plot layers and aesthetics when making more complex plots - for example plots with multiple geoms. In the example below, the `size =` aesthetic is assigned twice - once for `geom_point()` and once for `geom_smooth()` - both times as a static value.

```
ggplot(data = linelist,
       mapping = aes(               # map aesthetics to columns
         x = age,
         y = wt_kg,
         color = age_years)
       ) +
  geom_point(                       # add points for each row of data
    size = 1,
    alpha = 0.5) +
  geom_smooth(                      # add a trend line
    method = "lm",                  # with linear method
    size = 2)                       # size (width of line) of 2
```
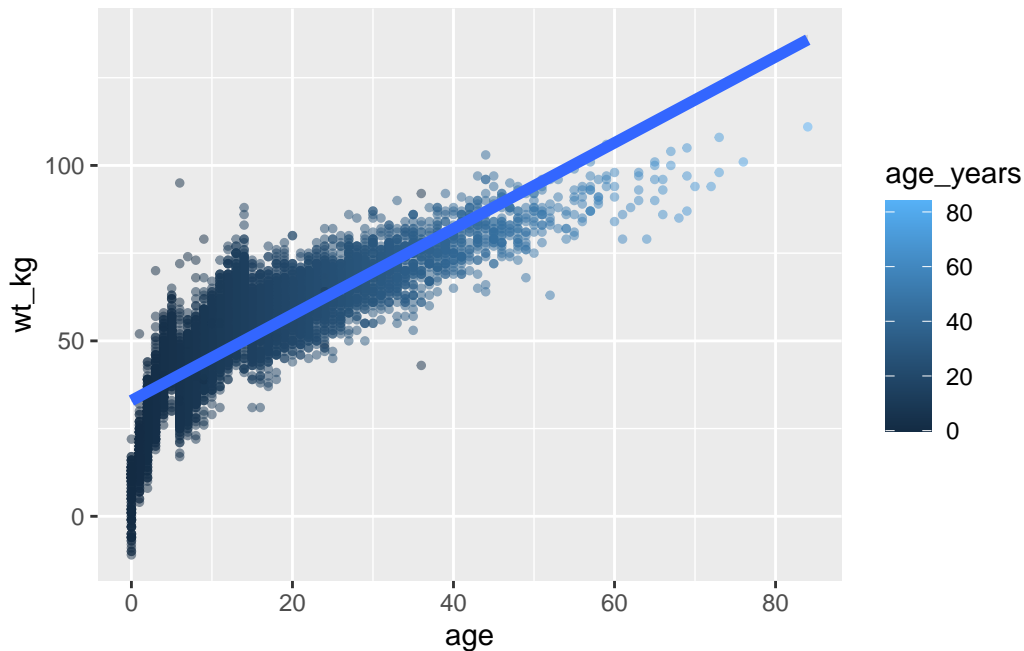
Where to make mapping assignments

Aesthetic mapping within `mapping = aes()` can be written in several places in your plotting commands and can even be written more than once. This can be written in the top `ggplot()` command, and/or for each individual geom beneath. The nuances include:

- Mapping assignments made in the top `ggplot()` command will be inherited as defaults across any geom below, like how `x =` and `y =` are inherited
- Mapping assignments made within one geom apply only to that geom

Likewise, `data =` specified in the top `ggplot()` will apply by default to any geom below, but you could also specify data for each geom (but this is more difficult).

Thus, each of the following commands will create the same plot:

```
# These commands will produce the exact same plot
ggplot(data = linelist, mapping = aes(x = age))+
  geom_histogram()

ggplot(data = linelist)+
  geom_histogram(mapping = aes(x = age))

ggplot()+
```

```
geom_histogram(data = linelist, mapping = aes(x = age))
```
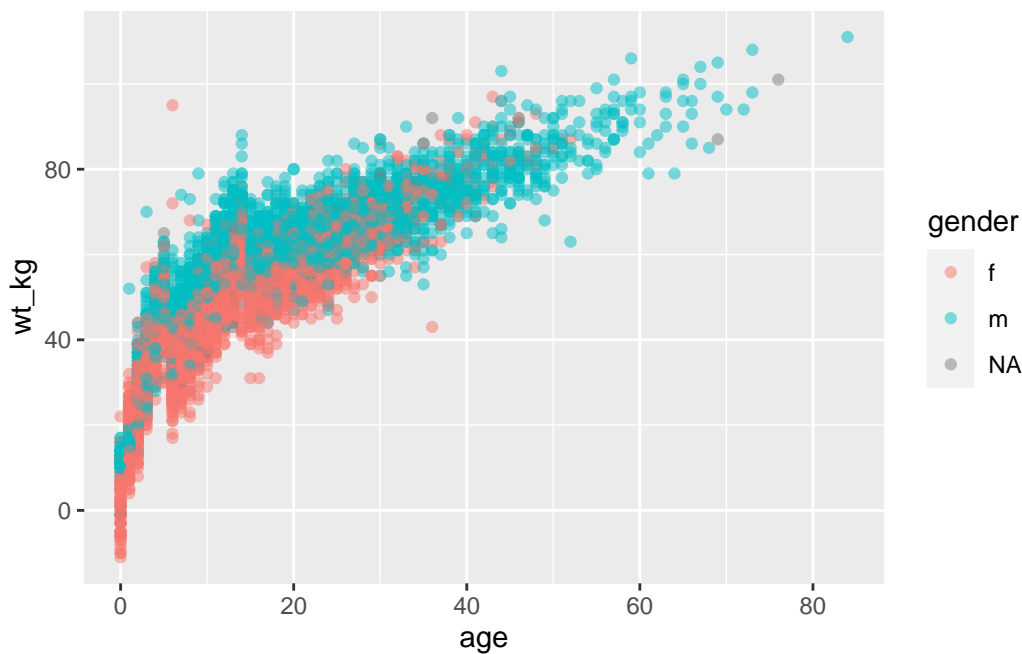
Groups

You can easily group the data and "plot by group". In fact, you have already done this!

Assign the "grouping" column to the appropriate plot aesthetic, within a `mapping = aes()`. Above, we demonstrated this using continuous values when we assigned point `size =` to the column `age`. However this works the same way for discrete/categorical columns.

For example, if you want points to be displayed by gender, you would set `mapping = aes(color = gender)`. A legend automatically appears. This assignment can be made within the `mapping = aes()` in the top `ggplot()` command (and be inherited by the geom), or it could be set in a separate `mapping = aes()` within the geom. Both approaches are shown below:

```
ggplot(data = linelist,
        mapping = aes(x = age, y = wt_kg, color = gender))+
    geom_point(alpha = 0.5)
```

```
# This alternative code produces the same plot
ggplot(data = linelist,
        mapping = aes(x = age, y = wt_kg))+
   geom_point(
      mapping = aes(color = gender),
      alpha = 0.5)
```

Note that depending on the geom, you will need to use different arguments to group the data. For `geom_point()` you will most likely use `color =`, `shape =` or `size =`. Whereas for `geom_bar()` you are more likely to use `fill =`. This just depends on the geom and what plot aesthetic you want to reflect the groupings.

For your information - the most basic way of grouping the data is by using only the `group =` argument within `mapping = aes()`. However, this by itself will not change the colors, fill, or shapes. Nor will it create a legend. Yet the data are grouped, so statistical displays may be affected.

Facets / Small-multiples

Facets, or "small-multiples", are used to split one plot into a multi-panel figure, with one panel ("facet") per group of data. The same type of plot is created multiple times, each one using a sub-group of the same dataset.

Faceting is a functionality that comes with **ggplot2**, so the legends and axes of the facet "panels" are automatically aligned. We would need to use other packages to combine completely different plots (**cowplot** and **patchwork**) into one figure.

Faceting is done with one of the following **ggplot2** functions:

1. `facet_wrap()` To show a different panel for each level of a *single* variable. One example of this could be showing a different epidemic curve for each hospital in a region. Facets are ordered alphabetically, unless the variable is a factor with other ordering defined.

