

Radvance

PhD Toolbox - R introductory course - stream II
Martino ADAMO
Matteo CHIALVA





welcome back!

Stream 2 R for Advanced Users

Day 1 - January 20, 2025, 10:00-13:00

- Scientific graphics in R
- Advanced data visualization with ggplot2 - Part I

Day 2 - January 23, 2025, 10:00-13:00

- Advanced data visualization with ggplot2 - Part I

Day 3 - January 27, 2025, 10:00-13:00

- Geographic data introduction
- Plotting maps in R - Part I

Day 4 - January 29, 2025, 10:00-13:00

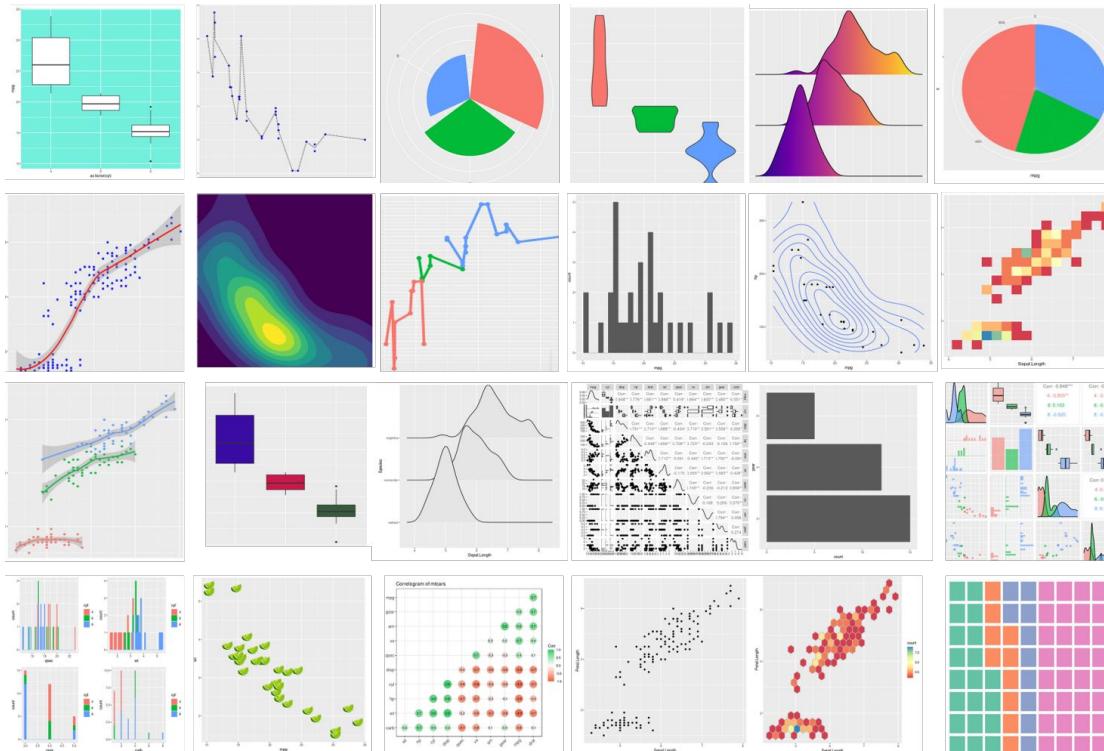
- Plotting maps in R - Part II
- Working with lists
- Reproducible data reporting with RMarkdown and Quarto

Day 5 - February 3, 2025, 14:00-18:00

**Aula 1 - Via Accademia Albertina 13
(DBIOS)**

- Final Data wrangling & visualization workshop

ggplot2 - the state of the art of graphics in R



check more on
<https://r-graph-gallery.com/>



prepare your session!

```
> library(tidyverse)  
> library(ggplot2)  
> library(ggrepel)  
> library(lemon)  
> library(RColorBrewer)
```



data import

Exercise! Import the Flowers dataset (flowers.txt and flowers_details.tsv)

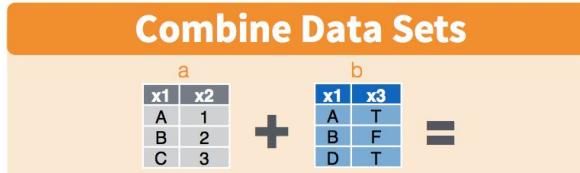
```
# download the flowers datasets from the course repository or import from your local folder
```

```
>flowers<-read.table("https://raw.githubusercontent.com/mchialva/PhDToolbox2026/main/Datasets/flowers/flowers.tsv", header=T)
```

```
>flowers_details<-read.table("https://raw.githubusercontent.com/mchialva/PhDToolbox2026/main/Datasets/flowers/flowers_details.tsv", header=T)
```



dplyr: left_join()



```
df %>% left_join(df2, index)
```

```
# We need to join the flowers dataset with flowers_details.tsv table which  
contains additional attributes
```

```
# join (or merge) and save into a new object
```

```
> flowers_all<- flowers %>% left_join(flowers_details, by="SP")
```

key components

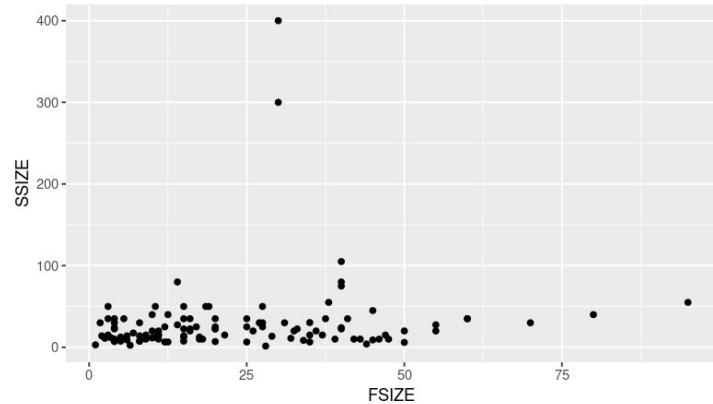
ggplot2 library implements a modular language.
Each element add/modifies your graph.

Every ggplot2 plot has four key components:

1. data (long data format)
2. aesthetic
3. layer
4. theme

```
> ggplot(flowers_all, aes(x = FSIZE, y = SSIZE)) +  
  geom_point() +  
  theme()
```

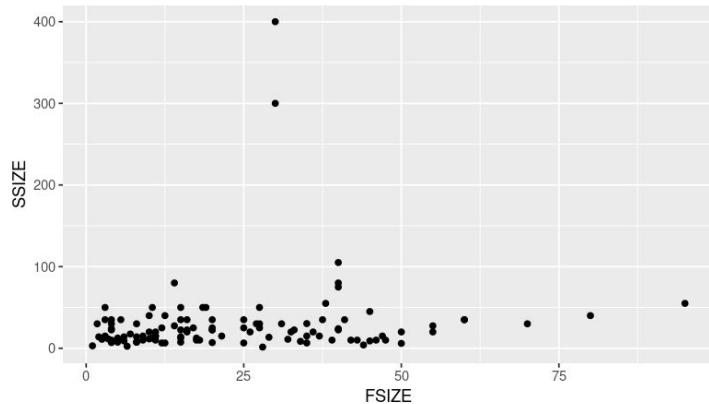
data aesthetic layer theme





key components

```
> ggplot(flowers_all, aes(x = FSIZE, y = SSIZE)) +  
  geom_point()
```



ggplot2 uses a specific grammar pay attention to those few things:

- **ggplot()** is the main function to create a plot object
- **aesthetics** are specified nested into ggplot()
- layers are appended using a “+” at the end of the element before

e.g. **this won't work!**

this will

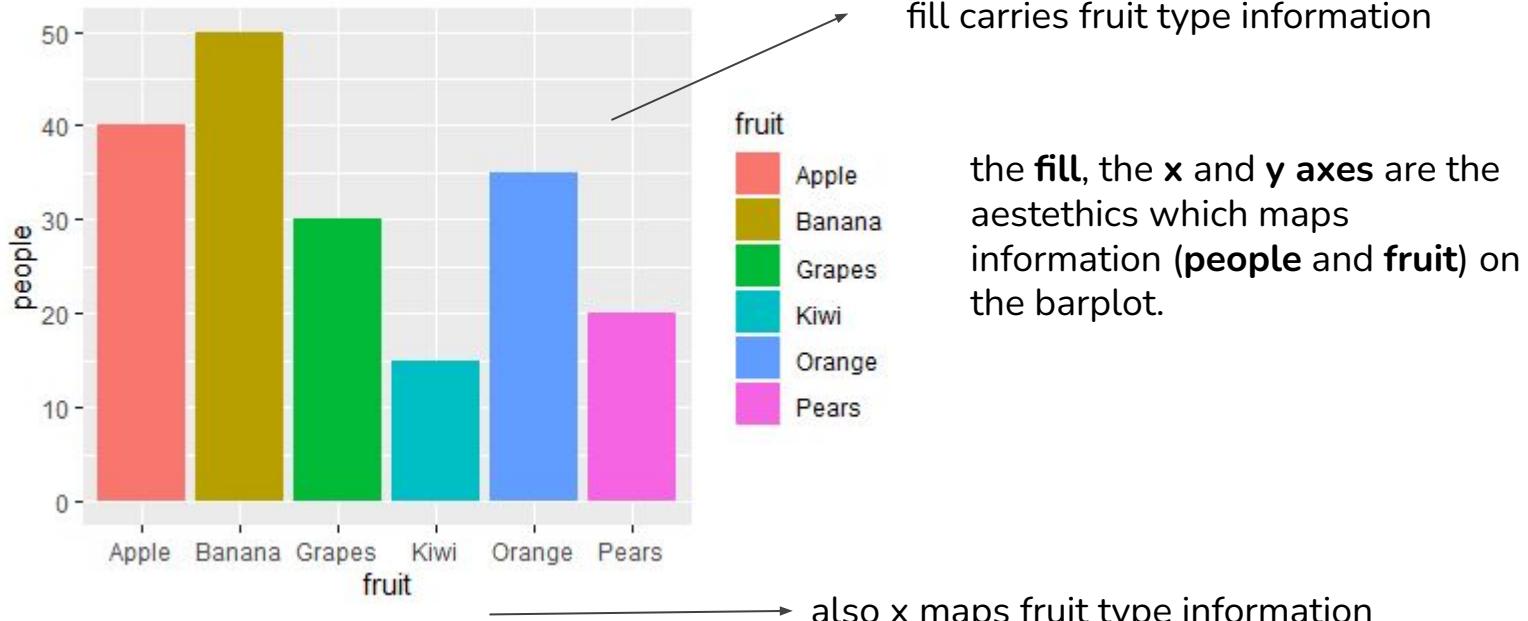
```
> ggplot()  
  +geom_point()
```

```
ggplot() +  
  geom_point()
```



aesthetic

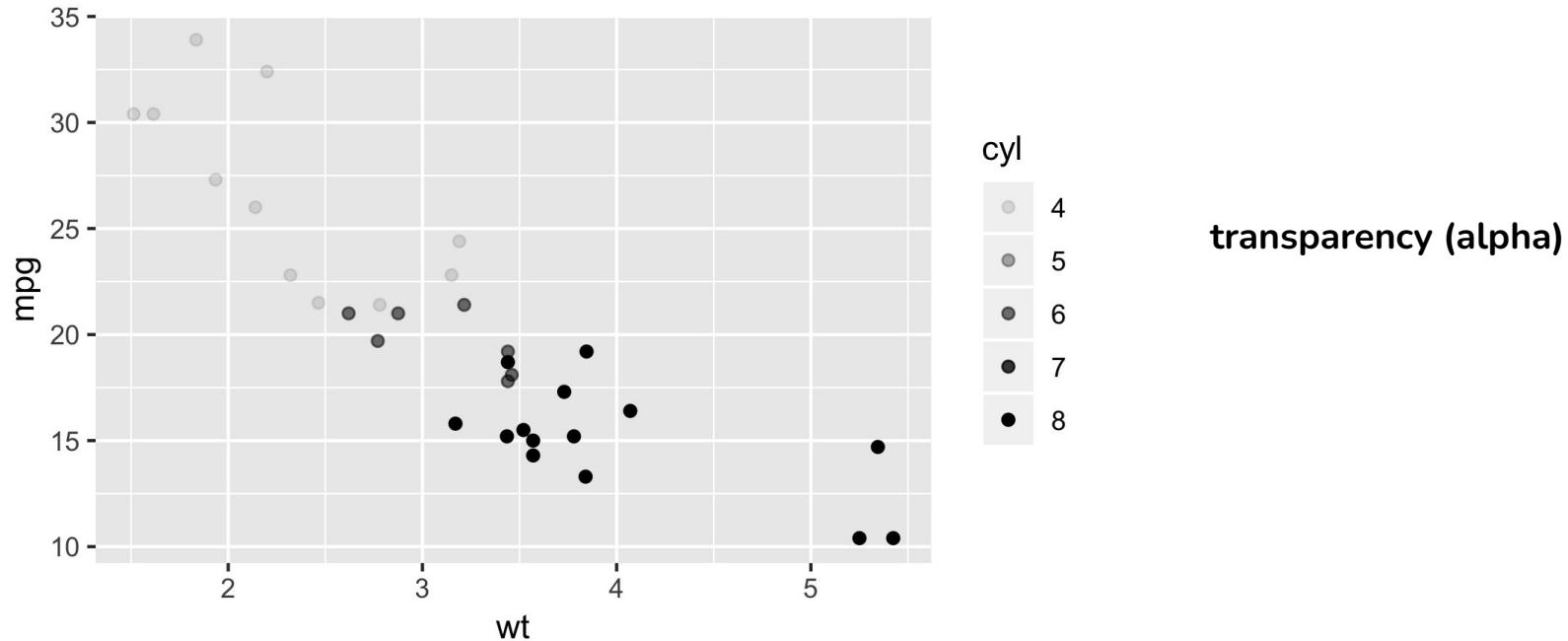
aesthetics are graphical features which map the information contained in your data variables





