CAGE, ggplot, and Tidyverse

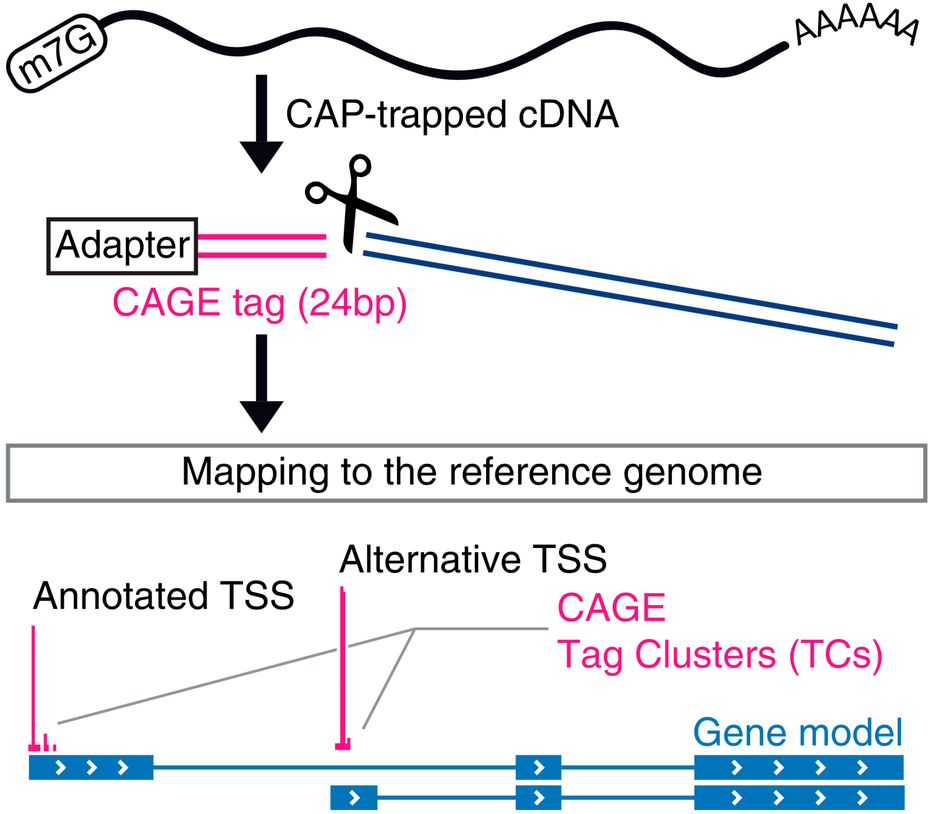
Axel Thieffry, PhD

October 2021

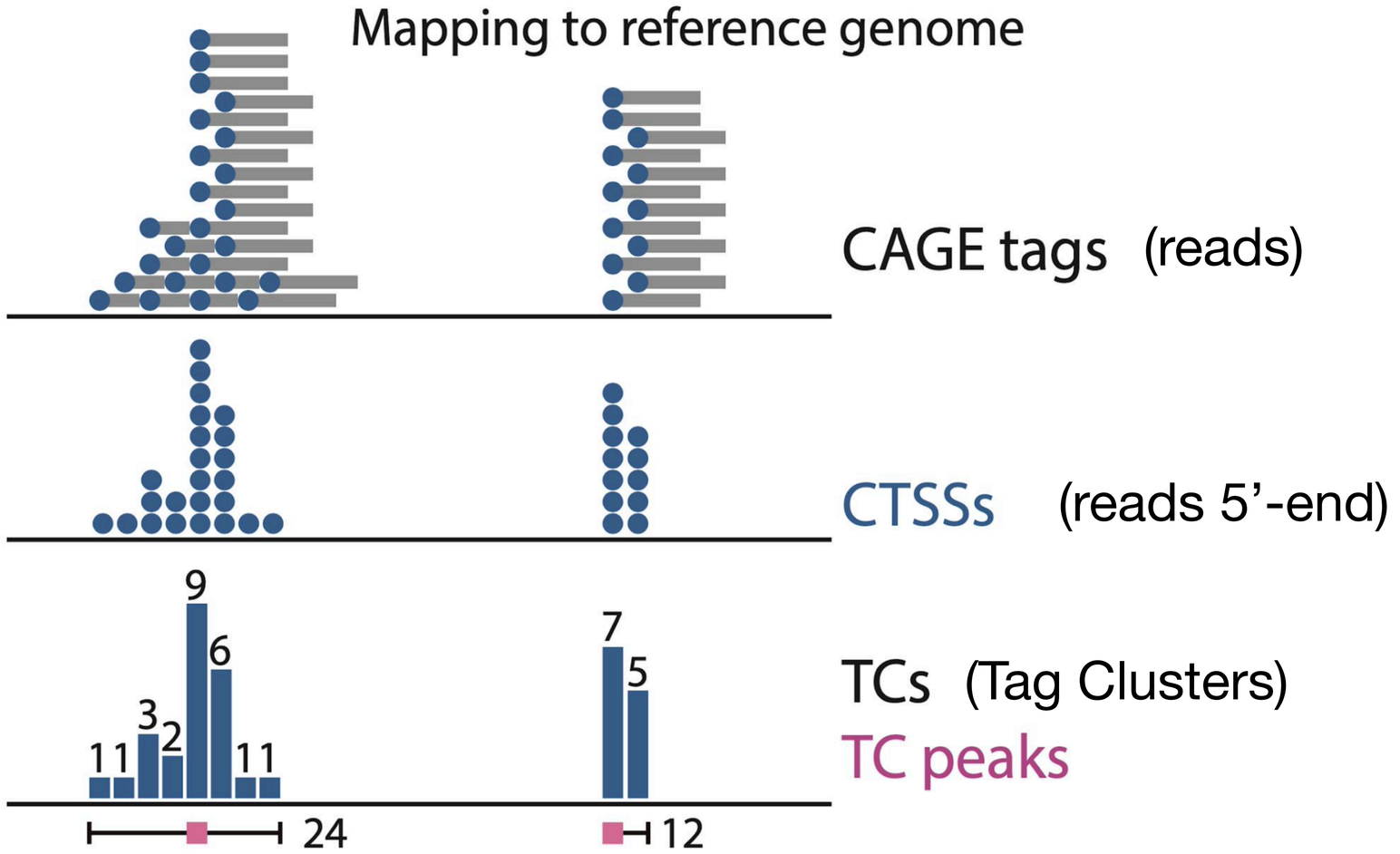
CAGE intro

Cap Analysis of Gene Expression

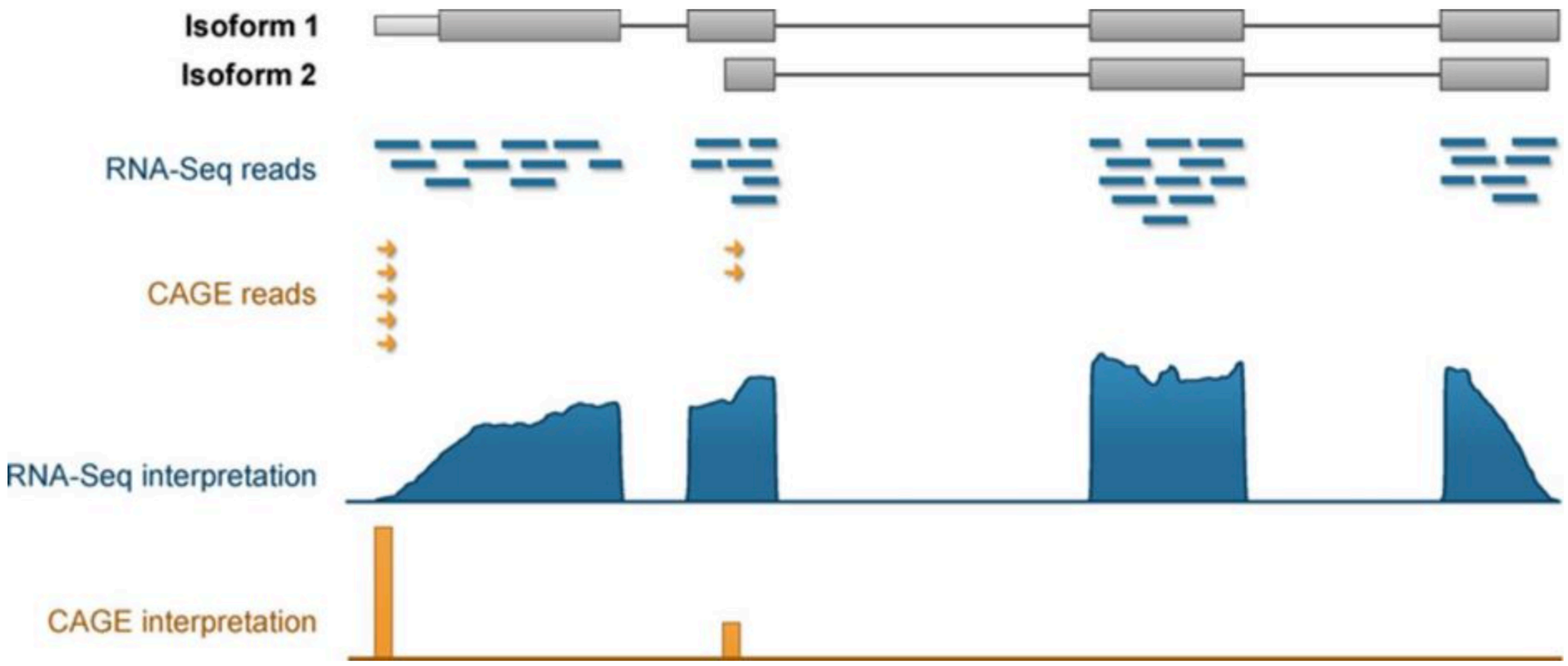