   - You can invoke certain options to determine the layout of the facets, e.g. `nrow = 1` or `ncol = 1` to control the number of rows or columns that the faceted plots are arranged within.

2. `facet_grid()` This is used when you want to bring a second variable into the faceting arrangement. Here each panel of a grid shows the intersection between values in *two columns*. For example, epidemic curves for each hospital-age group combination with hospitals along the top (columns) and age groups along the sides (rows).

- **nrow** and **ncol** are not relevant, as the subgroups are presented in a grid

Each of these functions accept a formula syntax to specify the column(s) for faceting. Both accept up to two columns, one on each side of a tilde ~.

- For `facet_wrap()` most often you will write only one column preceded by a tilde ~ like `facet_wrap(~hospital)`. However you can write two columns `facet_wrap(outcome ~ hospital)` - each unique combination will display in a separate panel, but they will not be arranged in a grid. The headings will show combined terms and these won't be specific logic to the columns vs. rows. If you are providing only one faceting variable, a period . is used as a placeholder on the other side of the formula - see the code examples.

- For `facet_grid()` you can also specify one or two columns to the formula (grid `rows ~ columns`). If you only want to specify one, you can place a period . on the other side of the tilde like `facet_grid(. ~ hospital)` or `facet_grid(hospital ~ .)`.

Facets can quickly contain an overwhelming amount of information - its good to ensure you don't have too many levels of each variable that you choose to facet by. Here are some quick examples with the malaria dataset which consists of daily case counts of malaria for facilities, by age group.

Below we import and do some quick modifications for simplicity:

```
# These data are daily counts of malaria cases, by facility-day
malaria_data <- import("data/malaria_facility_count_data.rds") %>%  # import
    select(-submitted_date, -Province, -newid)                              # remove unne
```

The first 50 rows of the malaria data are below. Note there is a column `malaria_tot`, but also columns for counts by age group (these will be used in the second, `facet_grid()` example).

```
# A tibble: 50 x 7
   location_name data_date  District `malaria_rdt_0-4` `malaria_rdt_5-14`
   <chr>         <date>     <chr>                <int>              <int>
 1 Facility 1    2020-08-11 Spring                  11                 12
 2 Facility 2    2020-08-11 Bolo                    11                 10
 3 Facility 3    2020-08-11 Dingo                    8                  5
 4 Facility 4    2020-08-11 Bolo                    16                 16
 5 Facility 5    2020-08-11 Bolo                     9                  2
 6 Facility 6    2020-08-11 Dingo                    3                  1
 7 Facility 6    2020-08-10 Dingo                    4                  0
 8 Facility 5    2020-08-10 Bolo                    15                 14
 9 Facility 5    2020-08-09 Bolo                    11                 11
10 Facility 5    2020-08-08 Bolo                    19                 15
# i 40 more rows
# i 2 more variables: malaria_rdt_15 <int>, malaria_tot <int>
```

`facet_wrap()`

For the moment, let's focus on the columns `malaria_tot` and `District`. Ignore the age-specific count columns for now. We will plot epidemic curves with `geom_col()`, which produces a column for each day at the specified y-axis height given in column `malaria_tot` (the data are already daily counts, so we use `geom_col()` - see the "Bar plot" section below).

When we add the command `facet_wrap()`, we specify a tilde and then the column to facet on (`District` in this case). You can place another column on the left side of the tilde, - this will create one facet for each combination - but we recommend you do this with `facet_grid()` instead. In this use case, one facet is created for each unique value of `District`.

```
# A plot with facets by district
ggplot(malaria_data, aes(x = data_date, y = malaria_tot)) +
  geom_col(width = 1, fill = "darkred") +          # plot the count data as columns
  theme_minimal()+                                  # simplify the background panels
  labs(                                             # add plot labels, title, etc.
    x = "Date of report",
    y = "Malaria cases",
    title = "Malaria cases by district") +
  facet_wrap(~District)                             # the facets are created
```
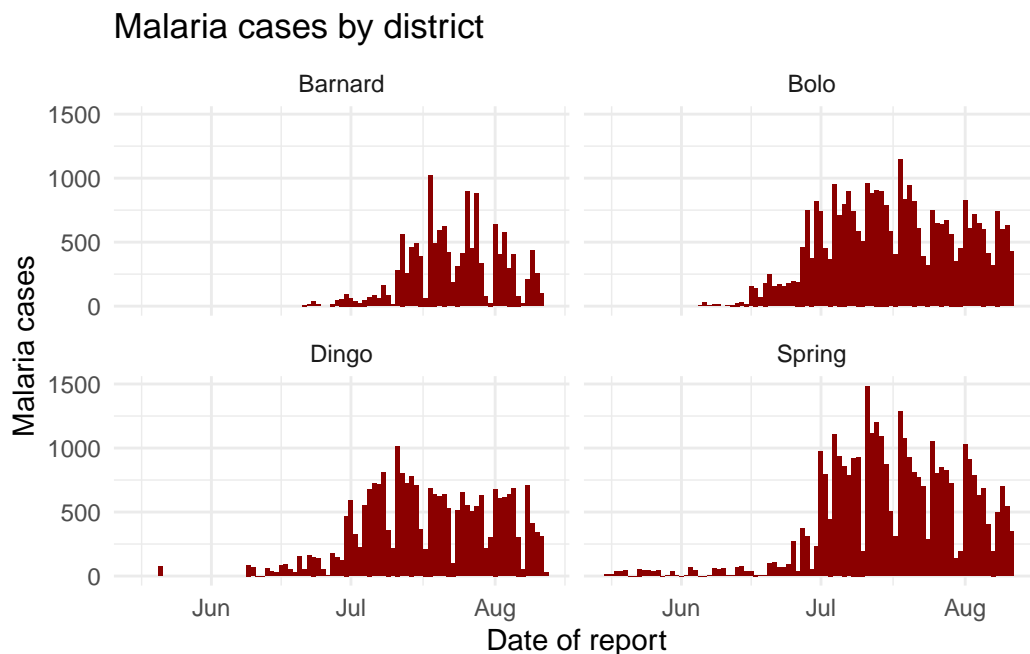
`facet_grid()`

We can use a `facet_grid()` approach to cross two variables. Let's say we want to cross `District` and age. Well, we need to do some data transformations on the age columns to get these data into ggplot-preferred "long" format. The age groups all have their own columns - we want them in a single column called `age_group` and another called `num_cases`.

```r
malaria_age <- malaria_data %>%
  select(-malaria_tot) %>%
  pivot_longer(
    cols = c(starts_with("malaria_rdt_")),  # choose columns to pivot longer
    names_to = "age_group",       # column names become age group
    values_to = "num_cases"       # values to a single column (num_cases)
  ) %>%
  mutate(
    age_group = str_replace(age_group, "malaria_rdt_", ""),
    age_group = forcats::fct_relevel(age_group, "5-14", after = 1))
```

Now the first 50 rows of data look like this:

```
# A tibble: 50 x 5
   location_name data_date  District age_group num_cases
   <chr>         <date>     <chr>    <fct>         <int>
 1 Facility 1    2020-08-11 Spring   0-4              11
 2 Facility 1    2020-08-11 Spring   5-14             12
 3 Facility 1    2020-08-11 Spring   15               23
 4 Facility 2    2020-08-11 Bolo     0-4              11
 5 Facility 2    2020-08-11 Bolo     5-14             10
 6 Facility 2    2020-08-11 Bolo     15                5
 7 Facility 3    2020-08-11 Dingo    0-4               8
 8 Facility 3    2020-08-11 Dingo    5-14              5
 9 Facility 3    2020-08-11 Dingo    15                5
10 Facility 4    2020-08-11 Bolo     0-4              16
# i 40 more rows
```