aesthetic

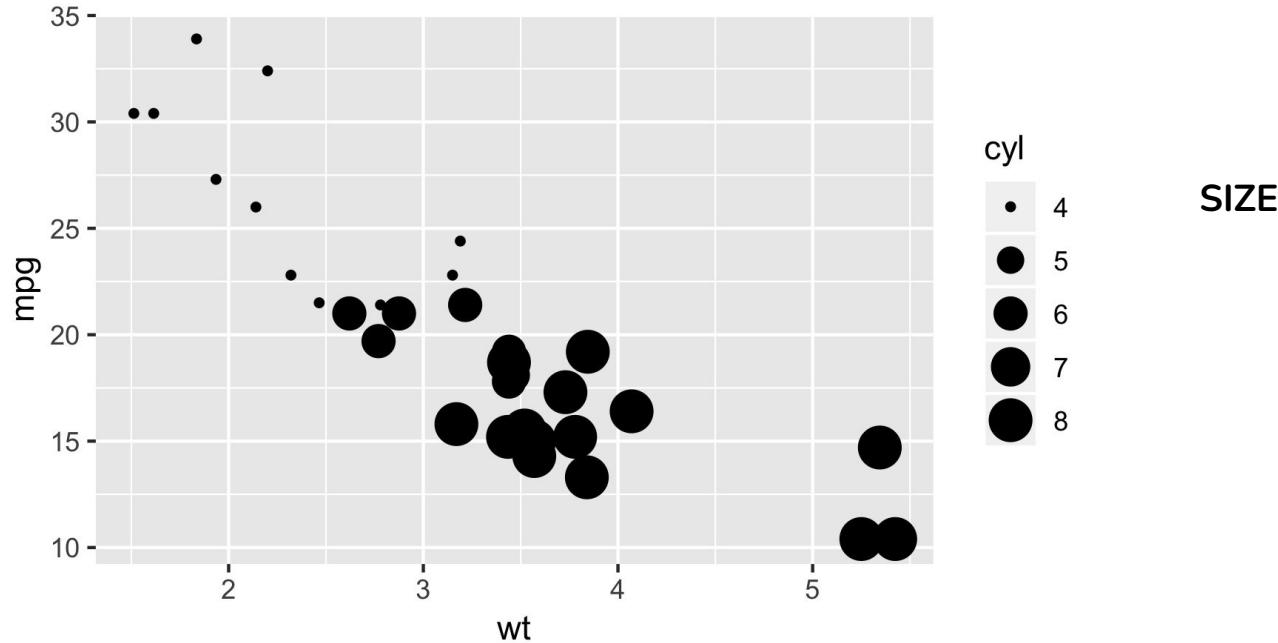
Information can be mapped on almost all the geometries and graphical features you can see on a plot





aesthetic

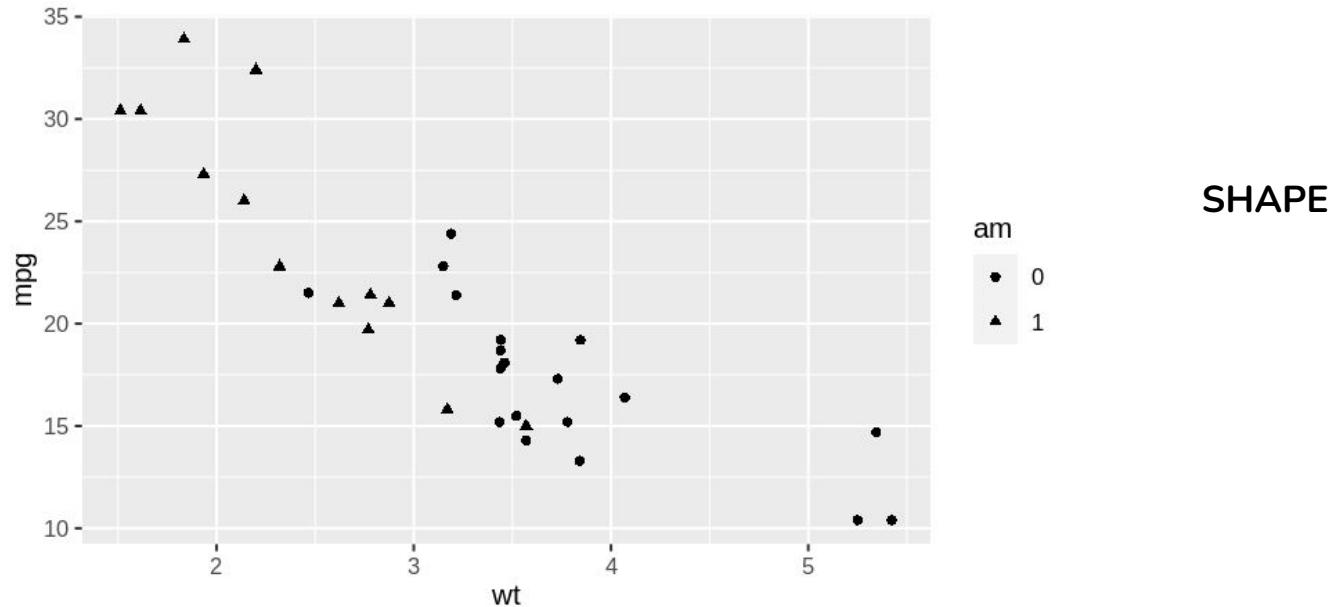
Information can be mapped on almost all the geometries and graphical features you can see on a plot





aesthetic

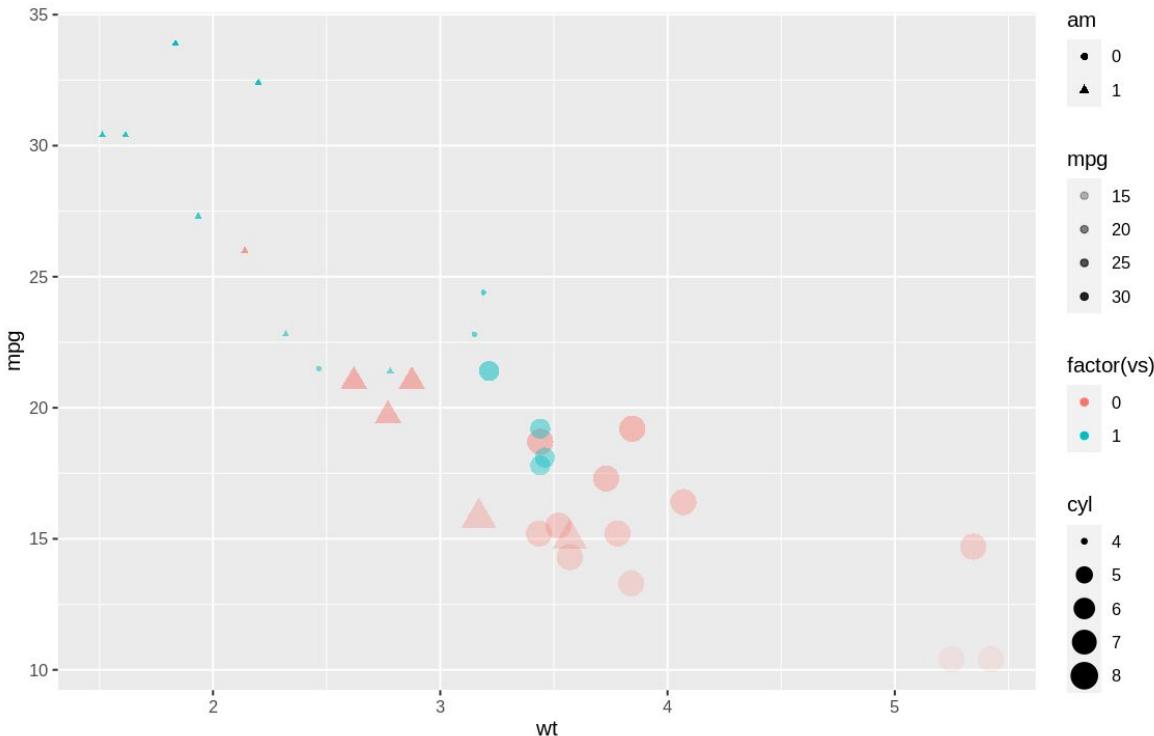
Information can be mapped on almost all the geometries and graphical features you can see on a plot





aesthetic

Information can be mapped on almost all the geometries and graphical features you can see on a plot



You can map multiple aesthetics at once

color: motor type (v- or in line)

size: number of cylinders

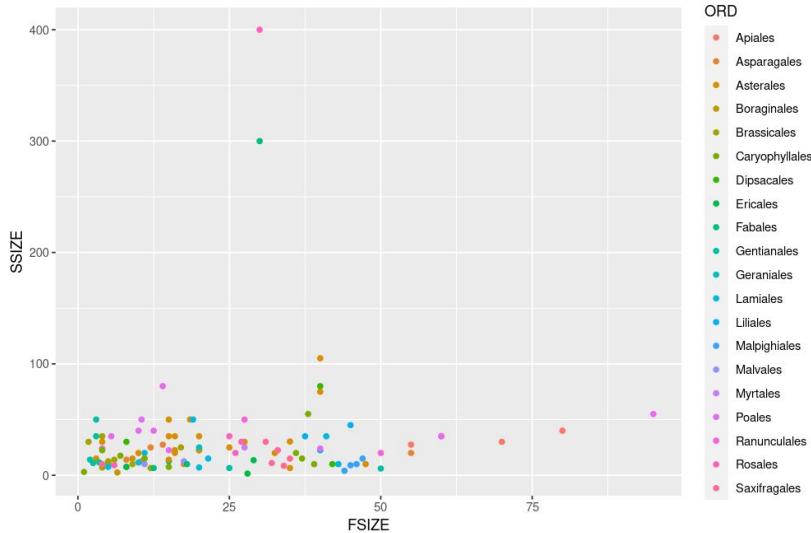
alpha: miles per gallon

shape: transmission type



aesthetic

`aes()` nested into the `ggplot()` function works as factors/variable classifier



```
> ggplot(flowers_all, aes(x = FSIZE, y = Ssize, color = ORD)) +  
  geom_point()
```



aesthetic

```
> ggplot(flowers_all, aes(x = FSIZE, y = SSIZE)) +  
  geom_point()
```

the first two arguments to be specified in the **aes()** nested into the **ggplot()** function
are always the x and the y

and now a little complication
aes() could be applied as **ggplot()** aesthetic or as layer aesthetic
two different meaning!



aesthetic

As in the previous case, **aes()** can be set within the **ggplot()** function.
In this case aesthetics applies to all the geometries (geom_...) plotted.

```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD)) + geom_point()
```

In the case of multiple geometries are applied to the same plot, you may want to set different **aes()** for each of them:

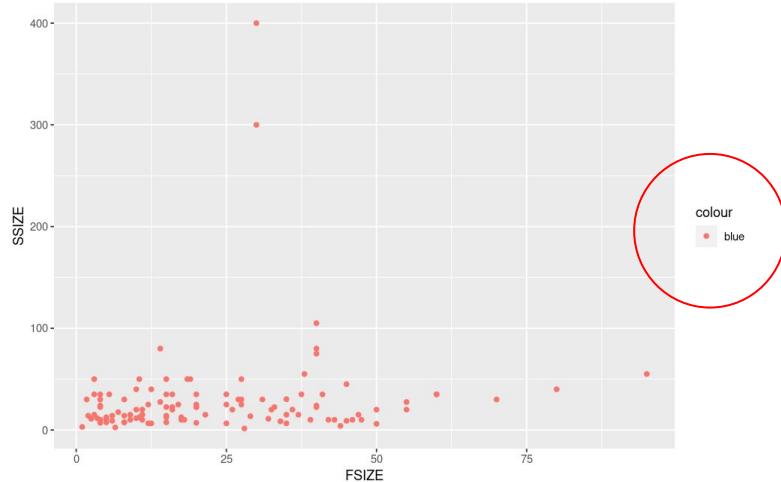
```
>ggplot(flowers_all) + geom_point(aes(x = FSIZE, y = SSIZE, color = ORD))
```

The two codes in this case output the same plot!