5’-end sequencing



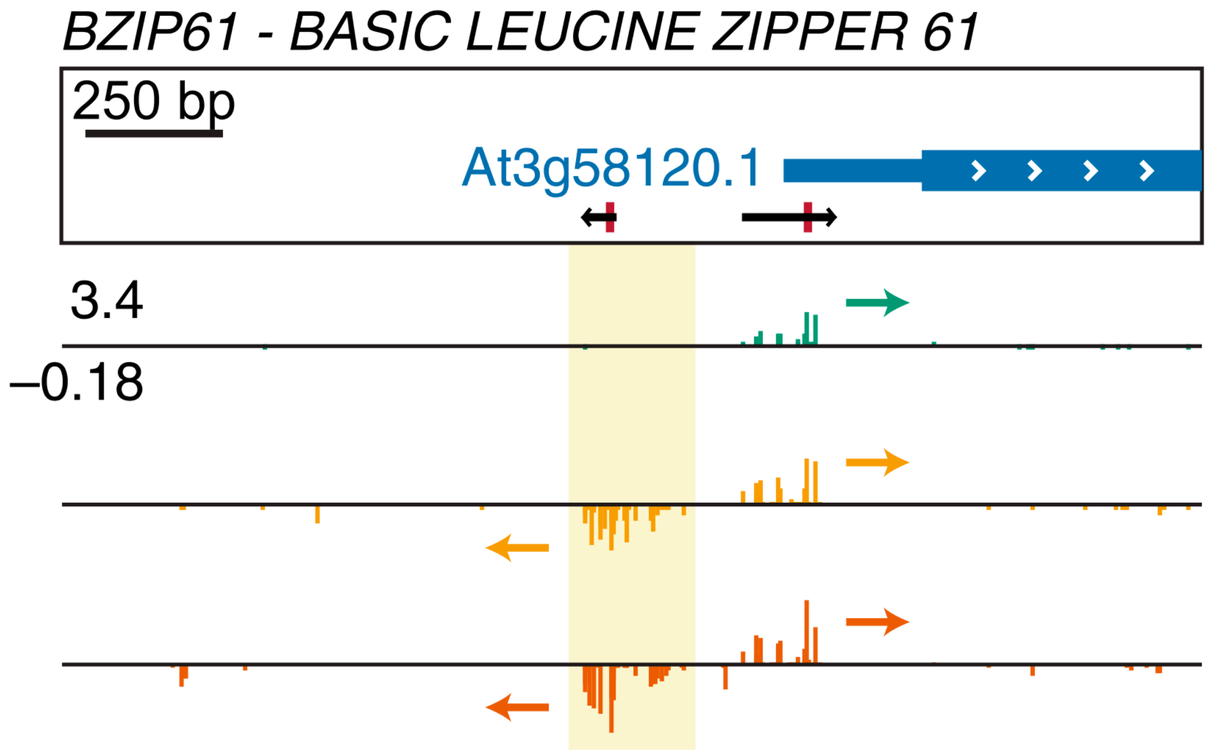
Conceptual CAGE pipeline



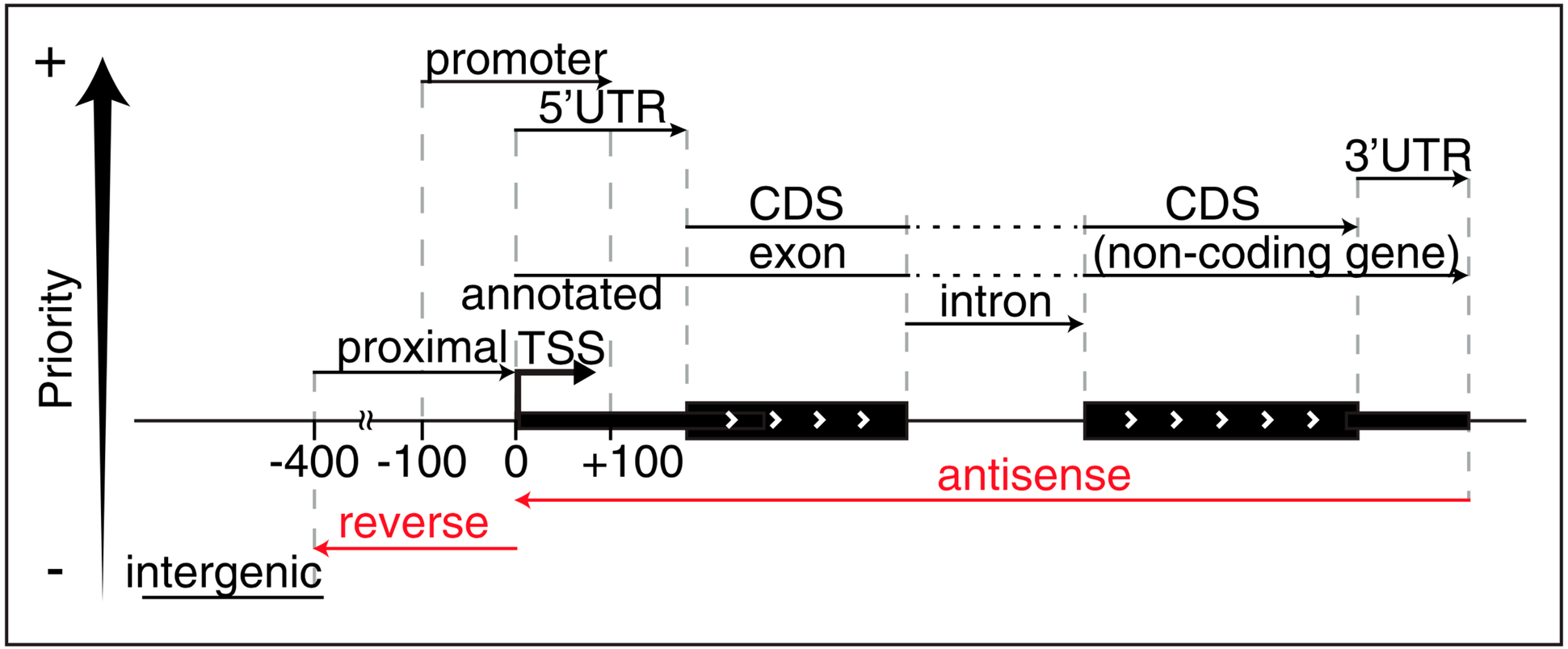
What CAGE looks like



What CAGE (really) looks like



CAGE annotation



Read the data!