When you pass the two variables to `facet_grid()`, easiest is to use formula notation (e.g. `x ~ y`) where x is rows and y is columns. Here is the plot, using `facet_grid()` to show the plots for each combination of the columns `age_group` and `District`.

```r
ggplot(malaria_age, aes(x = data_date, y = num_cases)) +
  geom_col(fill = "darkred", width = 1) +
  theme_minimal()+
```

```
labs(
  x = "Date of report",
  y = "Malaria cases",
  title = "Malaria cases by district and age group"
) +
facet_grid(District ~ age_group)
```


Malaria cases by district and age group
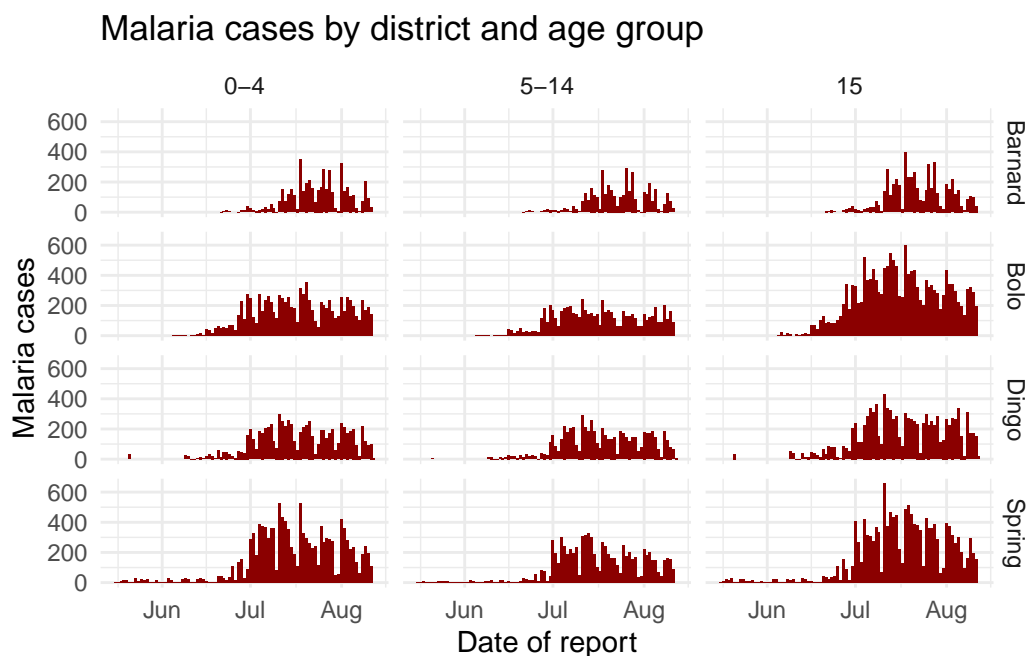
Free or fixed axes

The axes scales displayed when faceting are by default the same (fixed) across all the facets. This is helpful for cross-comparison, but not always appropriate.

When using `facet_wrap()` or `facet_grid()`, we can add `scales = "free_y"` to "free" or release the y-axes of the panels to scale appropriately to their data subset. This is particularly useful if the actual counts are small for one of the subcategories and trends are otherwise hard to see. Instead of "free_y" we can also write "free_x" to do the same for the x-axis (e.g. for dates) or "free" for both axes. Note that in `facet_grid`, the y scales will be the same for facets in the same row, and the x scales will be the same for facets in the same column.

When using `facet_grid` only, we can add `space = "free_y"` or `space = "free_x"` so that the actual height or width of the facet is weighted to the values of the figure within. This only works if `scales = "free"` (y or x) is already applied.

```
# Free y-axis
ggplot(malaria_data, aes(x = data_date, y = malaria_tot)) +
  geom_col(width = 1, fill = "darkred") +          # plot the count data as columns
  theme_minimal()+                                 # simplify the background panels
  labs(                                            # add plot labels, title, etc.
    x = "Date of report",
    y = "Malaria cases",
    title = "Malaria cases by district - 'free' x and y axes") +
  facet_wrap(~District, scales = "free")           # the facets are created
```


Malaria cases by district – 'free' x and y axes
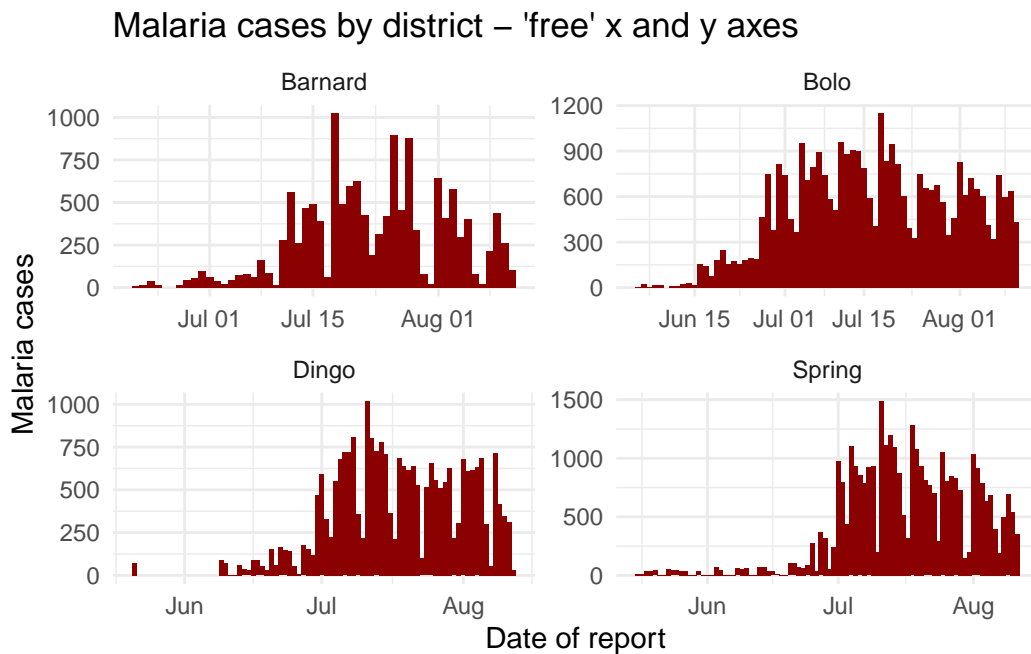
Exporting plots

Exporting ggplots is made easy with the **ggsave()** function from **ggplot2**. It can work in two ways, either:

- Specify the name of the plot object, then the file path and name with extension

    - For example: ggsave(my_plot, "plots/my_plot.png"))

- Run the command with only a file path, to save the last plot that was printed

    - For example: ggsave("plots/my_plot.png"))

23

You can export as png, pdf, jpeg, tiff, bmp, svg, or several other file types, by specifying the file extension in the file path.

You can also specify the arguments `width =`, `height =`, and `units =` (either "in", "cm", or "mm"). You can also specify `dpi =` with a number for plot resolution (e.g. 300). See the function details by entering `?ggsave` or reading the documentation online.

Labels

Surely you will want to add or adjust the plot's labels. These are most easily done within the `labs()` function which is added to the plot with `+` just as the geoms were.