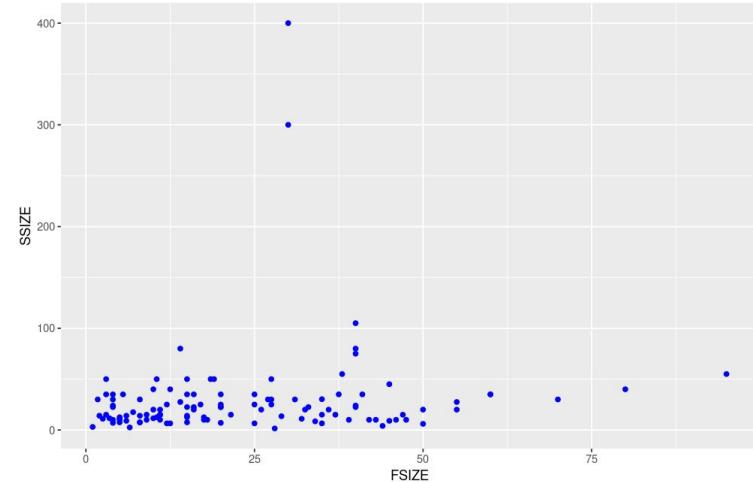


aesthetic

Be careful: aesthetics cannot be used to define fixed colours!



```
>ggplot(flowers_all) +  
  geom_point(aes(x = FSIZE, y = SSIZE,  
                 color = "blue"))
```



```
>ggplot(flowers_all) +  
  geom_point(aes(x = FSIZE, y = SSIZE),  
             color = "blue")
```

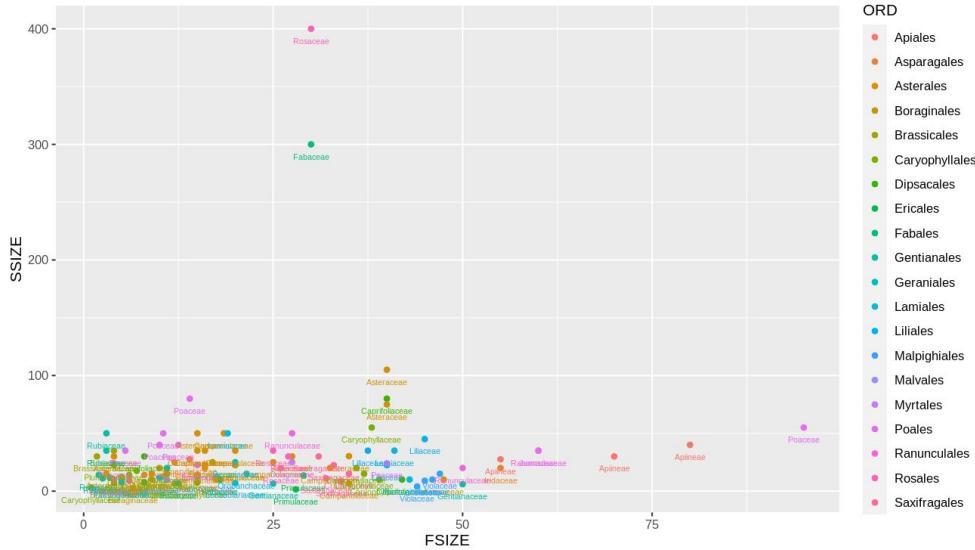
NB: the same terms used to define aesthetics (except x and y) can be used outside the `aes()` function with a different behaviour



aesthetic

we want to add text labels to point using the `geom_text()` function

```
> ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD, label=ORD)) +  
  geom_point() +  
  geom_text()
```

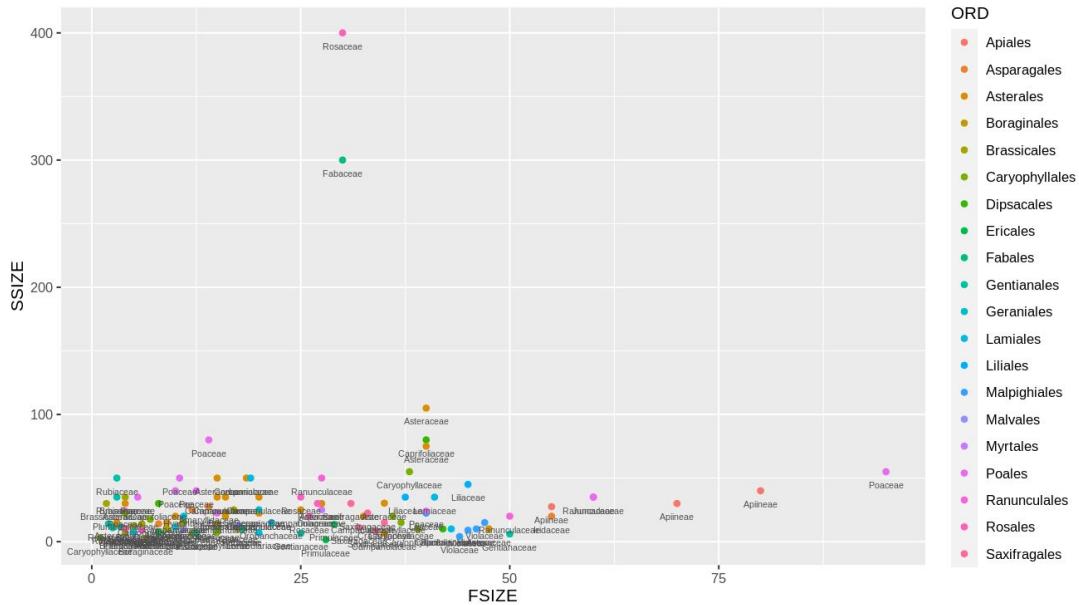




aesthetic

We want label text in a different colour (gray)!!

```
> ggplot(flowers_all, aes(x = FSIZE, y = SSIZE)) +  
  geom_point(aes(color = ORD)) +  
  geom_text(aes(label=FAM), color="gray30")
```





aesthetic and graphic variables

`aes()` function arguments are many and they depend by the layer type:

`color` layer color (including the stroke around)

`fill` layer color (excluding the stroke around) **# requires specific shapes**

`shape` dot shape (integer 0:25 + each available ASCII)

`size` how big is

`stroke` stroke thickness

`alpha` transparency

`linetype` only for lines (solid, dashed, dotter, ...)

`lineend` how the line ends (round/squared)

`linewidth` similarly to size

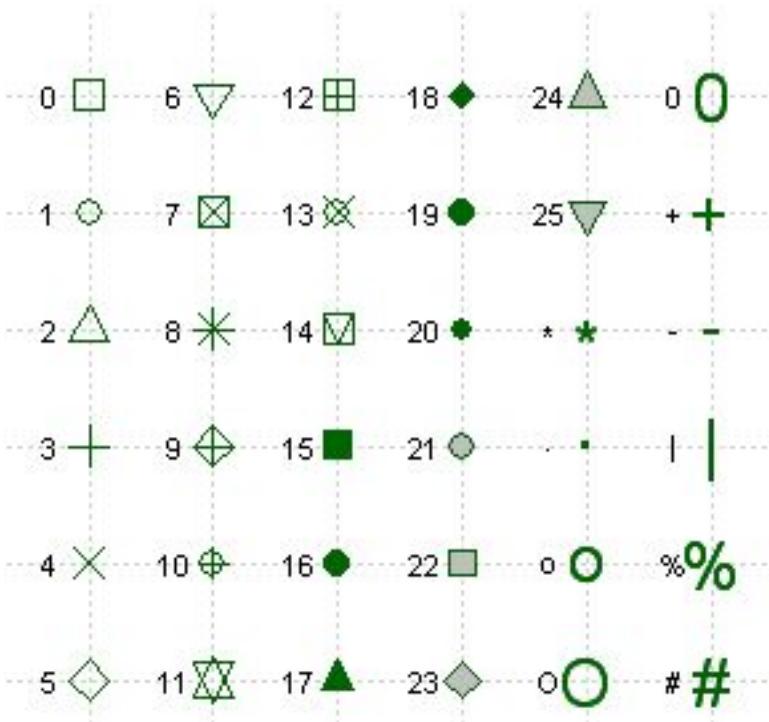
`label` gives a name to the object

`family` only for `geom_text()` corresponds to the font family

`fontface` only for `geom_text()` corresponds to the font (plain) **font (bold)** *font (italic)*



aesthetic



ggplot()
plot() => shape = n
 => pch = n



ggplot()
plot() => linetype = "type name"
 => lty = n



Exercise 1

using the flowers_all dataset draw a dot plot starting from this example:

```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD)) +  
  geom_point()
```

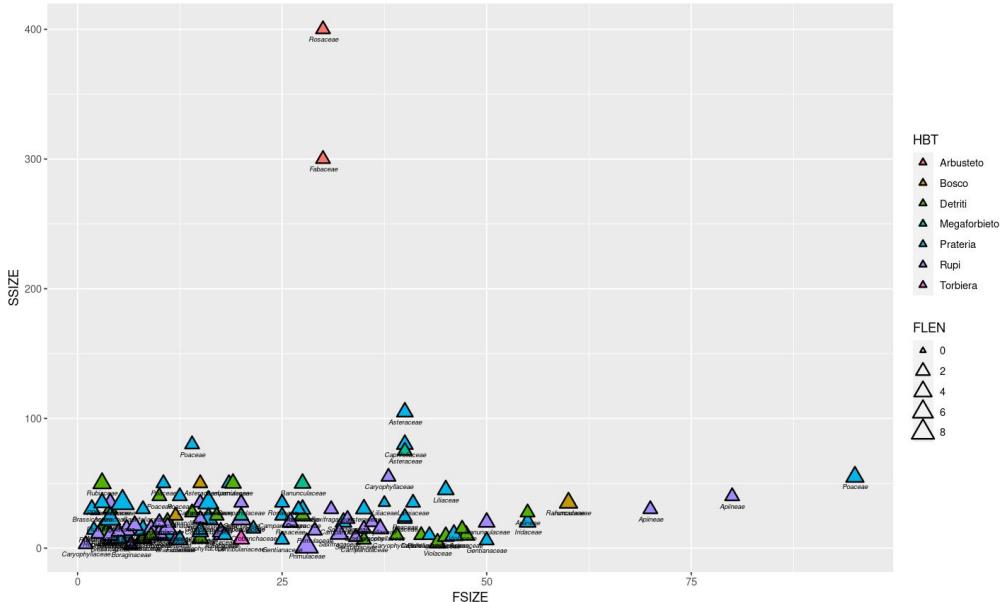
points should be triangles colored by FAM and sized by FLEN. Strokes must be black and thick. Each point should be labeled in *italic* with the FAM name at the base of the triangle (use vjust).

You will see an awful graph 😅



aesthetic exercise

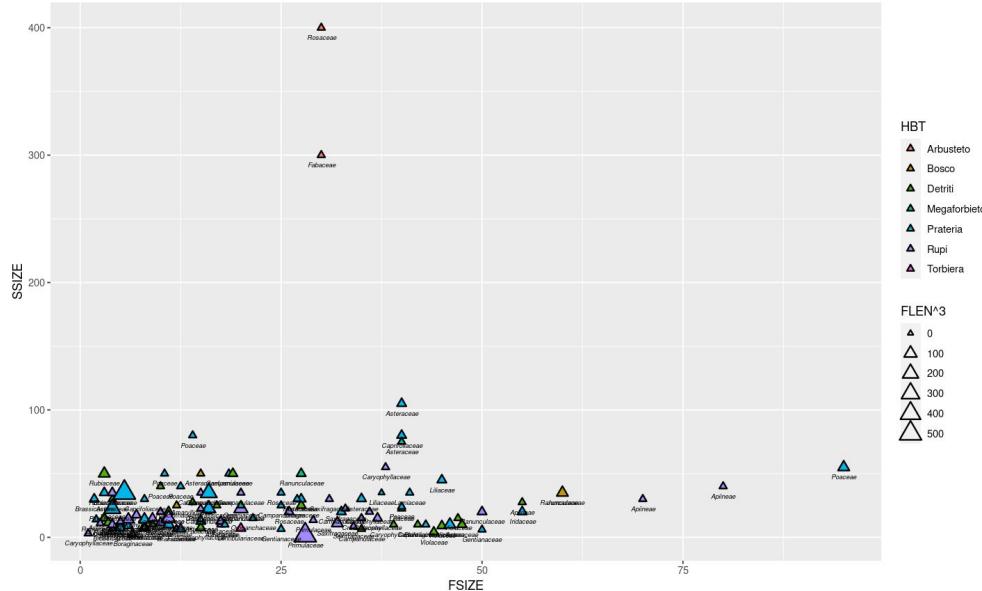
```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, fill = HBT, size = FLEN)) +  
  geom_point(shape = 24, color = "black", stroke = 1) +  
  geom_text(aes(label = FAM), size = 2, vjust = 2.5, fontface = "italic")
```





math on ggplot

variables can be modified directly into ggplot2, as well datasets can be subsetted/modified, or data ordered