=> https://github.com/athieffry/HEH_ATH_2021

- Download the file **CAGE_TC_counts.xlsx**, the matrix of CAGE raw counts
- Read the file in R using `read_xlsx()` from the package readxl
  - call it "**TCs**"
  - this might require to install **`readxl`**
  - good idea: have a quick look at the file first
- What data structure resulted from the `read_xlsx()` import?
- How many samples are there?
- How many CAGE TCs are there?

Sanity checks

- It's always a good idea to sanity check data you receive, whoever they come from!
- How many tags (reads) were mapped in each sample?
  - use `select()` to remove the `TC_id` and `colSums()`
  - use the `magrittr pipe`
- Do we have TCs with non attributed (`NA`) expression?
  - tip: `is.na()` tests whether a value is `NA`
- Try `sum(c(1:3, NA))`
  - What happened?
  - What do you deduce with regards to the 2 previous exercices?
  - Can you fix it? Tip: look at the Vignette of `sum()`
- Get an overview of the TCs expression distribution
  - Use `summary()`

ggplot2

Overview of ggplot2 geometries

GRAPHICAL PRIMITIVES

a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))

a + geom_blank() and **a + expand_limits()**
Ensure limits include values across all plots.

b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1**)** - x, xend, y, yend, alpha, angle, color, curvature, linetype, size

a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1**)**
x, y, alpha, color, group, linetype, size

a + geom_polygon(aes(alpha = 50)**)** - x, y, alpha, color, fill, group, subgroup, linetype, size

b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)**)** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)**)** - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept = 0, slope = 1)**)**
b + geom_hline(aes(yintercept = lat)**)**
b + geom_vline(aes(xintercept = long)**)**

b + geom_segment(aes(yend = lat + 1, xend = long + 1)**)**
b + geom_spoke(aes(angle = 1:1155, radius = 1)**)**

ONE VARIABLE continuous

c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

c + geom_area(stat = "bin"**)**
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian"**)**
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5**)**
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy)**)**
x, y, alpha, color, fill, linetype, size, weight

discrete

d <- ggplot(mpg, aes(fl))

d + geom_bar()
x, alpha, color, fill, linetype, size, weight

TWO VARIABLES

both continuous

e <- ggplot(mpg, aes(cty, hwy))

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1**)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e + geom_point()
x, y, alpha, color, fill, shape, size, stroke

e + geom_quantile()
x, y, alpha, color, group, linetype, size, weight

e + geom_rug(sides = "bl"**)**
x, y, alpha, color, linetype, size

e + geom_smooth(method = lm**)**
x, y, alpha, color, fill, group, linetype, size, weight

e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1**)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

continuous bivariate distribution

h <- ggplot(diamonds, aes(carat, price))

h + geom_bin2d(binwidth = c(0.25, 500)**)**
x, y, alpha, color, fill, linetype, size, weight

h + geom_density_2d()
x, y, alpha, color, group, linetype, size

h + geom_hex()
x, y, alpha, color, fill, size

continuous function

i <- ggplot(economics, aes(date, unemploy))

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

i + geom_step(direction = "hv"**)**
x, y, alpha, color, group, linetype, size

one discrete, one continuous

f <- ggplot(mpg, aes(class, hwy))

f + geom_col()
x, y, alpha, color, fill, group, linetype, size

f + geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f + geom_dotplot(binaxis = "y", stackdir = "center"**)**
x, y, alpha, color, fill, group

f + geom_violin(scale = "area"**)**
x, y, alpha, color, fill, group, linetype, size, weight

visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))

j + geom_crossbar(fatten = 2**)** - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width
Also **geom_errorbarh()**.

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

both discrete

g <- ggplot(diamonds, aes(cut, color))

g + geom_count()
x, y, alpha, color, fill, shape, size, stroke

e + geom_jitter(height = 2, width = 2**)**
x, y, alpha, color, fill, shape, size

maps

data <- data.frame(murder = USArrests$Murder,
state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))

k + geom_map(aes(map_id = state), map = map**)**
+ expand_limits(x = map$long, y = map$lat**)**
map_id, alpha, color, fill, linetype, size

THREE VARIABLES

seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))

l + geom_contour(aes(z = z)**)**
x, y, z, alpha, color, group, linetype, size, weight

l + geom_contour_filled(aes(fill = z)**)**
x, y, alpha, color, fill, group, linetype, size, subgroup

l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE**)**
x, y, alpha, fill

l + geom_tile(aes(fill = z)**)**
x, y, alpha, color, fill, linetype, size, width

Overview of ggplot2 geometries

Stats An alternative way to build a layer.

A stat builds new variables to plot (e.g., count, prop).

Visualize a stat by changing the default stat of a geom function, **geom_bar(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function). Use **..name..** syntax to map stat variables to aesthetics.

geom to use | stat function | geommappings

i + stat_density_2d(aes(fill = ..level..), geom = "polygon"**)**

variable created by stat

c + stat_bin(binwidth = 1, boundary = 10**)**
x, y | ..count.., ..ncount.., ..density.., ..ndensity..

c + stat_count(width = 1**) x, y** | ..count.., ..prop..

c + stat_density(adjust = 1, kernel = "gaussian"**)**
x, y | ..count.., ..density.., ..scaled..

e + stat_bin_2d(bins = 30, drop = T**)**
x, y, fill | ..count.., ..density..

e + stat_bin_hex(bins = 30**) x, y, fill** | ..count.., ..density..

e + stat_density_2d(contour = TRUE, n = 100**)**
x, y, color, size | ..level..

e + stat_ellipse(level = 0.95, segments = 51, type = "t"**)**

l + stat_contour(aes(z = z)**) x, y, z, order** | ..level..

l + stat_summary_hex(aes(z = z), bins = 30, fun = max**)**
x, y, z, fill | ..value..

l + stat_summary_2d(aes(z = z), bins = 30, fun = mean**)**
x, y, z, fill | ..value..

f + stat_boxplot(coef = 1.5**)**
x, y | ..lower.., ..middle.., ..upper.., ..width.. , ..ymin.., ..ymax..

f + stat_ydensity(kernel = "gaussian", scale = "area"**) x, y**
| ..density.., ..scaled.., ..count.., ..n.., ..violinwidth.., ..width..

e + stat_ecdf(n = 40**) x, y** | ..x.., ..y..

e + stat_quantile(quantiles = c(0.1, 0.9),
formula = y ~ log(x), method = "rq"**) x, y** | ..quantile..

e + stat_smooth(method = "lm", formula = y ~ x, se = T,
level = 0.95**) x, y** | ..se.., ..x.., ..y.., ..ymin.., ..ymax..

ggplot() + xlim(-5, 5**) + stat_function(**fun = dnorm,
n = 20, geom = "point"**) x** | ..x.., ..y..

ggplot() + stat_qq(aes(sample = 1:100)**)**
x, y, sample | ..sample.., ..theoretical..

e + stat_sum() x, y, size | ..n.., ..prop..

e + stat_summary(fun.data = "mean_cl_boot"**)**

h + stat_summary_bin(fun = "mean", geom = "bar"**)**

e + stat_identity()

e + stat_unique()

Scales Override defaults with **scales** package.

Scales map data values to the visual values of an aesthetic. To change a mapping, add a new scale.

n <- d + geom_bar(aes(fill = fl))

scale_ | aesthetic to adjust | prepackaged scale to use | scale-specific arguments

n + scale_fill_manual(
values = c("skyblue", "royalblue", "blue", "navy"),
limits = c("d", "e", "p", "r"), breaks =c("d", "e", "p", "r"),
name = "fuel", labels = c("D", "E", "P", "R"))

range of values to include in mapping | title to use in legend/axis | labels to use in legend/axis | breaks to use in legend/axis

GENERAL PURPOSE SCALES

Use with most aesthetics

scale_*_continuous() - Map cont' values to visual ones.
scale_*_discrete() - Map discrete values to visual ones.
scale_*_binned() - Map continuous values to discrete bins.
scale_*_identity() - Use data values as visual ones.
scale_*_manual(values = c()**)** - Map discrete values to manually chosen visual ones.
scale_*_date(date_labels = "%m/%d"**)**,
date_breaks = "2 weeks"**)** - Treat data values as dates.
scale_*_datetime() - Treat data values as date times. Same as scale_*_date(). See ?strptime for label formats.