Within `labs()` you can provide character strings to these arguements:

- `x =` and `y =` The x-axis and y-axis title (labels)

- `title =` The main plot title

- `subtitle =` The subtitle of the plot, in smaller text below the title

- `caption =` The caption of the plot, in bottom-right by default

Here is a plot we made earlier, but with nicer labels:

```
age_by_wt <- ggplot(
  data = linelist,   # set data
  mapping = aes(     # map aesthetics to column values
        x = age,           # map x-axis to age
        y = wt_kg,         # map y-axis to weight
        color = age))+     # map color to age
  geom_point()+          # display data as points
  labs(
    title = "Age and weight distribution",
    subtitle = "Fictional Ebola outbreak, 2014",
    x = "Age in years",
    y = "Weight in kilos",
    color = "Age",
    caption = stringr::str_glue("Data as of {max(linelist$date_hospitalisation, na.rm=T)}"

  age_by_wt
```

Age and weight distribution
Fictional Ebola outbreak, 2014

Data as of 2015−04−30
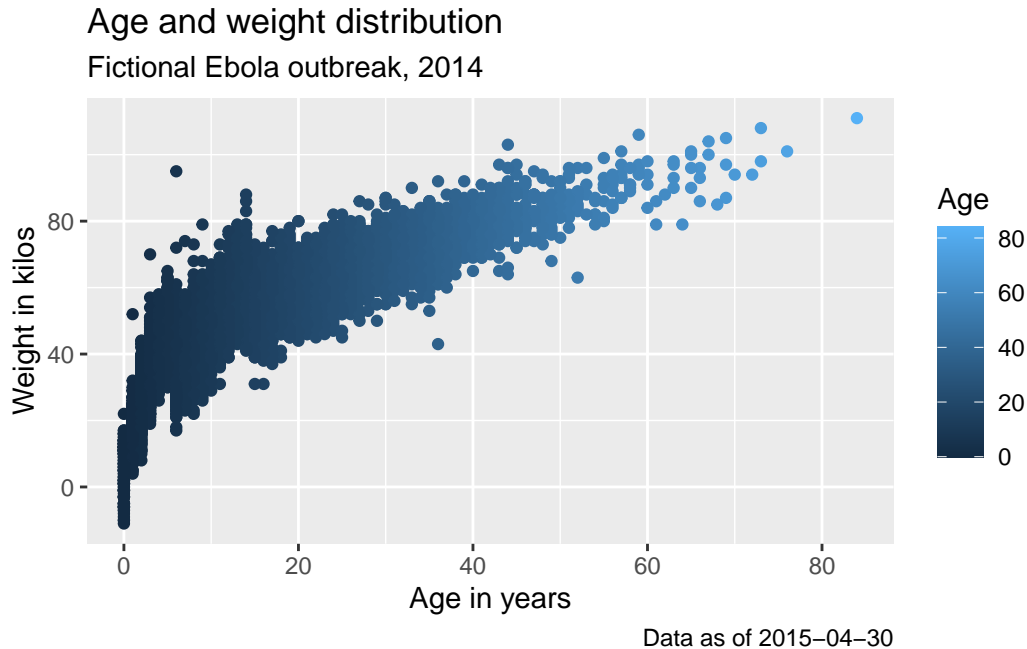
Note how in the caption assignment we used `str_glue()` from the **stringr** package to implant dynamic R code within the string text. The caption will show the "Data as of:" date that reflects the maximum hospitalization date in the linelist.

Plot continuous data

Throughout this page, you have already seen many examples of plotting continuous data. Here we briefly consolidate these and present a few variations.

Visualisations covered here include:

- Plots for one continuous variable:

  - **Histogram**, a classic graph to present the distribution of a continuous variable.
  - **Box plot** (also called box and whisker), to show the 25th, 50th, and 75th percentiles, tail ends of the distribution, and outliers (important limitations).

  - **Jitter plot**, to show all values as points that are 'jittered' so they can (mostly) all be seen, even where two have the same value.

  - **Violin plot**, show the distribution of a continuous variable based on the symmetrical width of the 'violin'.

– **Sina plot**, are a combination of jitter and violin plots, where individual points are shown but in the symmetrical shape of the distribution (via **ggforce** package).

- **Scatter plot** for two continuous variables.

- **Heat plots** for three continuous variables

Histograms

Histograms may look like bar charts, but are distinct because they measure the distribution of a *continuous* variable. There are no spaces between the "bars", and only one column is provided to `geom_histogram()`.

Below is code for generating **histograms**, which group continuous data into ranges and display in adjacent bars of varying height. This is done using `geom_histogram()`. See the "Bar plot" section of the ggplot basics page to understand difference between `geom_histogram()`, `geom_bar()`, and `geom_col()`.

We will show the distribution of ages of cases. Within `mapping = aes()` specify which column you want to see the distribution of. You can assign this column to either the x or the y axis.

The rows will be assigned to "bins" based on their numeric age, and these bins will be graphically represented by bars. If you specify a number of bins with the `bins =` plot aesthetic, the break points are evenly spaced between the minimum and maximum values of the histogram. If `bins =` is unspecified, an appropriate number of bins will be guessed and this message displayed after the plot:

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

If you do not want to specify a number of bins to `bins =`, you could alternatively specify `binwidth =` in the units of the axis. We give a few examples showing different bins and bin widths:

```
# A) Regular histogram
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram()+
  labs(title = "A) Default histogram (30 bins)")

# B) More bins
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram(bins = 50)+
  labs(title = "B) Set to 50 bins")
```

```
# C) Fewer bins
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram(bins = 5)+
  labs(title = "C) Set to 5 bins")


# D) More bins
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram(binwidth = 1)+
  labs(title = "D) binwidth of 1")
```

A) Default histogram (30 bins)

B) Set to 50 bins

C) Set to 5 bins

D) binwidth of 1
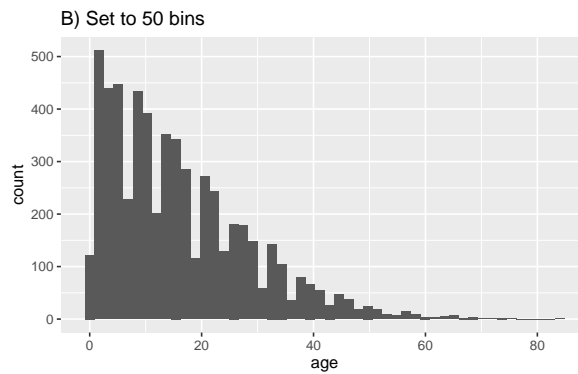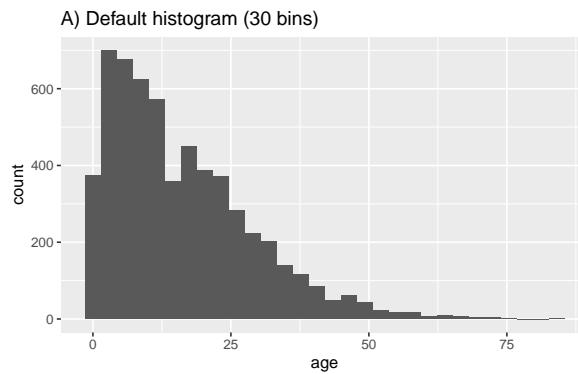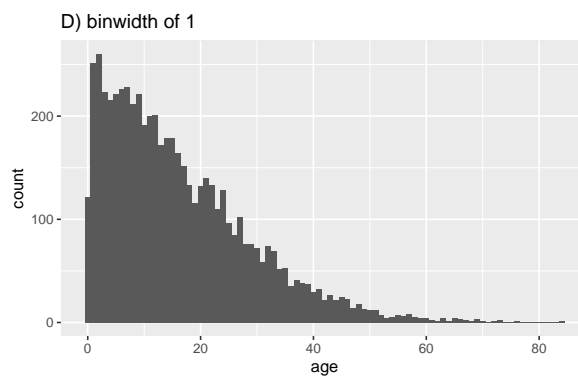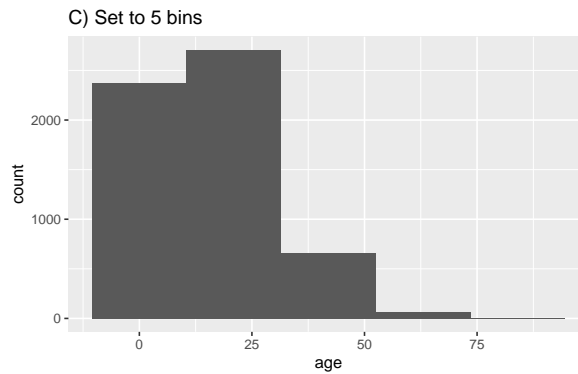
To get smoothed proportions, you can use `geom_density()`:

```
# Frequency with proportion axis, smoothed
ggplot(data = linelist, mapping = aes(x = age)) +
  geom_density(size = 2, alpha = 0.2)+
  labs(title = "Proportional density")

# Stacked frequency with proportion axis, smoothed
ggplot(data = linelist, mapping = aes(x = age, fill = gender)) +
  geom_density(size = 2, alpha = 0.2, position = "stack")+
  labs(title = "'Stacked' proportional densities")
```