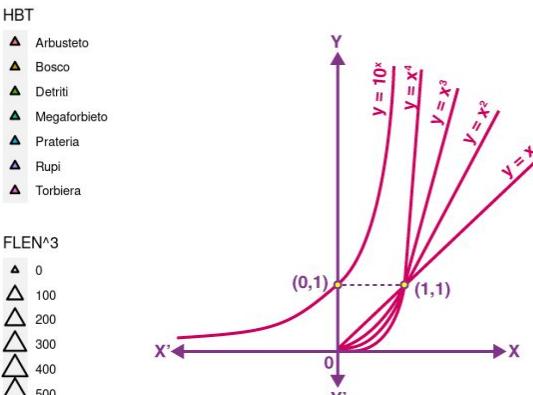
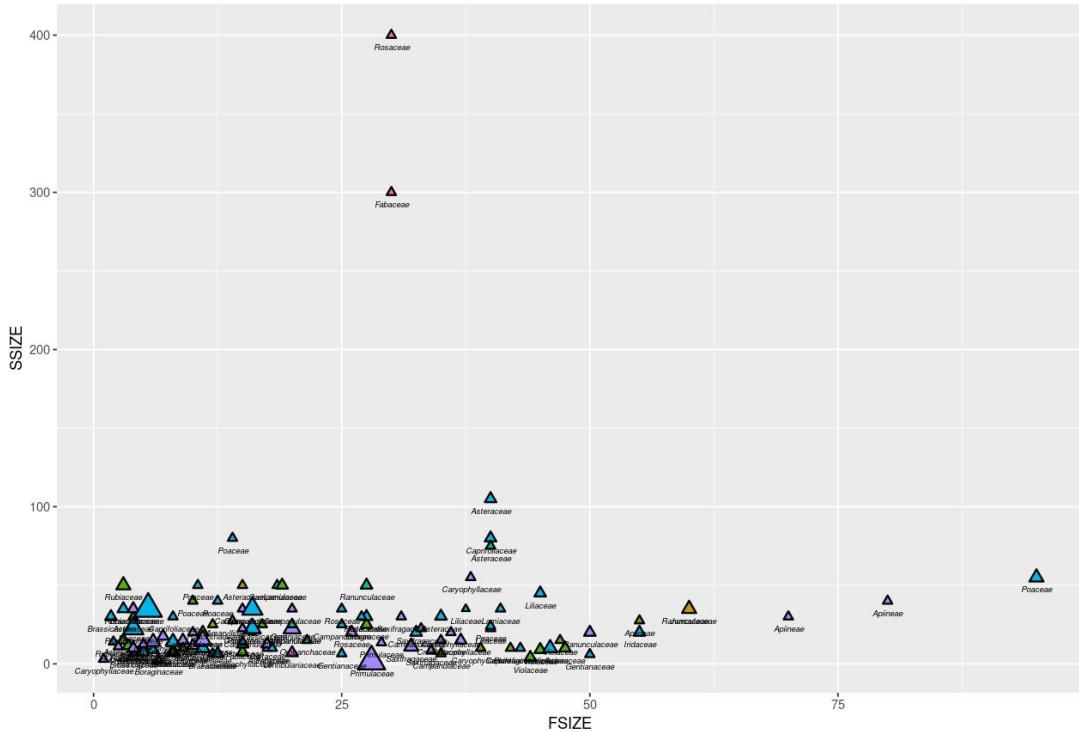
common R
function can be
applied directly to
variable names

```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, fill = HBT, size = FLEN^3)) +  
  geom_point(shape = 24, color = "black", stroke = 1) +  
  geom_text(aes(label = FAM), size = 2, vjust = 2.5, fontface = "italic")
```



math on ggplot

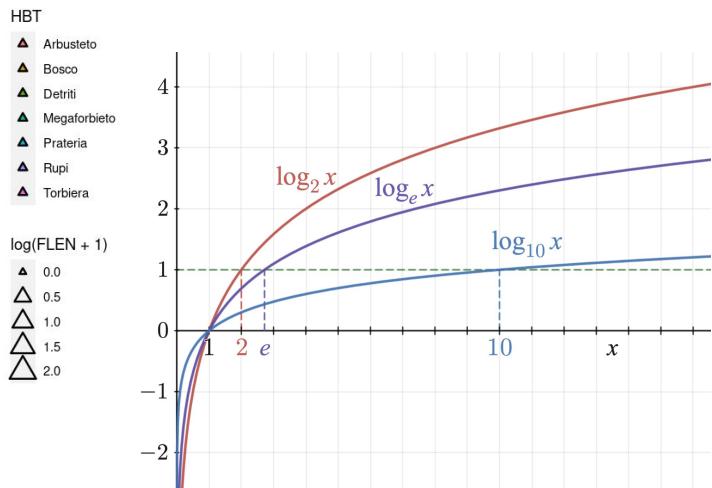
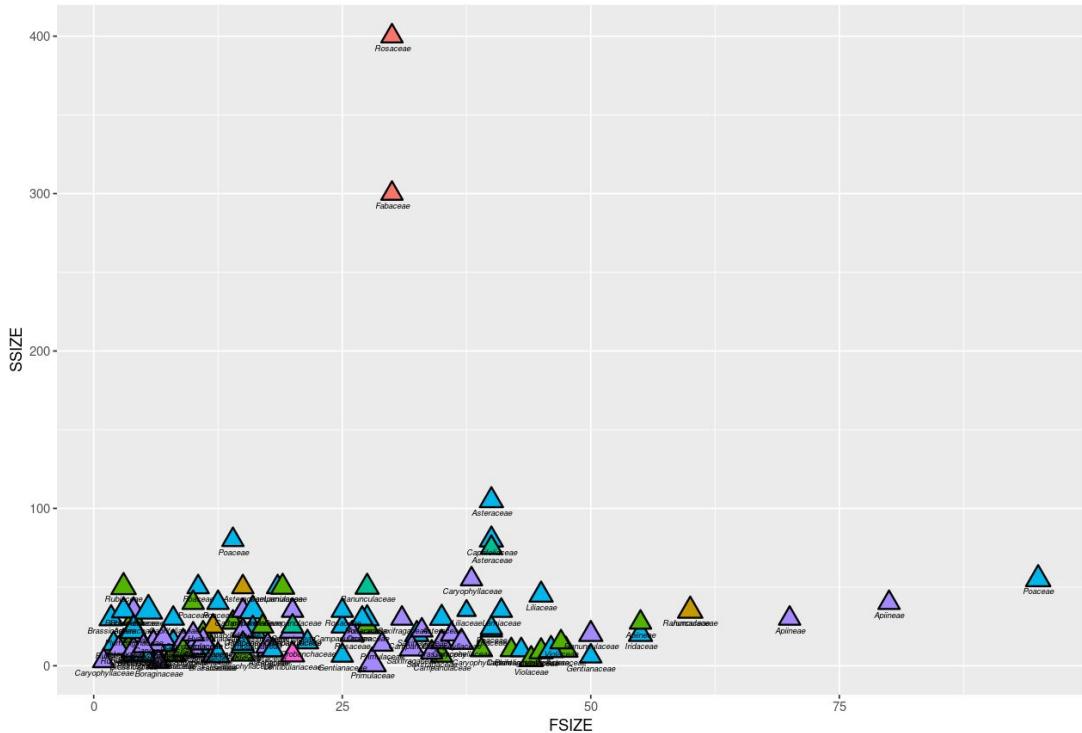
exponential function extremize differences





math on ggplot

log functions uniformize differences

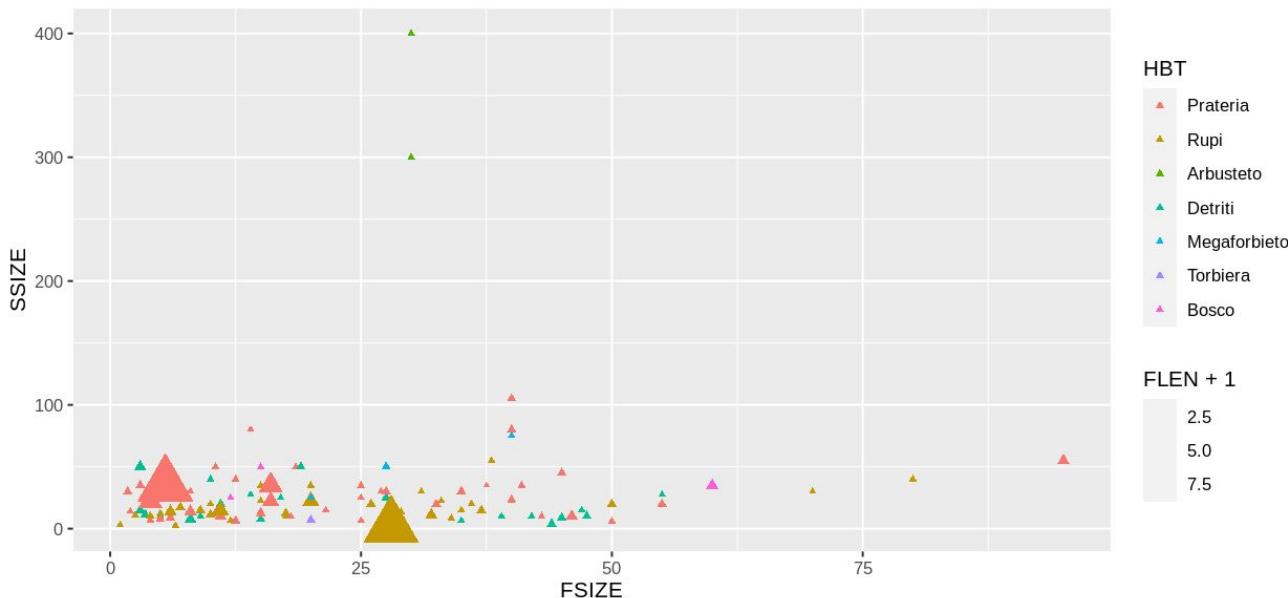




math on ggplot :: set manual scales

`scale_size_continuous()`

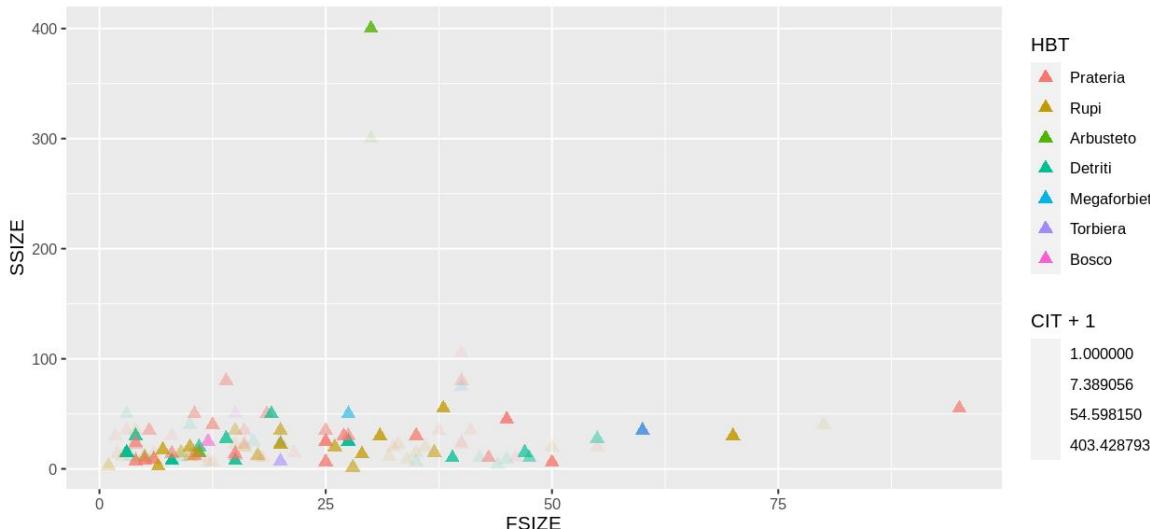
```
ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, fill = HBT, size = FLEN+1)) +  
  geom_point(shape = 24, color = "black", stroke = 0) +  
  geom_text(aes(label = FAM), size = 2, vjust = 2.5, fontface = "italic") +  
  scale_size_continuous(range = c(1, 10), trans = "exp")
```



math on ggplot :: set manual scales

scale_alpha_continuous

```
ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, fill = HBT, alpha = CIT+1)) +  
  geom_point(shape = 24, size=2, color = "black", stroke = 0) +  
  #geom_text(aes(label = FAM), size = 2, vjust = 2.5, fontface = "italic") +  
  scale_alpha_continuous(range = c(0.1, 1), trans = "log")
```