X & Y LOCATION SCALES

Use with x or y aesthetics (x shown here)

scale_x_log10() - Plot x on log10 scale.
scale_x_reverse() - Reverse the direction of the x axis.
scale_x_sqrt() - Plot x on square root scale.

COLOR AND FILL SCALES (DISCRETE)

n + scale_fill_brewer(palette = "Blues"**)**
For palette choices:
RColorBrewer::display.brewer.all()

n + scale_fill_grey(start = 0.2,
end = 0.8, na.value = "red"**)**

COLOR AND FILL SCALES (CONTINUOUS)

o <- c + geom_dotplot(aes(fill = ..x..))

o + scale_fill_distiller(palette = "Blues"**)**

o + scale_fill_gradient(low="red", high="yellow"**)**

o + scale_fill_gradient2(low = "red", high = "blue",
mid = "white", midpoint = 25**)**

o + scale_fill_gradientn(colors = topo.colors(6)**)**
Also: rainbow(), heat.colors(), terrain.colors(),
cm.colors(), RColorBrewer::brewer.pal()

SHAPE AND SIZE SCALES

p <- e + geom_point(aes(shape = fl, size = cyl))

p + scale_shape() + scale_size()
p + scale_shape_manual(values = c(3:7)**)**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

p + scale_radius(range = c(1,6)**)**
p + scale_size_area(max_size = 6**)**

Coordinate Systems

r <- d + geom_bar()

r + coord_cartesian(xlim = c(0, 5)**)** - xlim, ylim
The default cartesian coordinate system.

r + coord_fixed(ratio = 1/2**)**
ratio, xlim, ylim - Cartesian coordinates with fixed aspect ratio between x and y units.

ggplot(mpg, aes(y = fl)**) + geom_bar()**
Flip cartesian coordinates by switching x and y aesthetic mappings.

r + coord_polar(theta = "x", direction=1**)**
theta, start, direction - Polar coordinates.

r + coord_trans(y = "sqrt"**)** - x, y, xlim, ylim
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

π + coord_quickmap()
π + coord_map(projection = "ortho", orientation = c(41, -74, 0)**)** - projection, xlim, ylim
Map projections from the mapproj package (mercator (default), azequalarea, lagrange, etc.).

Position Adjustments

Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

s <- ggplot(mpg, aes(fl, fill = drv))

s + geom_bar(position = "dodge")
Arrange elements side by side.

s + geom_bar(position = "fill")
Stack elements on top of one another, normalize height.

e + geom_point(position = "jitter")
Add random noise to X and Y position of each element to avoid overplotting.

e + geom_label(position = "nudge")
Nudge labels away from points.

s + geom_bar(position = "stack")
Stack elements on top of one another.

Each position adjustment can be recast as a function with manual **width** and **height** arguments:
s + geom_bar(position = position_dodge(width = 1))

Themes

r + theme_bw()
White background with grid lines.

r + theme_gray()
Grey background (default theme).

r + theme_dark()
Dark for contrast.

r + theme_classic()

r + theme_light()

r + theme_linedraw()

r + theme_minimal()
Minimal theme.

r + theme_void()
Empty theme.

r + theme() Customize aspects of the theme such as axis, legend, panel, and facet properties.
r + ggtitle("Title") + theme(plot.title.postion = "plot")
r + theme(panel.background = element_rect(fill = "blue"))

Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables.

t <- ggplot(mpg, aes(cty, hwy)) + geom_point()

t + facet_grid(cols = vars(fl))
Facet into columns based on fl.

t + facet_grid(rows = vars(year))
Facet into rows based on year.

t + facet_grid(rows = vars(year), cols = vars(fl))
Facet into both rows and columns.

t + facet_wrap(vars(fl))
Wrap facets into a rectangular layout.

Set **scales** to let axis limits vary across facets.

t + facet_grid(rows = vars(drv), cols = vars(fl),
scales = "free")
x and y axis limits adjust to individual facets:
"free_x" - x axis limits adjust
"free_y" - y axis limits adjust

Set **labeller** to adjust facet label:

t + facet_grid(cols = vars(fl), labeller = label_both)

fl: c	fl: d	fl: e	fl: p	fl: r

t + facet_grid(rows = vars(fl),
labeller = label_bquote(alpha ^ .(fl)**))**

α^c	α^d	α^e	α^p	α^r

Labels and Legends

Use **labs()** to label the elements of your plot.

t + labs(x = "New x axis label", **y** = "New y axis label",
title ="Add a title above the plot",
subtitle = "Add a subtitle below title",
caption = "Add a caption below plot",
alt = "Add alt text to the plot",
<AES> = "New **<AES>** legend title"**)**

t + annotate(geom = "text", x = 8, y = 9, label = "A"**)**
Places a geom with manually selected aesthetics.

p + guides(x = guide_axis(n.dodge = 2)**)** Avoid crowded or overlapping labels with guide_axis(n.dodge or angle).

n + guides(fill = "none"**)** Set legend type for each aesthetic: colorbar, legend, or none (no legend).

n + theme(legend.position = "bottom"**)**
Place legend at "bottom", "top", "left", or "right".

n + scale_fill_discrete(name = "Title",
labels = c("A", "B", "C", "D", "E"**)**
Set legend title and labels with a scale function.

Zooming

Without clipping (preferred):

t + coord_cartesian(xlim = c(0, 100), ylim = c(10, 20)**)**

With clipping (removes unseen data points):

t + xlim(0, 100**) + ylim(**10, 20**)**

t + scale_x_continuous(limits = c(0, 100)**) +**
scale_y_continuous(limits = c(0, 100)**)**

Our first plot!

- Looking at numbers in a table is a difficult mental exercise: a figure is needed!
- Reproduce the following code and try to dissect it:

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id) %>%
29      colSums() %>%
30      enframe() %>%
31      ggplot(aes(x=name, y=value)) +
32          geom_col() +
33          labs(x='sample', y='library size')
```

Tips: run it line by line and see the intermediate steps.

Let’s go through it.

- Here we use `select()` to remove (- sign) the column names `TC_id`

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id)
```

- The only reason for the pipe here is to better understand what is the intput once we have a larger block of code

Let’s go through it.

- Now that we only have numerical values left, we can compute the column-wise sums with `colSums()`

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id) %>%
29      colSums()
```

Let's go through it.

- `ggplot2` only takes data frames as input, but the output of `colSums()` is a vector

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id) %>%
29      colSums() %>%
30      enframe()
```

- The `enframe()` function nicely coerce the vector object into a data frame
- `as.data.frame()` could be used too, but the formatting is not as nice

Let's go through it.