Proportional density

'Stacked' proportional densities
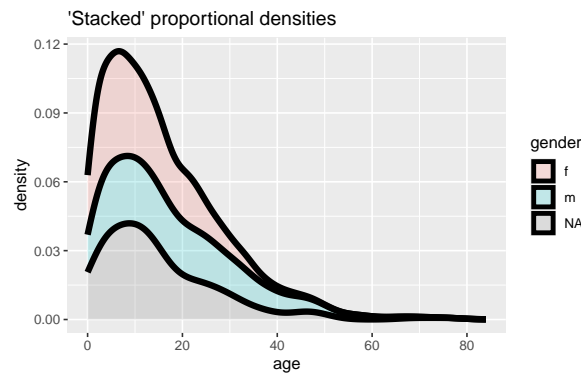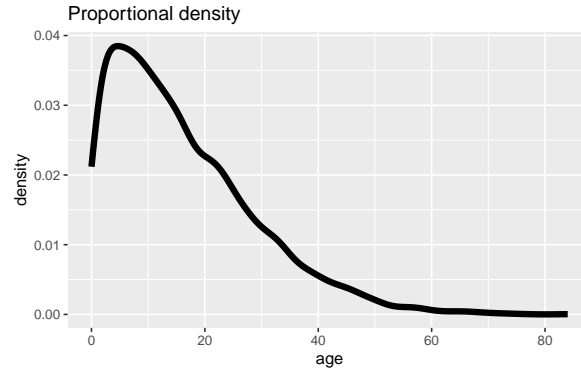
To get a "stacked" histogram (of a continuous column of data), you can do one of the following:

1) Use `geom_histogram()` with the `fill =` argument within `aes()` and assigned to the grouping column, or

2) Use `geom_freqpoly()`, which is likely easier to read (you can still set `binwidth =`)

3) To see proportions of all values, set the `y = after_stat(density)` (use this syntax exactly - not changed for your data). Note: these proportions will show *per group*.

Each is shown below (*note use of `color =` vs. `fill =` in each):

```
# "Stacked" histogram
ggplot(data = linelist, mapping = aes(x = age, fill = gender)) +
  geom_histogram(binwidth = 2)+
  labs(title = "'Stacked' histogram")

# Frequency
```

```
ggplot(data = linelist, mapping = aes(x = age, color = gender)) +
  geom_freqpoly(binwidth = 2, size = 2)+
  labs(title = "Freqpoly")

# Frequency with proportion axis
ggplot(data = linelist, mapping = aes(x = age, y = after_stat(density), color = gender)) +
  geom_freqpoly(binwidth = 5, size = 2)+
  labs(title = "Proportional freqpoly")

# Frequency with proportion axis, smoothed
ggplot(data = linelist, mapping = aes(x = age, y = after_stat(density), fill = gender)) +
  geom_density(size = 2, alpha = 0.2)+
  labs(title = "Proportional, smoothed with geom_density()")
```

Proportional freqpoly

Proportional, smoothed with geom_density()
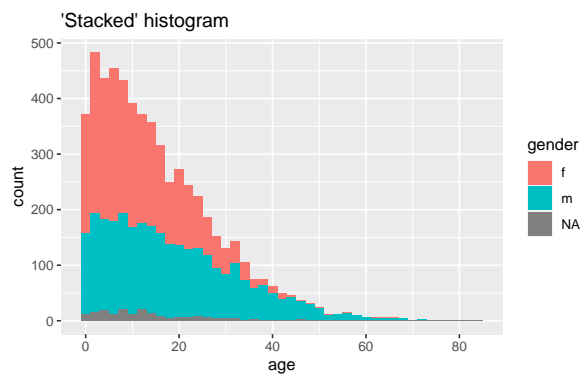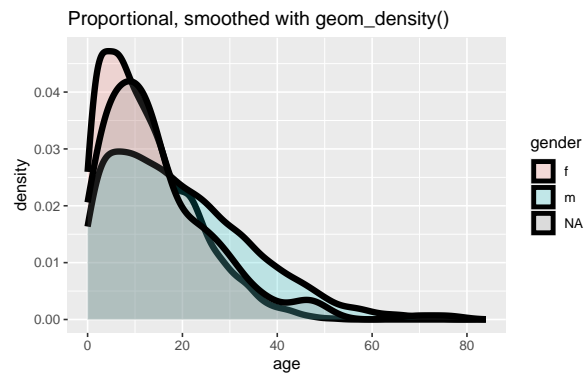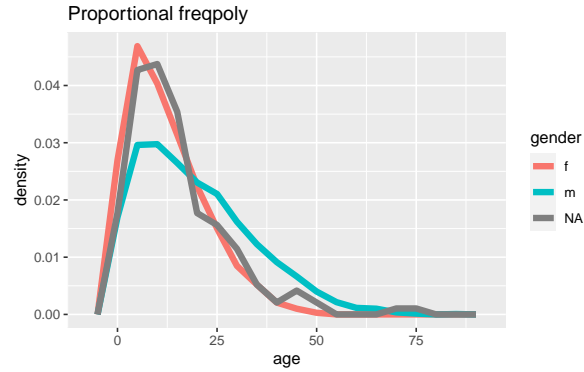
If you want to have some fun, try `geom_density_ridges` from the **ggridges** package (vignette here.

Read more in detail about histograms at the **tidyverse** page on geom_histogram().

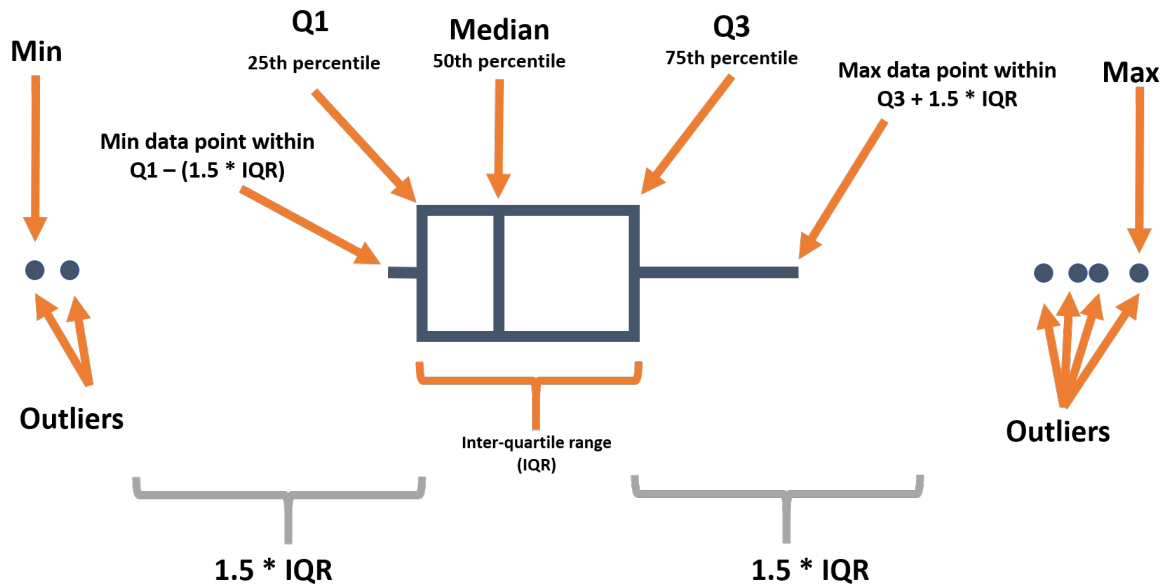### Box plots

Box plots are common, but have important limitations. They can obscure the actual distribution - e.g. a bi-modal distribution. See this R graph gallery and this data-to-viz article for more details. However, they do nicely display the inter-quartile range and outliers - so they can be overlaid on top of other types of plots that show the distribution in more detail.