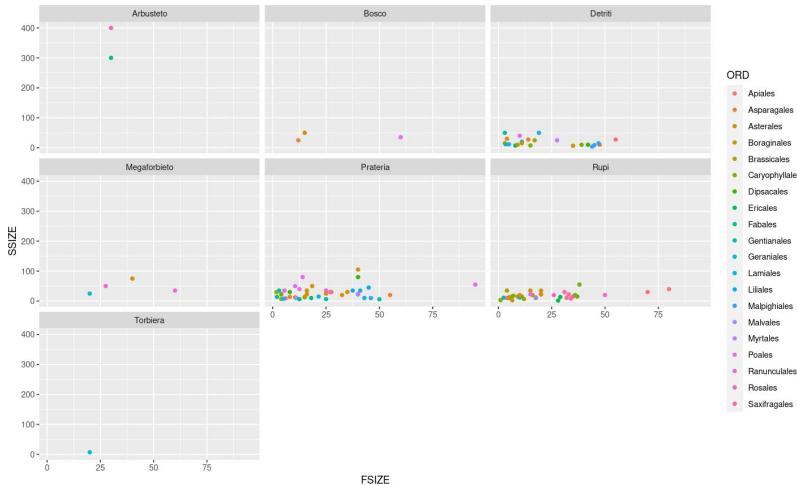




faceting

Another technique for displaying additional categorical variables on a plot is faceting. It creates tables by splitting the data and displaying the same graph for each subset.

to facet the most useful function is `facet_wrap()` it works just adding the category to which create separate displays after the character ~

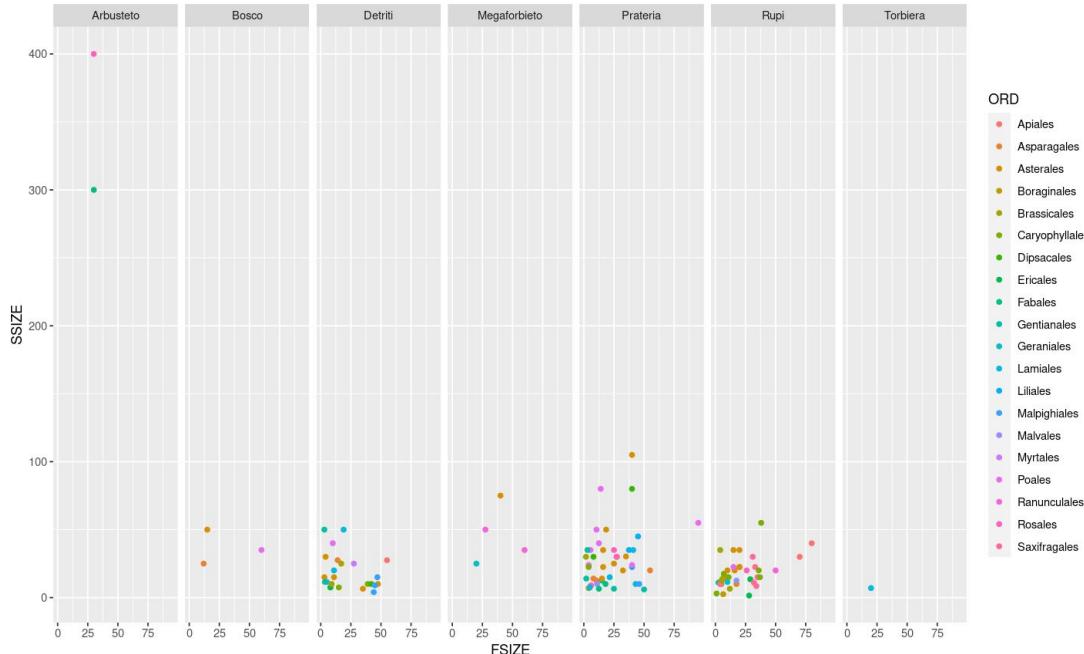


```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD)) +  
  geom_point() + facet_wrap(~HBT)
```



faceting

or you can tidy define the number of rows or columns you want

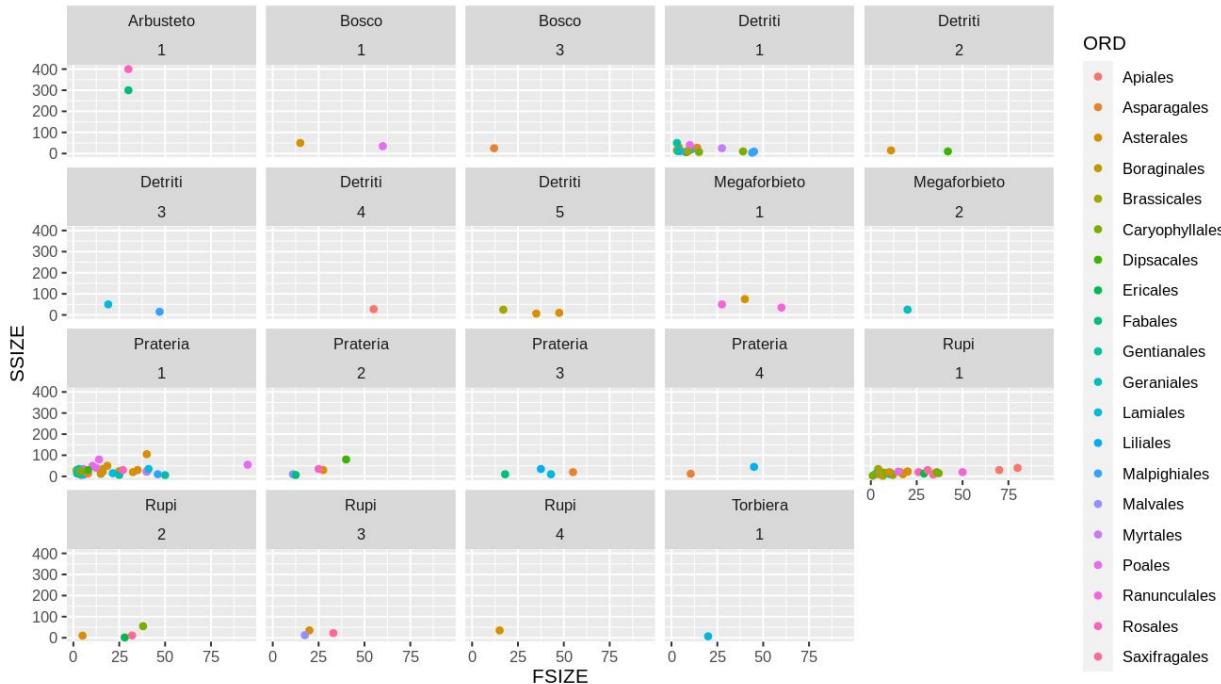


```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD)) +  
  geom_point() + facet_wrap(~HBT, ncol=7)
```



faceting

you can also make a mess adding variables to the faceting using!

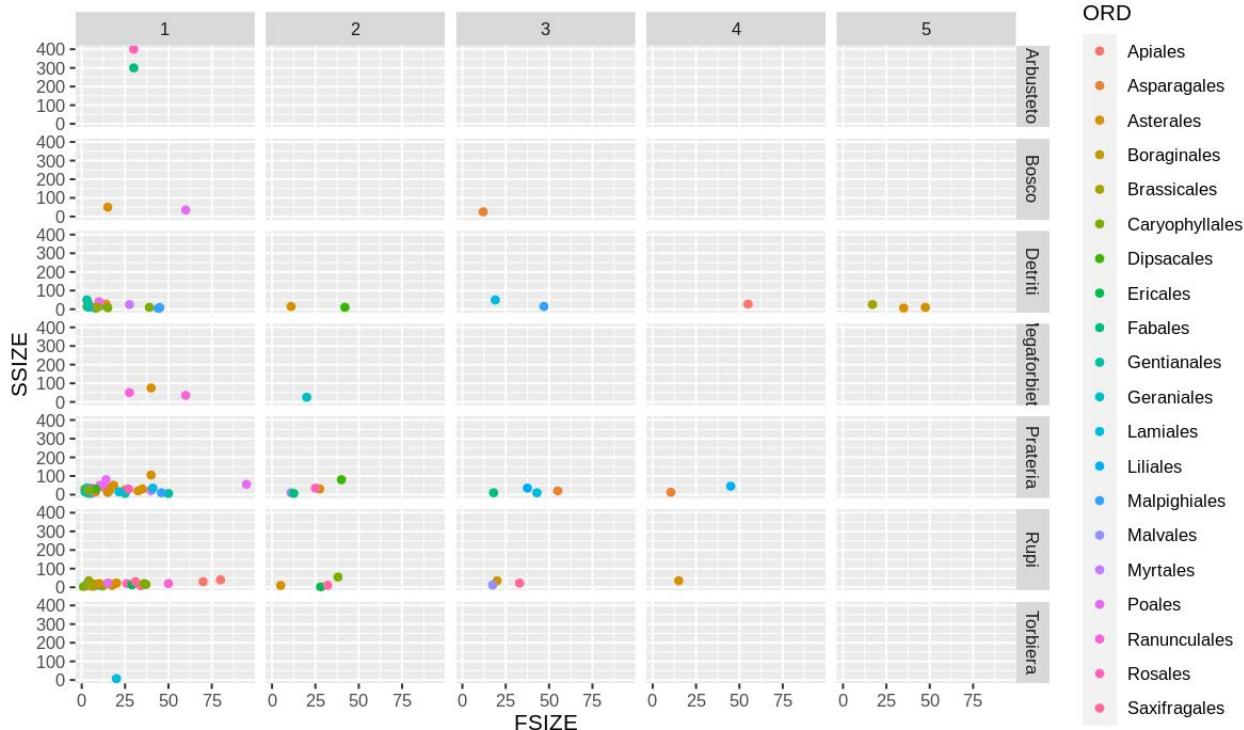


```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD)) +  
  geom_point() + facet_wrap(~HBT+IUCN)
```



faceting

Also the `facet_grid()` function is available

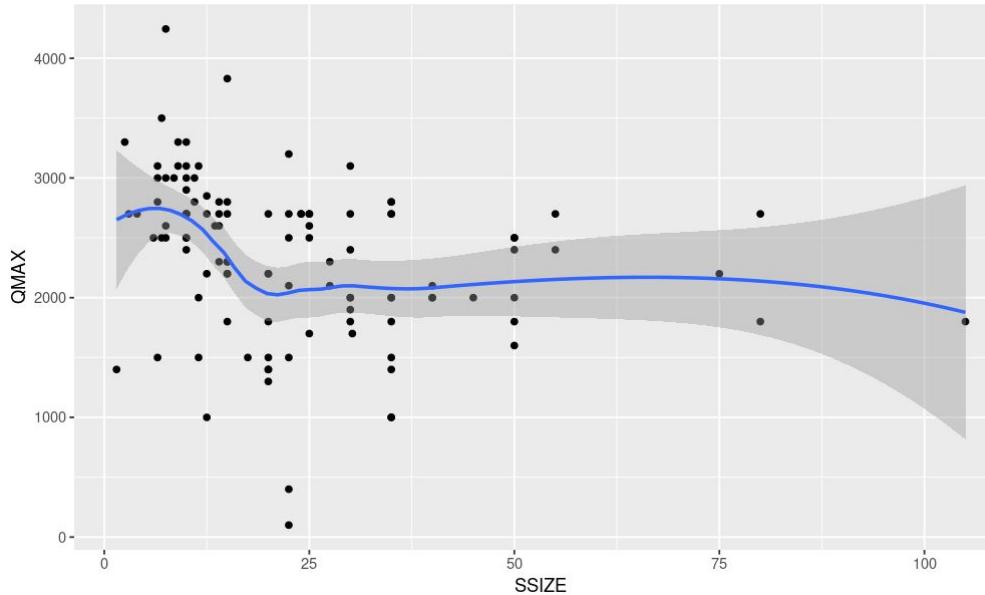


```
>ggplot(flowers_all, aes(x = FSIZE, y = SSIZE, color = ORD)) +  
  geom_point() + facet_grid( HBT~IUCN)
```



math on ggplot II - smooooooths

noisy scatters can be more readable adding smooths (trendlines), they use a specific layer that makes your figures nicer ... at it seems that biologists know math [joke, it is obvious that we don't (joke)]