- Now we pass the data frame to `ggplot()`

- `aes()` is where we define the plot-wide aesthetic and map variables to features of the plot. Here we map the X-axis to the sample names, and Y-axis to the value (those are default names of the `enframe()` function)

- Running `ggplot()` alone will just create a canvas, not geometric layer will be there

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id) %>%
29      colSums() %>%
30      enframe() %>%
31      ggplot(aes(x=name, y=value))
```

Let's go through it.

- To add any layer to a `ggplot()` figure, we use '+'
- Here we use a column geometry by using `geom_col()`
- `geom_col()` will **inherit** the aesthetics that we declared in the main call of `ggplot()`

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id) %>%
29      colSums() %>%
30      enframe() %>%
31      ggplot(aes(x=name, y=value)) +
32          geom_col()
```

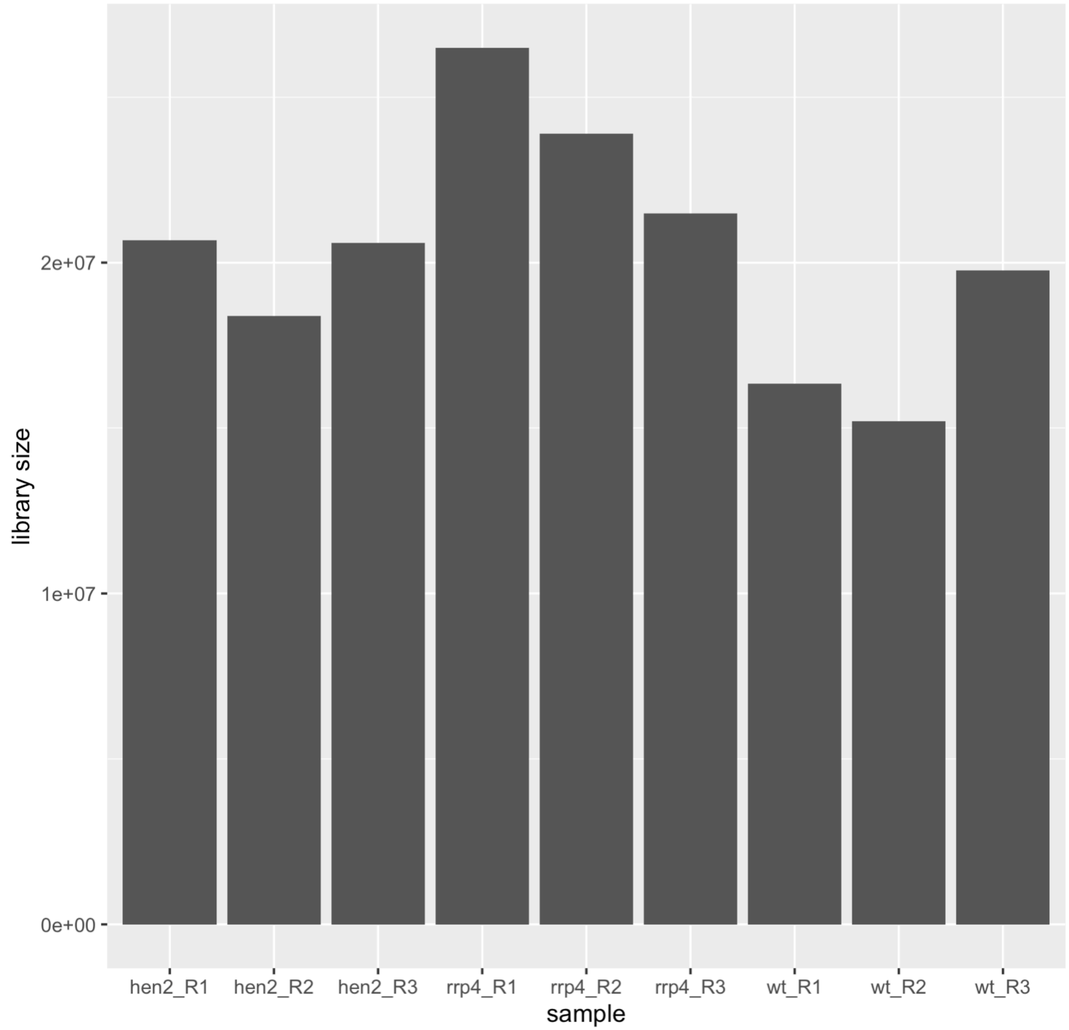
Let's go through it.

- Let's rename the axis labels to better fit the data
- We add another layer (+) called `labs()`
- Other parameters of `labs()` are `title`, `subtitle`, `caption`, `x`, `y`, ...

```
26  # c. our first plot!
27  TCs %>%
28      select(-TC_id) %>%
29      colSums() %>%
30      enframe() %>%
31      ggplot(aes(x=name, y=value)) +
32          geom_col() +
33          labs(x='sample', y='library size')
```

Let’s go through it.

- Et voilà.

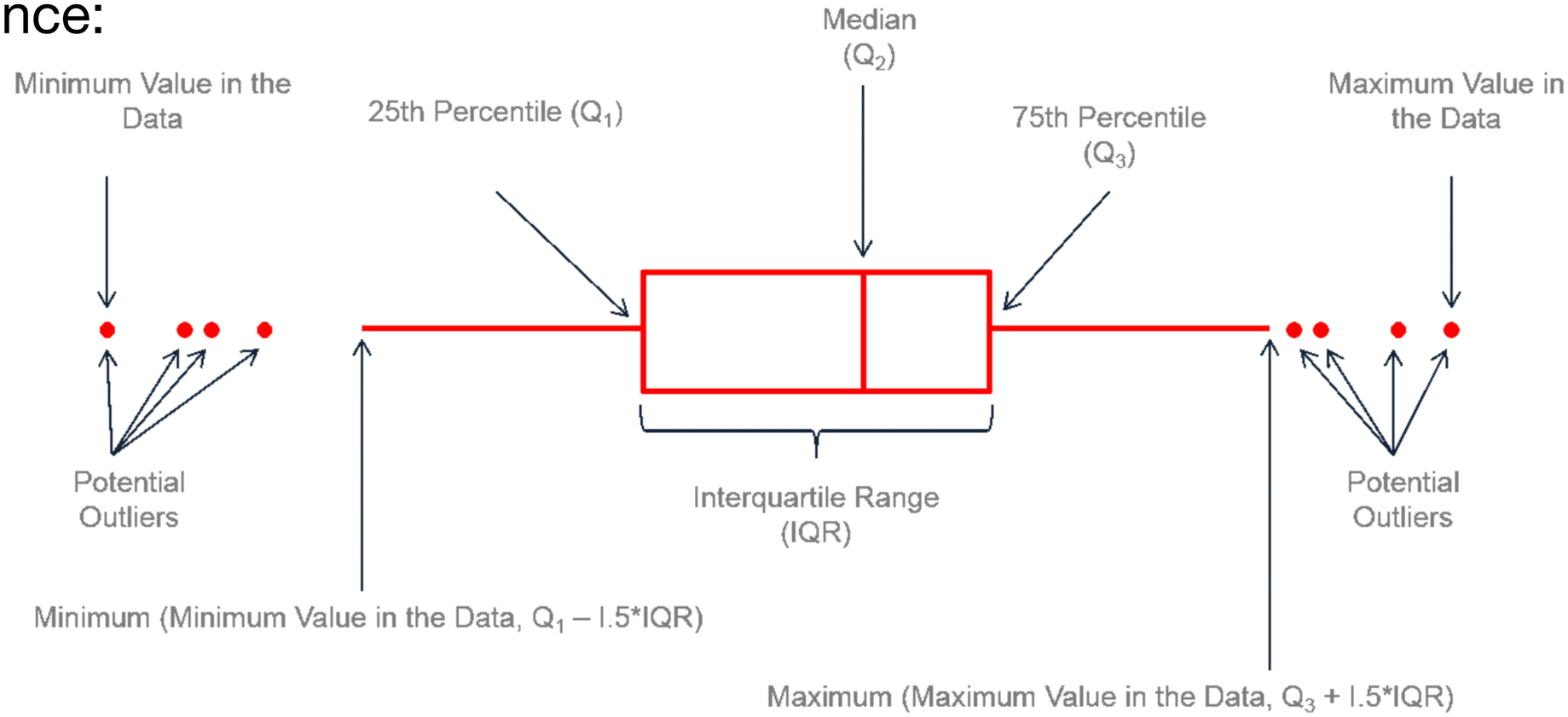


- `ggplot()` is a big library
- Many, many stuff are possible and customisable
- It relies on the **g**rammar of **g**raphics
- MUST HAVE:
  ggplot2 cheat sheet