Below we remind you of the various components of a boxplot:

When using `geom_boxplot()` to create a box plot, you generally map only one axis (x or y) within `aes()`. The axis specified determines if the plots are horizontal or vertical.

In most geoms, you create a plot per group by mapping an aesthetic like `color =` or `fill =` to a column within `aes()`. However, for box plots achieve this by assigning the grouping column to the un-assigned axis (x or y). Below is code for a boxplot of *all* age values in the dataset,

and second is code to display one box plot for each (non-missing) gender in the dataset. Note that `NA` (missing) values will appear as a separate box plot unless removed. In this example we also set the `fill` to the column `outcome` so each plot is a different color - but this is not necessary.

```
# A) Overall boxplot
ggplot(data = linelist)+
  geom_boxplot(mapping = aes(y = age))+   # only y axis mapped (not x)
  labs(title = "A) Overall boxplot")

# B) Box plot by group
ggplot(data = linelist, mapping = aes(y = age, x = gender, fill = gender)) +
  geom_boxplot()+
  theme(legend.position = "none")+   # remove legend (redundant)
  labs(title = "B) Boxplot by gender")
```

A) Overall boxplot

B) Boxplot by gender
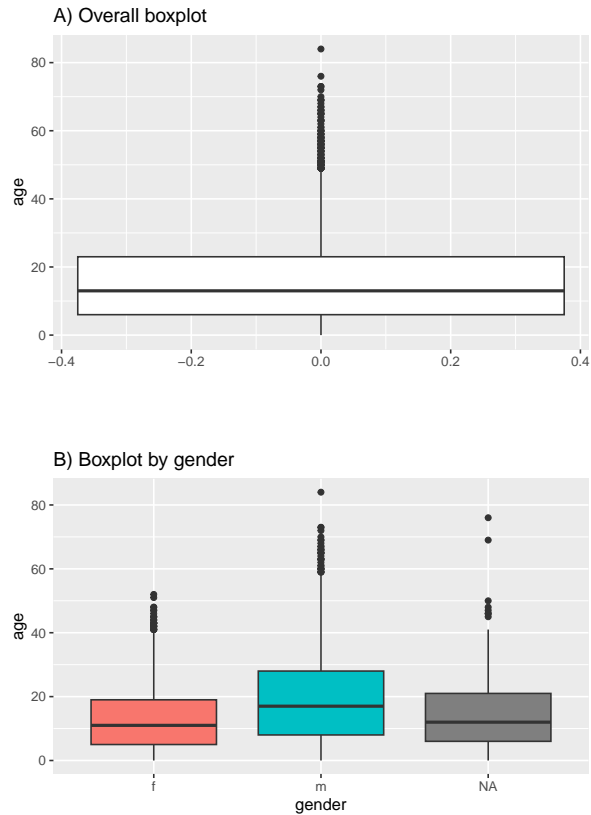
Violin, jitter, and sina plots

Below is code for creating **violin plots** (geom_violin) and **jitter plots** (geom_jitter) to show distributions. You can specify that the fill or color is also determined by the data, by inserting these options within aes().

```
# A) Jitter plot by group
ggplot(data = linelist %>% drop_na(outcome),    # remove missing values
       mapping = aes(y = age,                    # Continuous variable
           x = outcome,                          # Grouping variable
           color = outcome))+                    # Color variable
  geom_jitter()+                                 # Create the violin plot
  labs(title = "A) jitter plot by gender")



# B) Violin plot by group
```
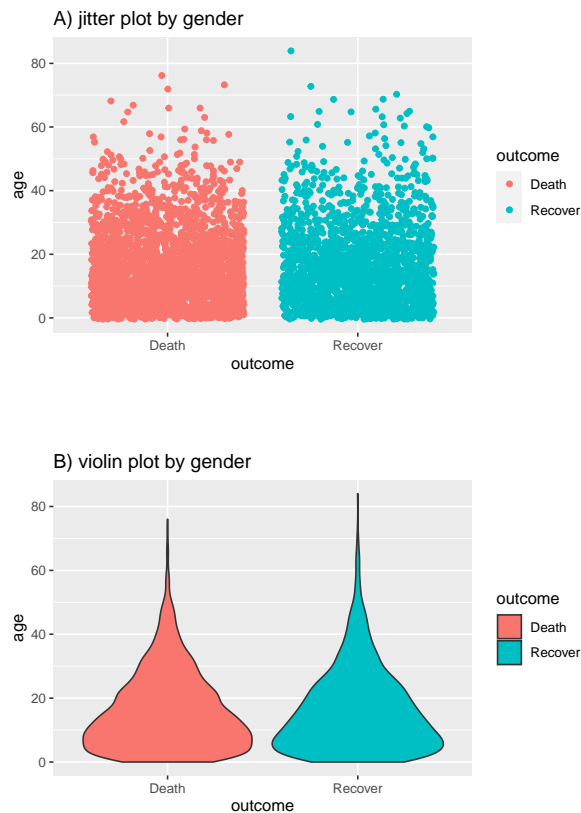
```
ggplot(data = linelist %>% drop_na(outcome),      # remove missing values
        mapping = aes(y = age,                     # Continuous variable
            x = outcome,                           # Grouping variable
            fill = outcome))+                      # fill variable (color)
  geom_violin()+                                   # create the violin plot
  labs(title = "B) violin plot by gender")
```


A) jitter plot by gender

B) violin plot by gender

You can combine the two using the **geom_sina()** function from the **ggforce** package. The sina plots the jitter points in the shape of the violin plot. When overlaid on the violin plot (adjusting the transparencies) this can be easier to visually interpret.

```
# A) Sina plot by group
ggplot(
  data = linelist %>% drop_na(outcome),
  aes(y = age,              # numeric variable
      x = outcome)) +       # group variable
  geom_violin(
```

```
  aes(fill = outcome), # fill (color of violin background)
  color = "white",     # white outline
  alpha = 0.2)+        # transparency
geom_sina(
  size=1,                   # Change the size of the jitter
  aes(color = outcome))+ # color (color of dots)
scale_fill_manual(       # Define fill for violin background by death/recover
  values = c("Death" = "#bf5300",
             "Recover" = "#11118c")) +
scale_color_manual(      # Define colours for points by death/recover
  values = c("Death" = "#bf5300",
             "Recover" = "#11118c")) +
theme_minimal() +                          # Remove the gray background
theme(legend.position = "none") +          # Remove unnecessary legend
labs(title = "B) violin and sina plot by gender, with extra formatting")
```