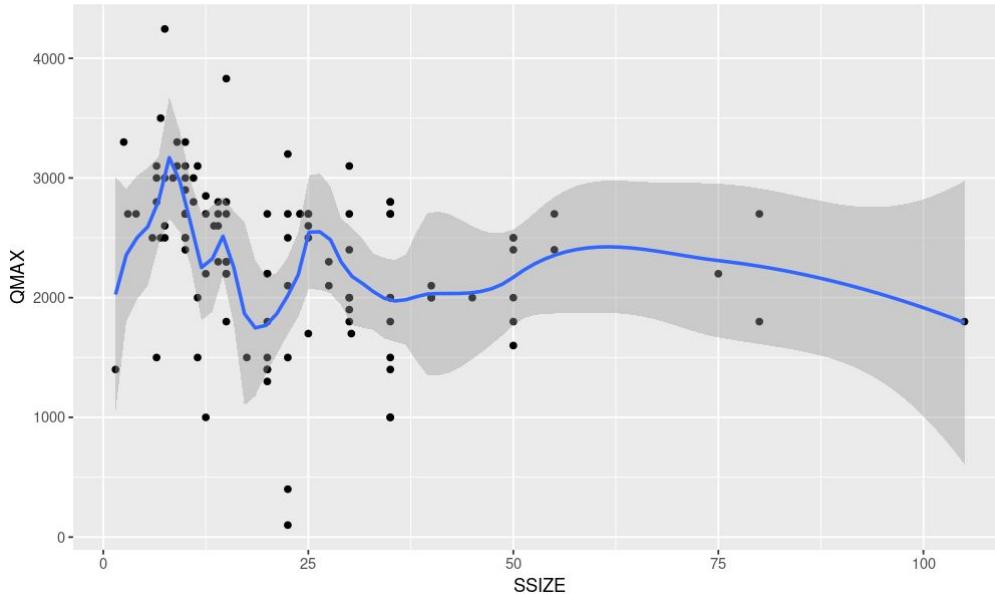


```
> ggplot(subset(flowers_all, SSIZE<200), aes(x = SSIZE, y = QMAX)) +  
  geom_point() + geom_smooth()  
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



math on ggplot II - smooooooths

line's approximation is controlled by the *span* parameter
it ranges from 0 (exceedingly wiggly) to 1 (not so wiggly)

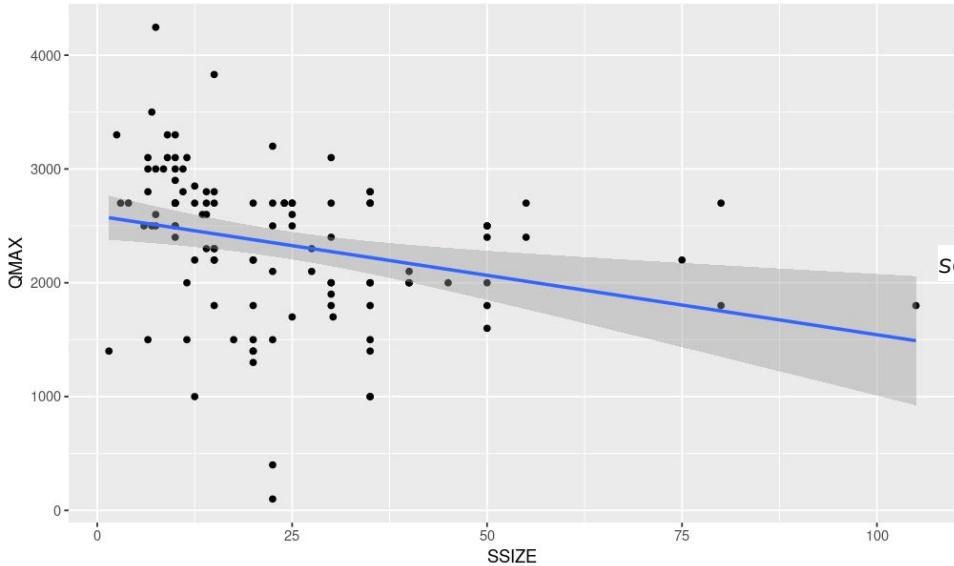


```
> ggplot(subset(flowers_all, SSIZE<200), aes(x = SSIZE, y = QMAX)) +  
  geom_point() + geom_smooth(span = 0.25)  
 `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



math on ggplot II - smooooooths

applied function can be changed with the *method* argument.
default = 'loess' it works for small datasets and locally approximate the smooth



useful methods are
'lm'
'gam'
...

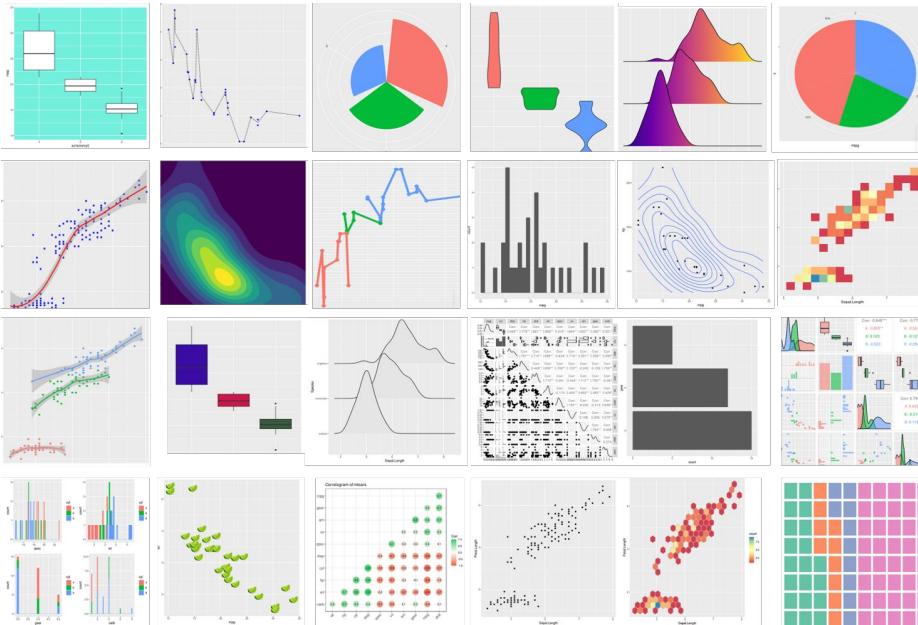
scatter meaning: plants growing at the higher elevations are smaller.
This is a linear ecological effect. It make sense uses the lm method

```
> ggplot(subset(flowers_all, SSIZE<200), aes(x = SSIZE, y = QMAX)) +  
  geom_point() + geom_smooth(method = "lm")  
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



graph types

until now we just see scatter plots, but with ggplot2 and its extensions (a sort of plugin packages) you can do almost every type of graphic!



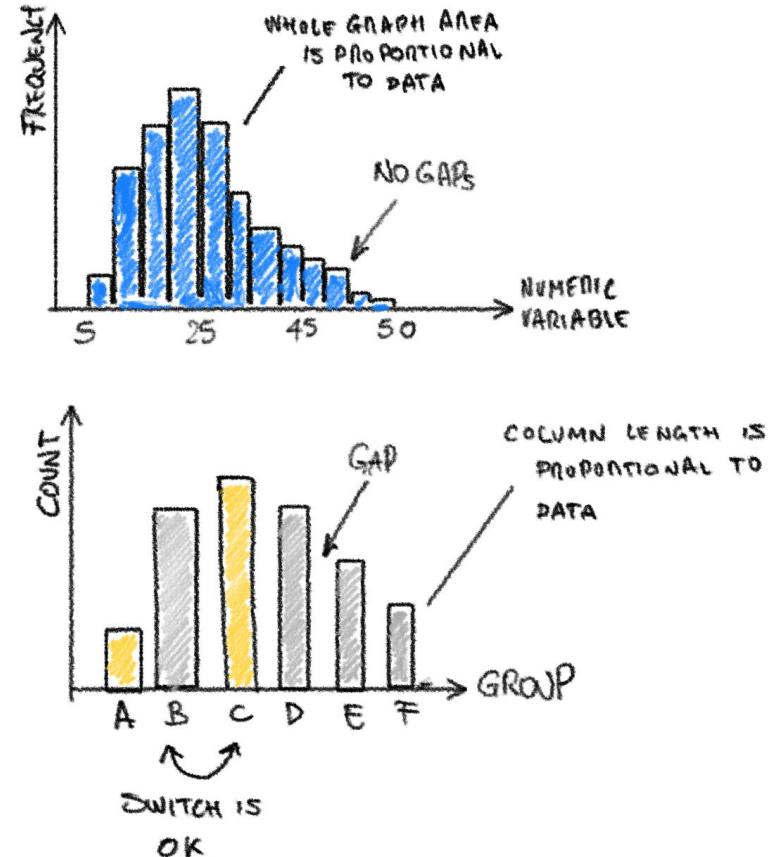
we are going to see:

- histograms
- barplots
- boxplots
- pies
- heatmaps



barplot vs histogram

	geom_hist()	geom_bar()
Meaning	Histogram refers to a graphical representation, that displays data by way of bars to show the frequency of numerical data.	Bar graph is a pictorial representation of data that uses bars to compare different categories of data
it indicates	Distribution of non-discrete variables along a gradient or as frequency classes	Comparison of discrete variables
it shows	Quantitative data	Categorical data

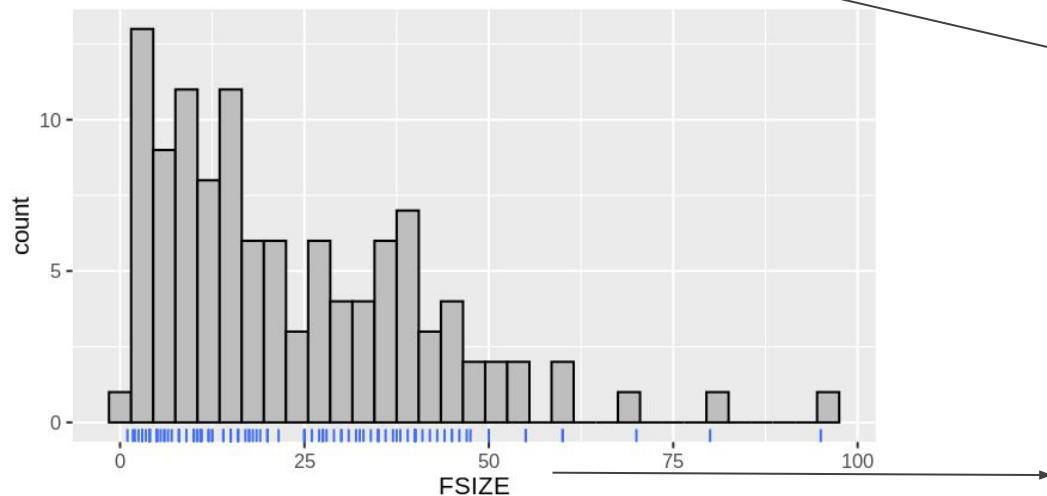




histograms with geom_histogram()

We need to examine frequencies of flower size across the dataset

```
> histogram<-ggplot(flowers_all, aes(x=FSIZE)) +  
  geom_histogram(binwidth = 3, fill="gray", colour="black")+      # add the histogram  
  geom_rug(color="royalblue2")                                     # add a rug plot
```

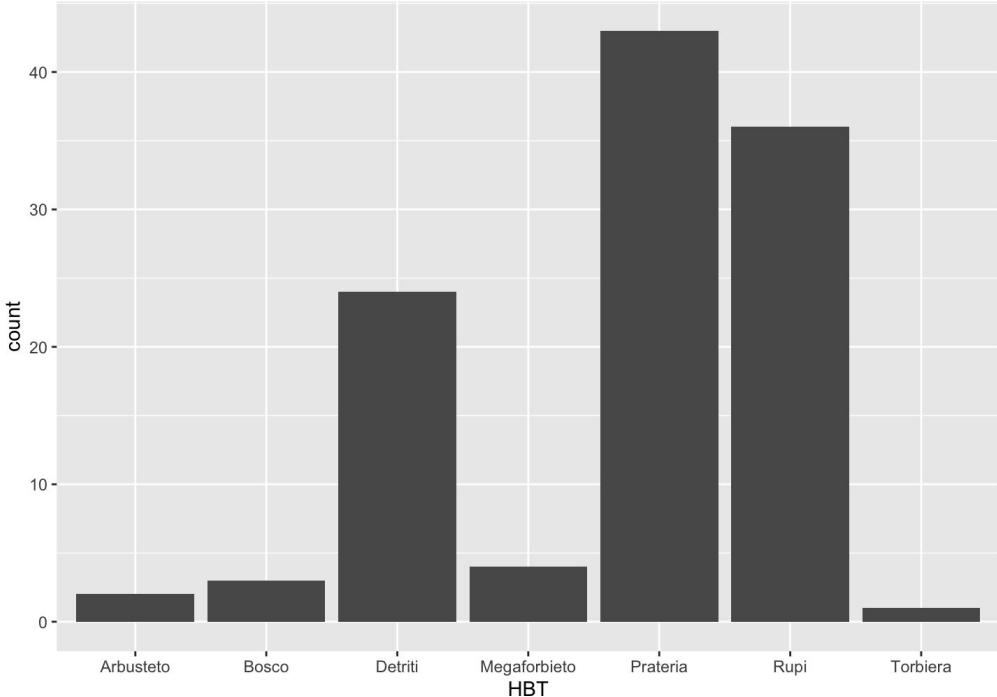


set the width of the range in which values are counted

The **rug plot** help to visualize where the density is higher. It is like an histogram but represents single data.



barplots