👉 www.rstudio.com/resources/cheatsheets/

Expression distribution: box plot

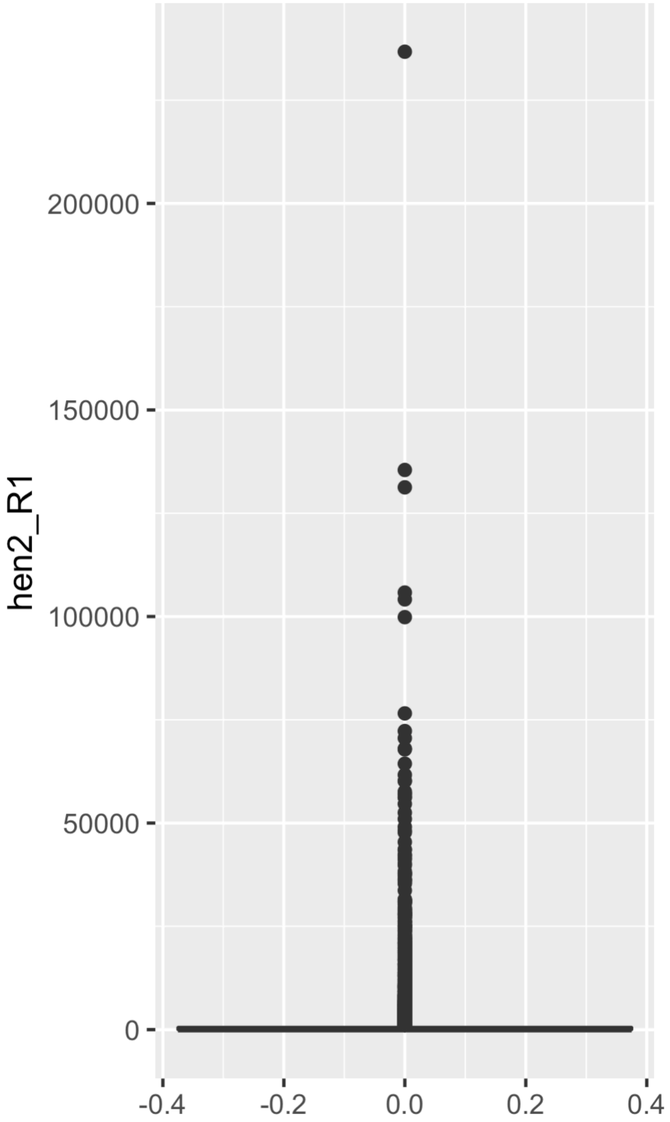
- A simple yet very informative plot for investigating distribution is the BOX PLOT
- Read more about boxplots https://www.data-to-viz.com/caveat/boxplot.html
- In essence:



Our first box plot

```
ggplot(data=TCs, aes(y=hen2_R1)) +
        geom_boxplot()
```

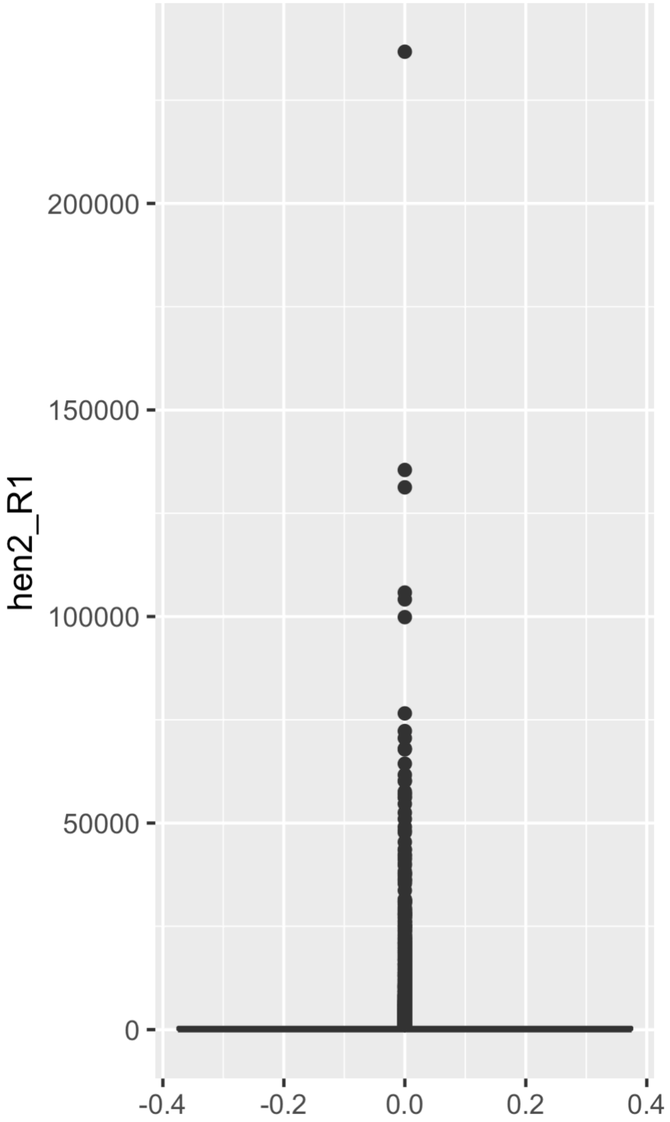
- What is happening?
- => Because some CAGE TCs have extremely high expression, they pull the scale up and crunch the distribution. We cannot see the box, but a lot of points (box plot “outliers”).
- Solution: log-scale!
  /!\ Do not log-transform the ***data***, log-transform the graphical scale!



Our first box plot

```
ggplot(data=TCs, aes(x='hen2 rep 1', y=hen2_R1)) +
    geom_boxplot()
```

- What is happening?
- => Because some CAGE TCs have extremely high expression, they pull the scale up and crunch the distribution. We cannot see the box, but a lot of points (box plot “outliers”).
- Solution: log-scale!
  /!\ Do not log-transform the ***data***, log-transform the graphical scale!



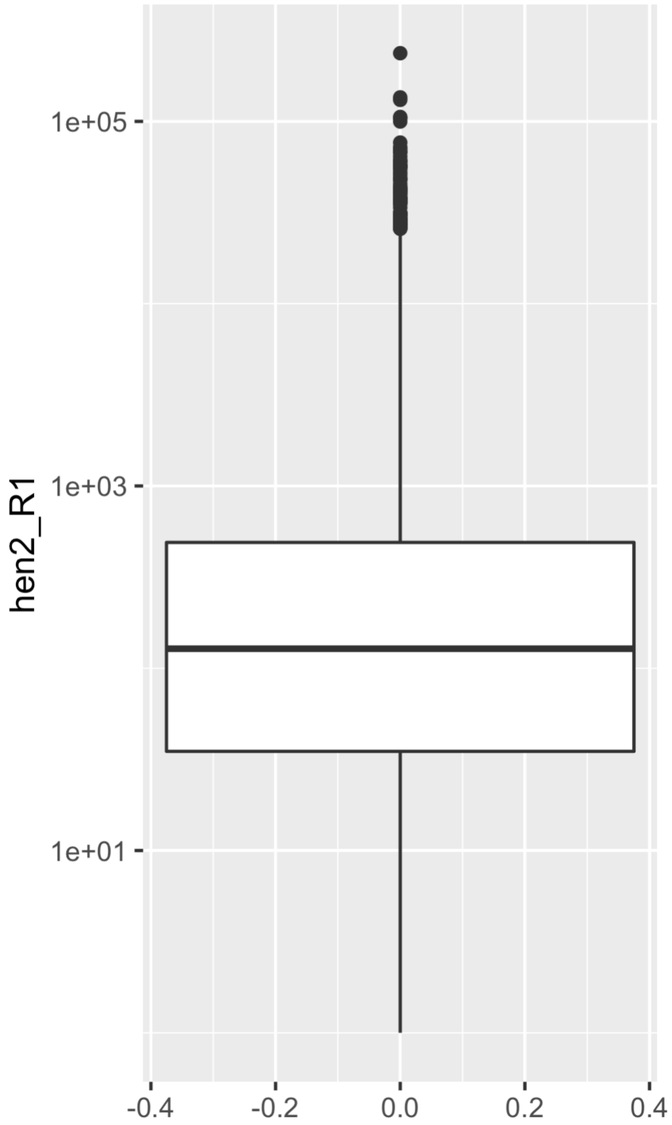
Our first box plot

```
ggplot(data=TCs, aes(x='hen2 rep 1', y=hen2_R1)) +
        geom_boxplot() +
        scale_y_log10()
```