B) violin and sina plot by gender, with extra formatting
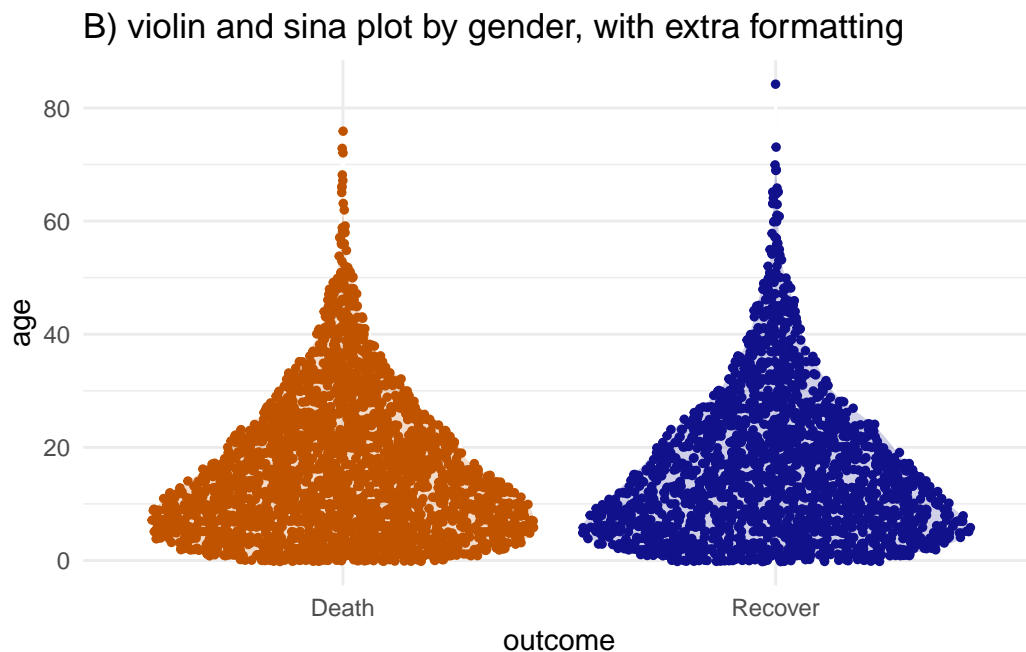
Two continuous variables

Following similar syntax, `geom_point()` will allow you to plot two continuous variables against each other in a **scatter plot**. This is useful for showing actual values rather than their distributions. A basic scatter plot of age vs weight is shown in (A). In (B) we again use

`facet_grid()` to show the relationship between two continuous variables in the linelist.

```
# Basic scatter plot of weight and age
ggplot(data = linelist,
       mapping = aes(y = wt_kg, x = age))+
  geom_point() +
  labs(title = "A) Scatter plot of weight and age")

# Scatter plot of weight and age by gender and Ebola outcome
ggplot(data = linelist %>% drop_na(gender, outcome), # filter retains non-missing gender/o
       mapping = aes(y = wt_kg, x = age))+
  geom_point() +
  labs(title = "B) Scatter plot of weight and age faceted by gender and outcome")+
  facet_grid(gender ~ outcome)
```


A) Scatter plot of weight and age

B) Scatter plot of weight and age faceted by gender and outcome
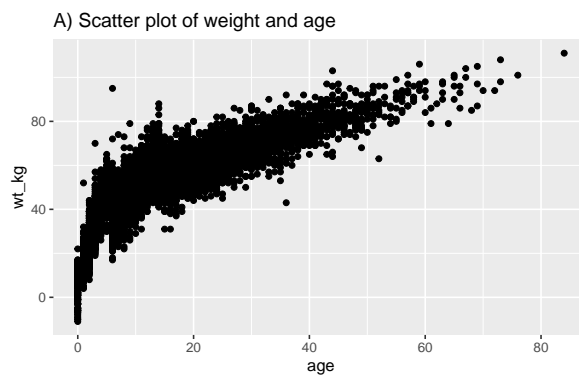
Three continuous variables

You can display three continuous variables by utilizing the `fill =` argument to create a *heat plot*. The color of each "cell" will reflect the value of the third continuous column of data. There are ways to make 3D plots in R, but for applied epidemiology these are often difficult to interpret and therefore less useful for decision-making.

Plot categorical data

Categorical data can be character values, could be logical (TRUE/FALSE), or factors.

Preparation

* Data structure

The first thing to understand about your categorical data is whether it exists as raw observations like a linelist of cases, or as a summary or aggregate data frame that holds counts or proportions. The state of your data will impact which plotting function you use:

- If your data are raw observations with one row per observation, you will likely use `geom_bar()`

- If your data are already aggregated into counts or proportions, you will likely use `geom_col()`

* Column class and value ordering

Next, examine the class of the columns you want to plot. We look at `hospital`, first with `class()` from **base** R, and with `tabyl()` from **janitor**.

```
# View class of hospital column - we can see it is a character
class(linelist$hospital)
```

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

```
# Look at values and proportions within hospital column
linelist %>%
  tabyl(hospital)
```

```
                     hospital    n    percent
             Central Hospital   454 0.07710598
            Military Hospital   896 0.15217391
                      Missing  1469 0.24949049
                        Other   885 0.15030571
                Port Hospital  1762 0.29925272
St. Mark's Maternity Hospital (SMMH)   422 0.07167120
```

We can see the values within are characters, as they are hospital names, and by default they
are ordered alphabetically. There are 'other' and 'missing' values, which we would prefer to
be the last subcategories when presenting breakdowns. So we change this column into a factor
and re-order it.

```
# Convert to factor and define level order so "Other" and "Missing" are last
linelist <- linelist %>%
  mutate(
    hospital = fct_relevel(hospital,
      "St. Mark's Maternity Hospital (SMMH)",
      "Port Hospital",
      "Central Hospital",
      "Military Hospital",
      "Other",
      "Missing"))

levels(linelist$hospital)
```

```
[1] "St. Mark's Maternity Hospital (SMMH)"
[2] "Port Hospital"
[3] "Central Hospital"
[4] "Military Hospital"
[5] "Other"
[6] "Missing"
```

geom_bar()

Use `geom_bar()` if you want bar height (or the height of stacked bar components) to reflect
the number of relevant rows in the data. These bars will have gaps between them, unless the
`width =` plot aesthetic is adjusted.

- Provide only one axis column assignment (typically x-axis). If you provide x and y, you will get `Error: stat_count() can only have an x or y aesthetic`.

- You can create stacked bars by adding a `fill =` column assignment within `mapping = aes()`

- The opposite axis will be titled "count" by default, because it represents the number of rows

Below, we have assigned outcome to the y-axis, but it could just as easily be on the x-axis. If you have longer character values, it can sometimes look better to flip the bars sideways and put the legend on the bottom. This may impact how your factor levels are ordered - in this case we reverse them with `fct_rev()` to put missing and other at the bottom.

```
# A) Outcomes in all cases
ggplot(linelist %>% drop_na(outcome)) +
  geom_bar(aes(y = fct_rev(hospital)), width = 0.7) +
  theme_minimal()+
  labs(title = "A) Number of cases by hospital",
       y = "Hospital")


# B) Outcomes in all cases by hosptial
ggplot(linelist %>% drop_na(outcome)) +
  geom_bar(aes(y = fct_rev(hospital), fill = outcome), width = 0.7) +
  theme_minimal()+
  theme(legend.position = "bottom") +
  labs(title = "B) Number of recovered and dead Ebola cases, by hospital",
       y = "Hospital")
```


A) Number of cases by hospital

B) Number of recovered and dead Ebo
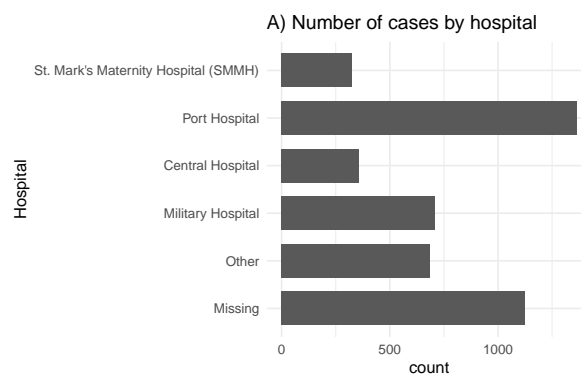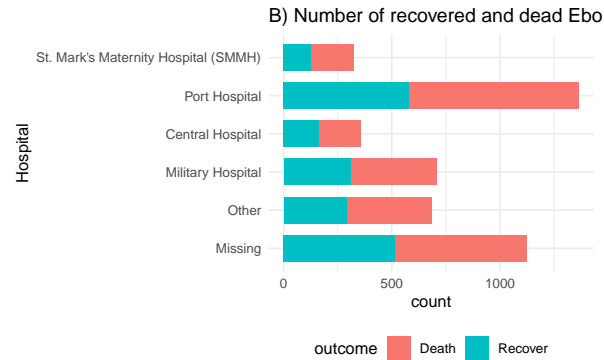
`geom_col()`

Use `geom_col()` if you want bar height (or height of stacked bar components) to reflect pre-calculated *values* that exists in the data. Often, these are summary or "aggregated" counts, or proportions.