```
> barplot<-ggplot(flowers_all,  
aes(HBT)) +  
geom_bar()
```

The raw function automatically count the **recurrence** of the category in the dataset.



barplots

in other case data are pre-summarized into the dataset we have to specify:

the x values = group

the y value = pre-summarized variable

the **stat** = “identity” argument: `geom_bar(stat = “identity”)`

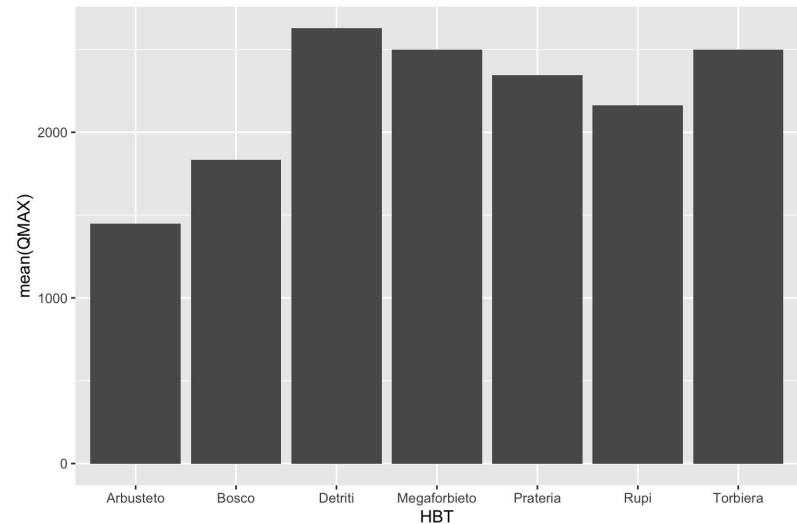
Exercise calculate QMAX means by habitat (HBT) and plot them as a barplot you would transform the dataset using a dplyr function



barplots

```
sumdata <- flowers_all %>% group_by(HBT) %>% summarize(mean(QMAX))  
  
ggplot(sumdata, aes(HBT, `mean(QMAX)`)) +  
  geom_bar(stat = "identity")
```

what happen if you don't specify the stat?

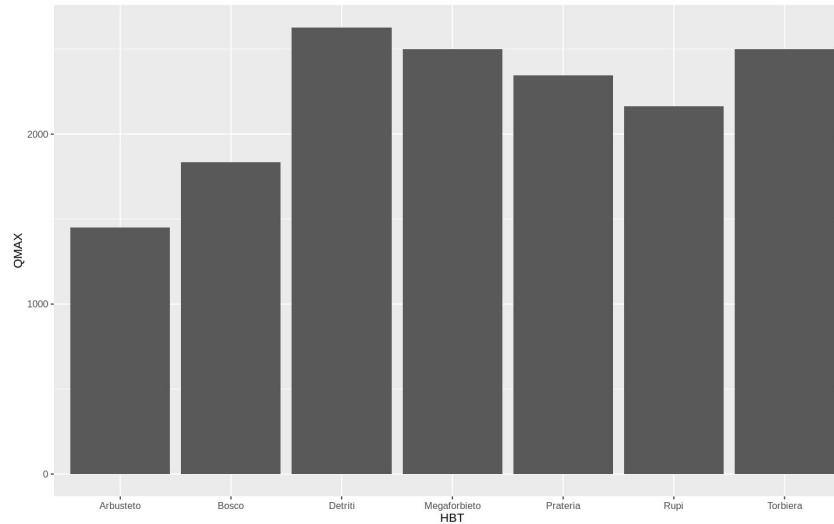




barplots

The short way

```
ggplot(flowers_all, aes(HBT, QMAX)) +  
  geom_bar(stat="summary", fun = "mean")
```

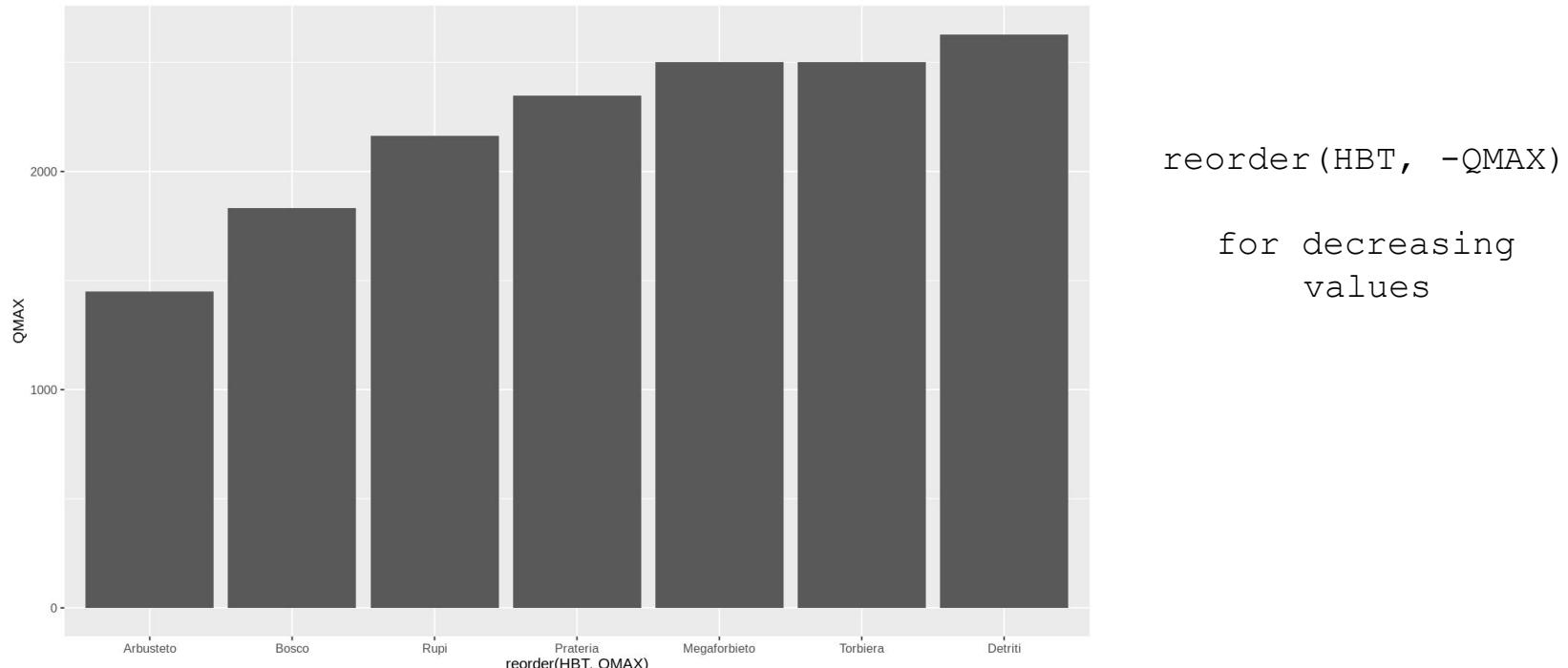




barplots

we can reorder categories on the x-axis by increasing QMAX 'on the fly'

```
ggplot(flowers_all, aes(reorder(HBT, QMAX), QMAX)) +  
  geom_bar(stat="summary", fun = "mean")
```





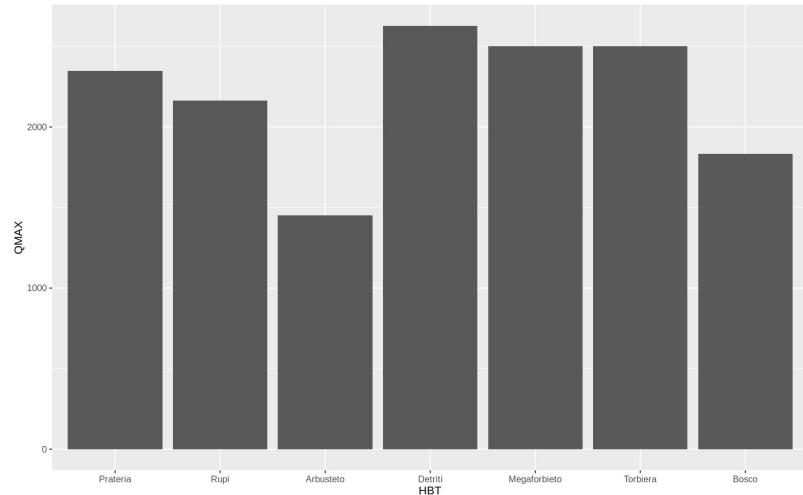
barplots

ggplot takes factor levels as input to plot categories.

If you need specific order of categories on your plot you have to re-level your factor.

```
> flowers_all$HBT<-factor(flowers_all$HBT,  
                           levels=c("Prateria", "Rupi", "Arbusteto", "Detriti", "Megaforbieto",  
                           "Torbiera", "Bosco"))
```

```
> ggplot(flowers_all, aes(HBT,QMAX)) +  
  geom_bar(stat="summary", fun = "mean")
```





Add errorbars using stat_summary()

```
> ggplot(flowers_all, aes(HBT, QMAX, fill = HBT)) +  
  geom_bar( stat="summary", fun = "mean") +  
  stat_summary(fun.data=mean_se, geom="errorbar", color="gray40", width=0.2, na.rm = T) +  
  stat_summary(fun="mean", geom="point", colour="red")
```

most common variables values to set:

fun.data/fun:

mean_se	(standard error)
mean_sdl	(standard dev)
mean_cl_boot	(nonparametric bootstrapped CI)
mean_cl_normal	(parametric CI)

min

max

mean

geom:

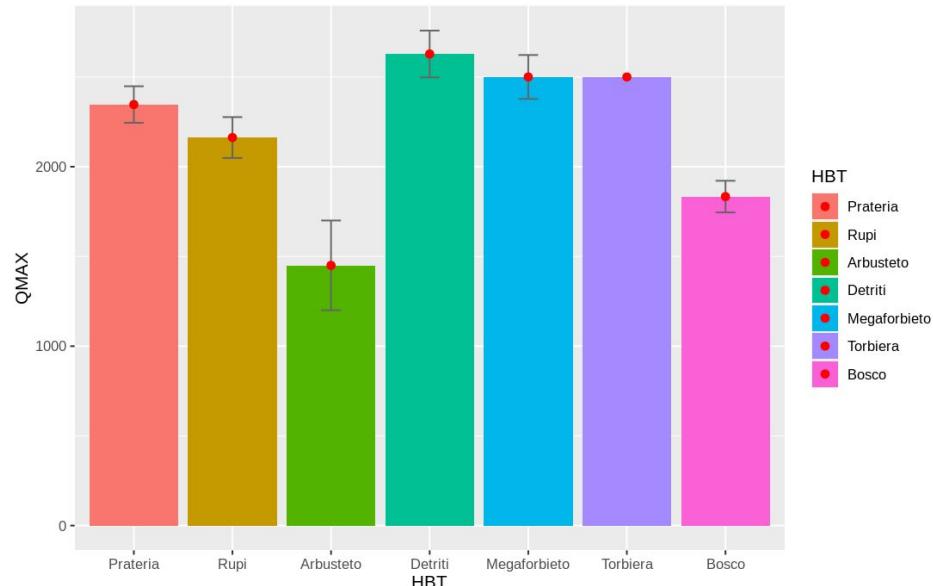
errorbar

point

pointrange

linerange

crossbar

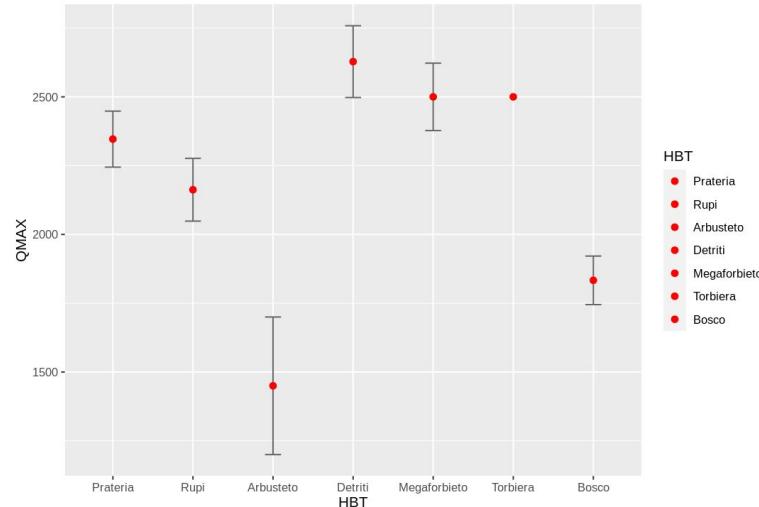




Add errorbars using stat_summary()