- The plot is much better, but…
- Tidylog outputs some warning: some data points have been lost… why?!

```
Warning messages:
1: Transformation introduced infinite values in continuous y-axis
2: Removed 158 rows containing non-finite values (stat_boxplot).
```

- Tips: try `sum(TCs$hen2_R1 == 0)`



Comparing samples using box plots

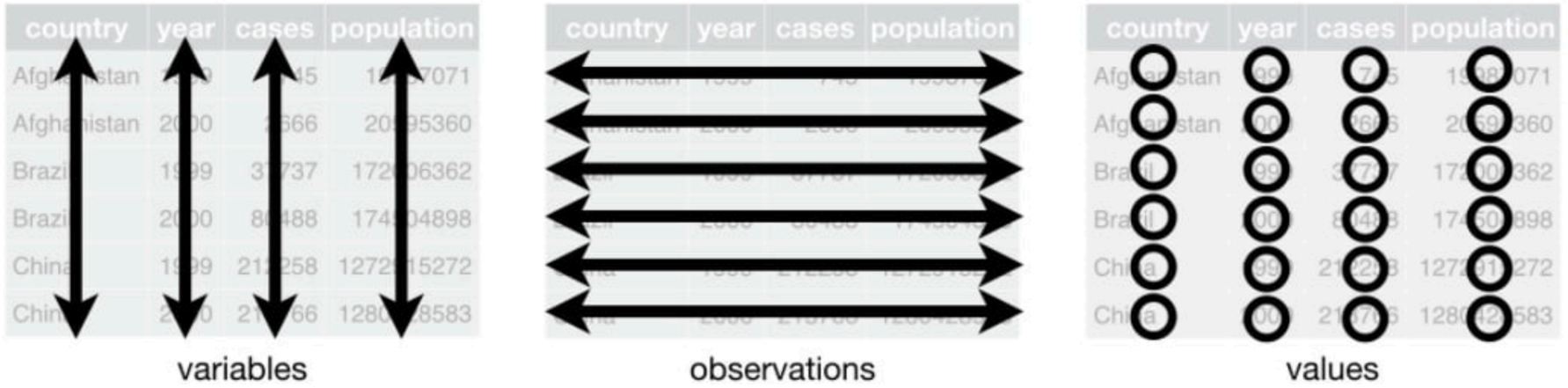
- Problem: all of our coordinate parameters (x=…, y=…) are already occupied. We cannot indicate more samples (aka more columns) in that way.

```
ggplot(data=TCs, aes(x='hen2 rep 1', y=hen2_R1)) +
        geom_boxplot() +
        scale_y_log10()
```

- Melt the data frame 🔥

Wide & long format

- A typical data frame (or Excel sheet) is in the wide format
- Rows = variables (f.ex. samples)
  Columns = observations (f.ex. genes or CAGE TCs)
  Cells = values (f.ex. expression)
- It is called “**wide**” because such data frame is usually… wide.



Wide & long format

- A so-called “tidy” data frame will have a **long** format: each row is **ONE** value associated to one variable and one observation

WIDE

id	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	5.1	3.5	1.4	0.2
2	4.9	3.0	1.4	0.2
3	4.7	3.2	1.3	0.2
4	4.6	3.1	1.5	0.2
5	5.0	3.6	1.4	0.2

LONG

id	variable	value
1	Sepal.Length	5.1
2	Sepal.Length	4.9
3	Sepal.Length	4.7
4	Sepal.Length	4.6
5	Sepal.Length	5.0
1	Sepal.Width	3.5
2	Sepal.Width	3.0
3	Sepal.Width	3.2
4	Sepal.Width	3.1
5	Sepal.Width	3.6
1	Petal.Length	1.4
2	Petal.Length	1.4
3	Petal.Length	1.3
4	Petal.Length	1.5
5	Petal.Length	1.4
1	Petal.Width	0.2
2	Petal.Width	0.2
3	Petal.Width	0.2
4	Petal.Width	0.2
5	Petal.Width	0.2

Wide & long format

- A wide data frame allows to map each column (via the `aes()`) to a geometric layer of `ggplot2`
- Conceptually:

```
ggplot(long_df, aes(x=variable, y=value) +
       geom_boxplot()
```

- Reformatting a data frame from **wide to long** format can be done with the `melt()` function from the `reshape2` package
- **Note:** The opposite, from **long to wide**, can be done with `pivot_wider()`
- Let's try it with out CAGE TC expression boxplot

LONG

id	variable	value
1	Sepal.Length	5.1
2	Sepal.Length	4.9
3	Sepal.Length	4.7
4	Sepal.Length	4.6
5	Sepal.Length	5.0
1	Sepal.Width	3.5
2	Sepal.Width	3.0
3	Sepal.Width	3.2
4	Sepal.Width	3.1
5	Sepal.Width	3.6
1	Petal.Length	1.4
2	Petal.Length	1.4
3	Petal.Length	1.3
4	Petal.Length	1.5
5	Petal.Length	1.4
1	Petal.Width	0.2
2	Petal.Width	0.2
3	Petal.Width	0.2
4	Petal.Width	0.2
5	Petal.Width	0.2

Wide & long format

- A wide data frame allows to map each column (via the `aes()`) to a geometric layer of `ggplot2`
- Conceptually:

```
ggplot(long_df, aes(x=variable, y=value) +
       geom_boxplot()
```

- Reformatting a data frame from wide to long format can be done with the `melt()` function from the `reshape2` package
- Let's try it with out CAGE TC expression boxplot

Wide & long format

- Reproduce the following command and explain what happens

```
12  # wide format
13  TCs
14  # melt df to long format, using TC_id as "key"
15  TCs %>%
16      melt(id.vars='TC_id') %>%
17      head()
```

- **Note:** our TCs data frame used to be a tibble. Certain operations on tibbles coerce a tibble back to a data frame. Use of the `head()` command allows to not be flooded by the whole data frame content (à la UNIX "`cat`").