Provide column assignments for *both* axes to `geom_col()`. Typically your x-axis column is discrete and your y-axis column is numeric.

Let's say we have this dataset `outcomes`:

```
# A tibble: 2 x 3
  outcome      n proportion
  <chr>    <int>      <dbl>
1 Death     1022       56.2
2 Recover    796       43.8
```

Below is code using `geom_col` for creating simple bar charts to show the distribution of Ebola patient outcomes. With geom_col, both x and y need to be specified. Here x is the categorical variable along the x axis, and y is the generated proportions column `proportion`.

```
# Outcomes in all cases
ggplot(outcomes) +
  geom_col(aes(x=outcome, y = proportion)) +
  labs(subtitle = "Number of recovered and dead Ebola cases")
```

Number of recovered and dead Ebola cases
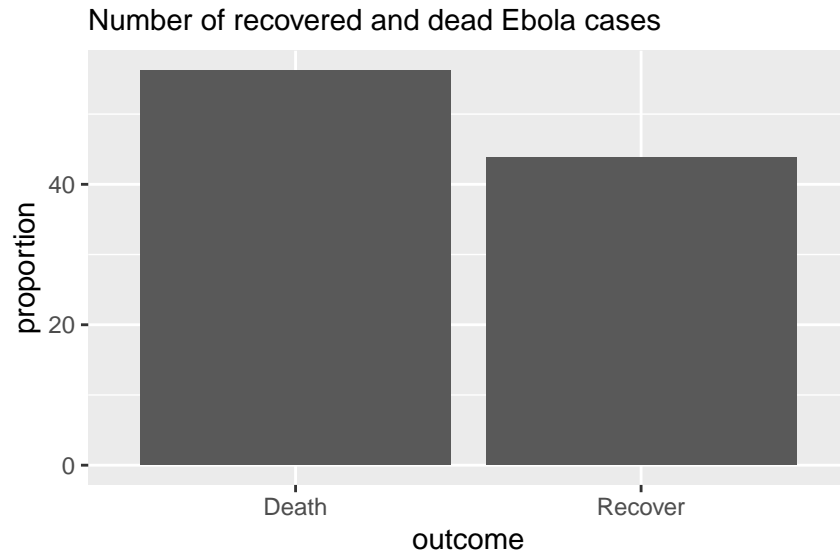
To show breakdowns by hospital, we would need our table to contain more information, and to be in "long" format. We create this table with the frequencies of the combined categories `outcome` and `hospital`.

```
outcomes2 <- linelist %>%
  drop_na(outcome) %>%
  count(hospital, outcome) %>%  # get counts by hospital and outcome
  group_by(hospital) %>%        # Group so proportions are out of hospital total
  mutate(proportion = n/sum(n)*100) # calculate proportions of hospital total

head(outcomes2) # Preview data
```

```
# A tibble: 6 x 4
# Groups:   hospital [3]
  hospital                          outcome     n proportion
  <fct>                             <chr>   <int>      <dbl>
1 St. Mark's Maternity Hospital (SMMH) Death   199       61.2
2 St. Mark's Maternity Hospital (SMMH) Recover 126       38.8
3 Port Hospital                     Death     785       57.6
4 Port Hospital                     Recover   579       42.4
5 Central Hospital                  Death     193       53.9
6 Central Hospital                  Recover   165       46.1
```

We then create the ggplot with some added formatting:

- **Axis flip**: Swapped the axis around with `coord_flip()` so that we can read the hospital names.
- **Columns side-by-side**: Added a `position = "dodge"` argument so that the bars for death and recover are presented side by side rather than stacked. Note stacked bars are the default.
- **Column width**: Specified 'width', so the columns are half as thin as the full possible width.
- **Column order**: Reversed the order of the categories on the y axis so that 'Other' and 'Missing' are at the bottom, with `scale_x_discrete(limits=rev)`. Note that we used that rather than `scale_y_discrete` because hospital is stated in the `x` argument of `aes()`, even if visually it is on the y axis. We do this because Ggplot seems to present categories backwards unless we tell it not to.

- **Other details**: Labels/titles and colours added within `labs` and `scale_fill_color` respectively.

```r
# Outcomes in all cases by hospital
ggplot(outcomes2) +
  geom_col(
    mapping = aes(
      x = proportion,                    # show pre-calculated proportion values
      y = fct_rev(hospital),             # reverse level order so missing/other at bottom
      fill = outcome),                   # stacked by outcome
    width = 0.5)+                        # thinner bars (out of 1)
  theme_minimal() +                      # Minimal theme
  theme(legend.position = "bottom")+
  labs(subtitle = "Number of recovered and dead Ebola cases, by hospital",
       fill = "Outcome",                 # legend title
       y = "Count",                      # y axis title
       x = "Hospital of admission")+     # x axis title
  scale_fill_manual(                     # adding colors manually
    values = c("Death"= "#3B1c8C",
               "Recover" = "#21908D" ))
```

Number of recovered and dead Ebola cases, by hospital
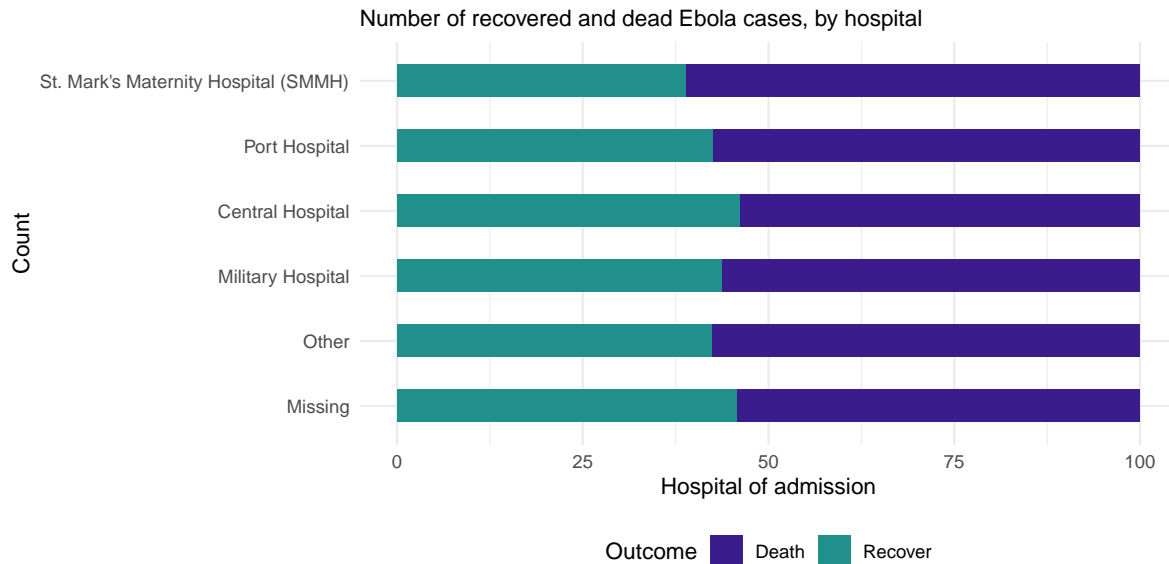
Note that the proportions are binary, so we may prefer to drop 'recover' and just show the proportion who died. This is just for illustration purposes.

If using `geom_col()` with dates data (e.g. an epicurve from aggregated data) - you will want to adjust the `width =` argument to remove the "gap" lines between the bars. If using daily data set `width = 1`. If weekly, `width = 7`. Months are not possible because each month has a different number of days.