Wide & long format

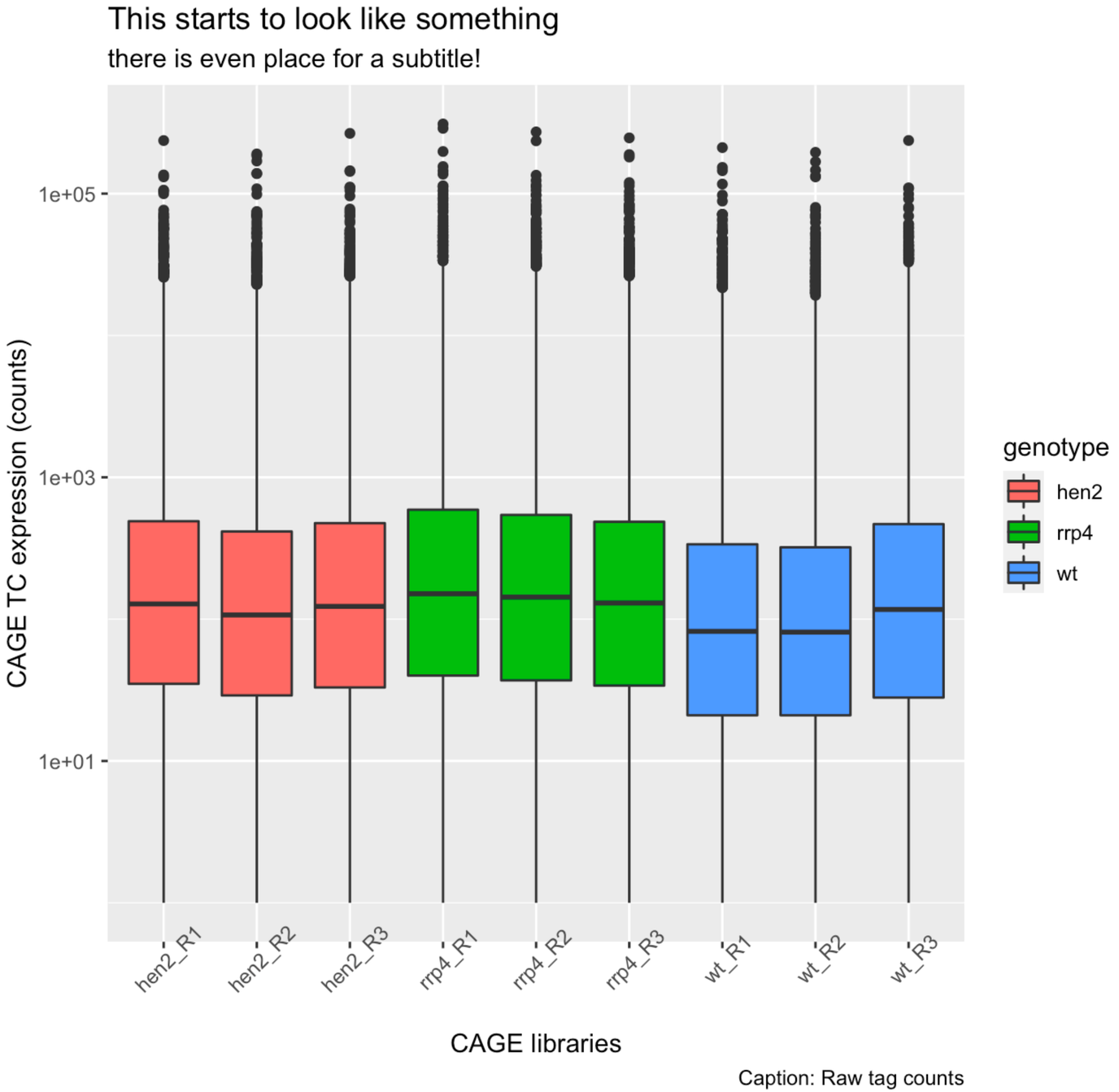
Reproduce the code **line by line** and dissect what happens

- Let's jump ahead:

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id') %>%
    separate(col='variable', into=c('genotype', 'replicate'), sep='_', remove=F) %>%
    ggplot(aes(x=variable, y=value, fill=genotype)) +
            geom_boxplot() +
            scale_y_log10() +
            theme(axis.text.x=element_text(angle=45),
                  aspect.ratio=1) +
            labs(x='CAGE libraries', y='CAGE TC expression (counts)',
                 title='This starts to look like something',
                 subtitle='there is even place for a subtitle!',
                 caption='Caption: Raw tag counts')
```

Wide & long format

- RESULT:



Wide & long format

Step 1: go from wide to long format

- Let’s jump ahead:

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id')
```

- Here we use the column “TC_id” as the “key”

Wide & long format

Step 2: extract genotype information only (not replicate) by splitting a column based on underscore

- Let's jump ahead:

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id') %>%
    separate(col='variable', into=c('genotype', 'replicate'), sep='_', remove=F)
```

- `separate()` is a function that splits a column into several columns, based on a character string (can be a regex!)

- one option is to not have the original column deleted (`remove=F`)

Wide & long format

Step 3: pass to ggplot() and map aesthetic (`aes`) to columns

- Let's jump ahead:

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id') %>%
    separate(col='variable', into=c('genotype', 'replicate'), sep='_', remove=F) %>%
    ggplot(aes(x=variable, y=value, fill=genotype))
```

- We can now initiate the `ggplot()` call, and use one column for the X variable, one for the values (Y), and use a filling color (`fill=`) thanks to the genotype column

- Important: `ggplot2` will use different fills for each genotype, and hence will create one boxplot for each genotype automatically, this is the power of the grammar of graphics

Wide & long format

Step 4: box plot on a log-scale

- Let's jump ahead:

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id') %>%
    separate(col='variable', into=c('genotype', 'replicate'), sep='_', remove=F) %>%
    ggplot(aes(x=variable, y=value, fill=genotype)) +
            geom_boxplot() +
            scale_y_log10()
```

- We impose a couple of supplemental graphic layers (+) that will specify a boxplot and a log10-scaled Y-axis

Wide & long format

Step 5: Improve the plot's general parameters, including layout

- Let's jump ahead:

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id') %>%
    separate(col='variable', into=c('genotype', 'replicate'), sep='_', remove=F) %>%
    ggplot(aes(x=variable, y=value, fill=genotype)) +
           geom_boxplot() +
           scale_y_log10() +
           theme(axis.text.x=element_text(angle=45),
                 aspect.ratio=1)
```

- The `theme()` layer controls a lot of theme-related features, such as the axis labels, overall plot aspect ratio, and SO many more.

Wide & long format

- Let's jump ahead:

Step 6: Make the plot understandable with legends (labels: `labs( )`)!!!
=> *very important*.

```
# wide format
TCs
# melt df to long format, using TC_id as "key", and plot
TCs %>%
    melt(id.vars='TC_id') %>%
    separate(col='variable', into=c('genotype', 'replicate'), sep='_', remove=F) %>%
    ggplot(aes(x=variable, y=value, fill=genotype)) +
            geom_boxplot() +
            scale_y_log10() +
            theme(axis.text.x=element_text(angle=45),
                  aspect.ratio=1)
            labs(x='CAGE libraries', y='CAGE TC expression (counts)',
                 title='This starts to look like something',
                 subtitle='there is even place for a subtitle!',
                 caption='Caption: Raw tag counts')
```

But… why?

- In this block of code, you:
  - 1. Reshaped the data (data wrangling)
  - 2. Extracted genotype information for each expression value (separate)
  - 3. Summarised the expression distribution of each sample (boxplot with quartiles, outliers, etc)
  - 4. Visualized the data with genotype indication (fill=genotype)
  - 5. Scaled the visualisation to make the figure intelligible
  - 6. Set a plot layout with all the essential scientific info (labels)

- In one command, very close to simple English

- Without altering the underlying dataset (`TCs` data frame)

- Without creating temporary variables

Normalize expression: Raw counts vs TPM

- **TPM** = **T**ags **P**er **M**illion mapped tags

- TPM = normalisation of expression data with regards to sequencing depth
  - accounts for technical sequencing differences
  - doesn't account for biological differences as RPKM

- **RPKM** = Reads Per Kilobase per Million mapped reads
  - accounts for technical sequencing differences
  - accounts for each gene's length (longer gene = more reads)

- RPKM is typically needed for RNA-seq when we need to compared different genes, but not when comparing the same gene between samples

- Since CAGE only captures TSSs, read length does not weigh in, so TPM is enough

Read more on this here: www.reneshbedre.com/blog/expression_units.html

Let’s normalise our CAGE TC expression

Difficult exercise

- Formula: *TPM = tags count / (total mapped tags / 1,000,000)*
- **1a)** Compute the sum of reads (aka tags) for each sample
  1b) and divide it by 1,000,000
  1c) and save it as “`TPM_factor`”

Let's normalise our CAGE TC expression

Difficult exercise

- Formula: *TPM = tags count / (total mapped tags / 1,000,000)*

- Apply the formula above:
  2a) Divide each gene's tag count by the TPM factor of the relevant library.

  - Tips 1: this is dividing each ROW of the TCs data frame by the vector of `TPM_factor`, the sort of operation `apply()` is good at ;)

  - Tips 2: look at the `dim()` before and after `apply()`, what happened? could `t()` be useful? What does `t()` do?

  2b) Save the TPM-normalized data frame as "`TCs_TPM`"
  2c) Does the sum of normalised TPM per sample amounts to 1million?

Let’s normalise our CAGE TC expression

Difficult exercise

- Formula: *TPM = tags count / (total mapped tags / 1,000,000)*
- **3)** Using our new data frame:
  Make a boxplot of the normalised CAGE TC expression across samples, color-filled by genotype
- **4)** What’s different compared to the raw count box plots?