```
> ggplot(flowers_all, aes(HBT, QMAX, fill = HBT)) +  
# geom_bar( stat="summary", fun = "mean") +  
stat_summary(fun.data=mean_se, geom="errorbar", color="gray40", width=0.2, na.rm = T) +  
stat_summary(fun="mean", geom="point", colour="red")
```

If you remove geom_bar() element you obtain an **errorplot**





grouped barplots()

We want to compare QMAX in Prateria and Detriti Habitats (HBT) across different UMD types

I want to group bars by UMD variable: use **group** aesthetic

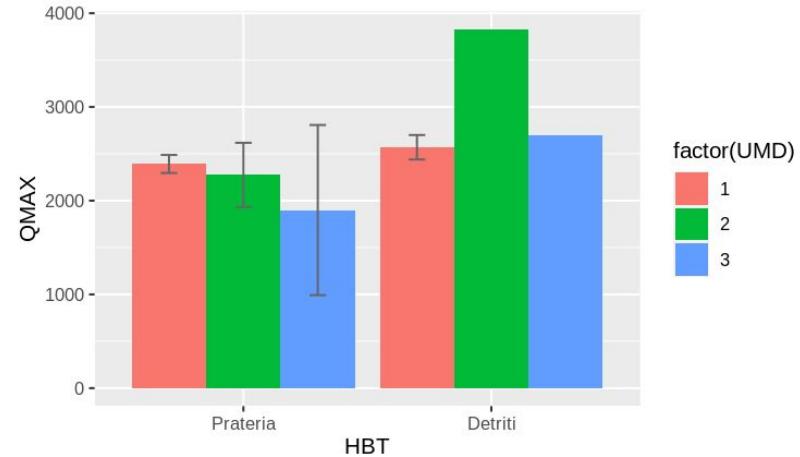
Bars must be “dodged”

use **position=position_dodge()**

```
> ggplot(  filter(flowers_all, HBT %in% c("Detriti", "Prateria")),
    aes(HBT, QMAX, fill = factor(UMD), group=factor(UMD))) +
```

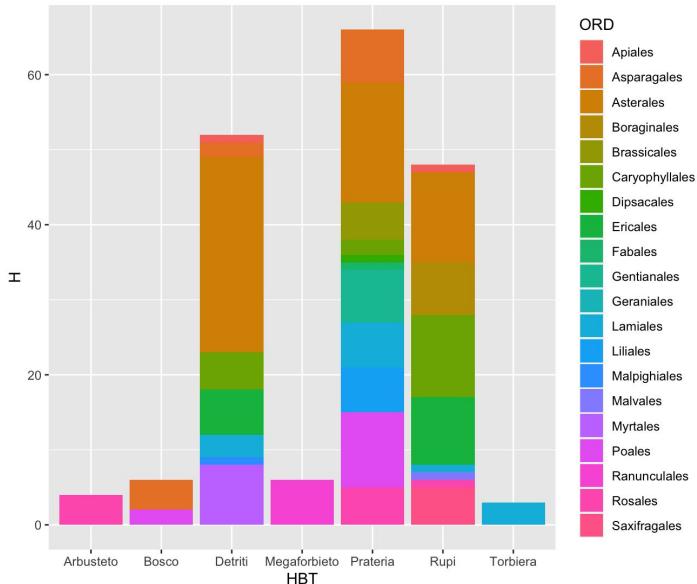
```
  geom_bar( stat="summary", fun = "mean", position=position_dodge(width=0.9))+
```

```
  stat_summary(fun.data=mean_se,
    geom="errorbar",
    position=position_dodge(width=0.9),
    color="gray40", width=0.2, na.rm = T)
```





stacked barplots



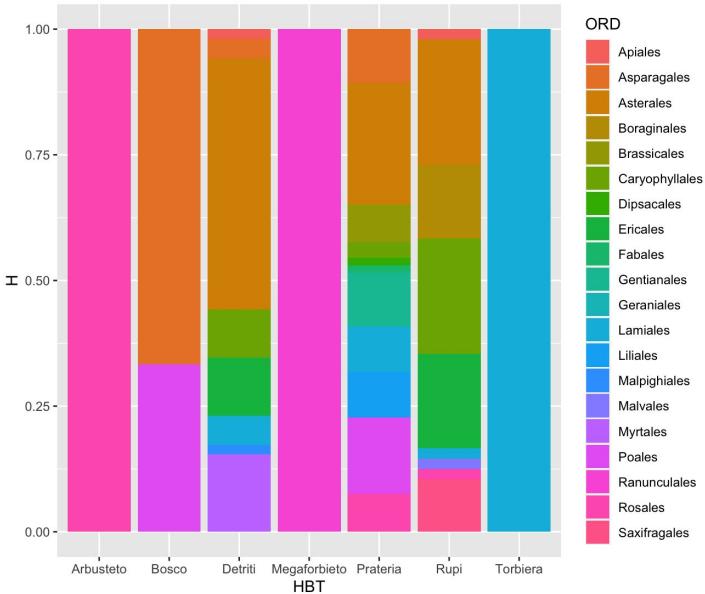
stacked barplots can be created simply specifying to which category bars should be colored.

They can correspond to absolute values

```
ggplot(flowers_all, aes(x = HBT, y = H, fill = ORD)) +  
  geom_bar(stat = "identity")
```



stacked barplots



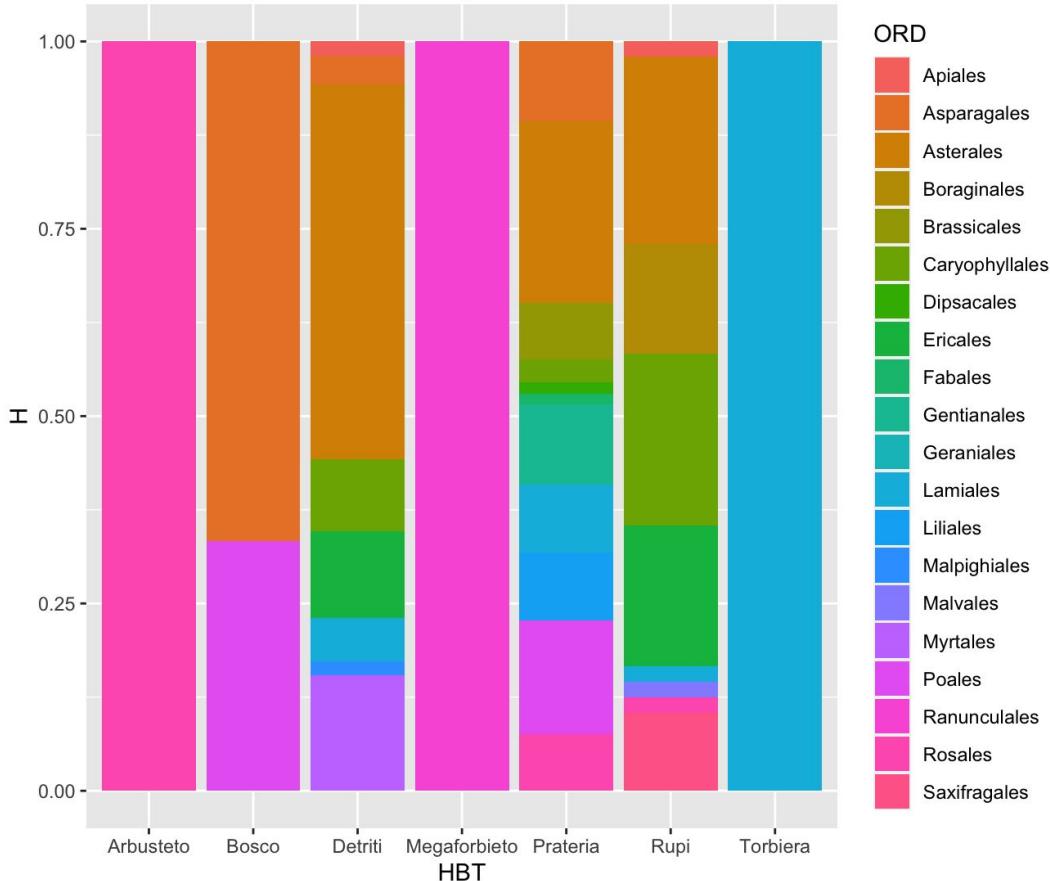
stacked barplots can be created simply specifying to which category bars should be colored.

They can correspond to absolute values ... or to relative values

```
ggplot(flowers_all, aes(x = HBT, y = H, fill = ORD)) +  
  geom_bar(stat = "identity", position = "fill")
```

stacked barplots

the palette 😞



when groups are more than 5 or 6 it starts to be problematic to use the default ggplot2 palette

colors start to be similar each other and graph readability decrease

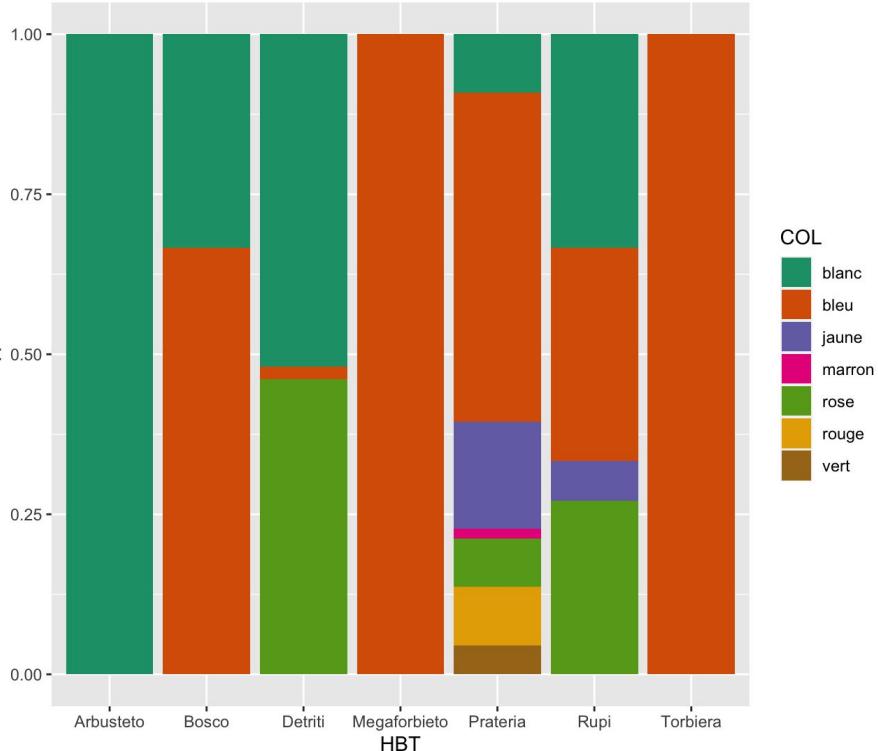
! ggplot2 default palette is not colorblind friendly !

it must be changed



stacked barplots :: the palette

<https://r-graph-gallery.com/38-rcolorbrewers-palettes.html>



Set ggplot color manually:

scale_fill_manual() for box plot, bar plot,...
scale_color_manual() for lines and points

Use RColorBrewer palettes:

scale_fill_brewer()
scale_color_brewer()

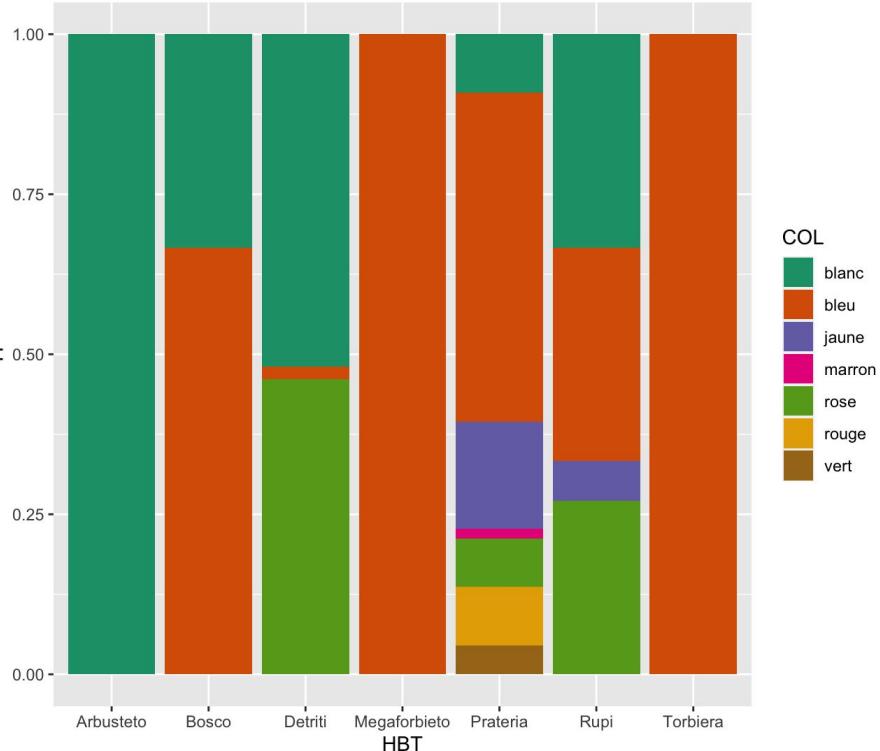
```
ggplot(flowers_all, aes(x=HBT, y = H, fill = COL)) +  
  geom_bar(stat = "identity", position = "fill") +  
  scale_fill_brewer(palette = "Dark2")
```

Dark2 is simply a vector of colors!



stacked barplots :: the palette

<https://r-graph-gallery.com/38-rcolorbrewers-palettes.html>



colors in R could be defined in 2 main ways:

- by name following a specific nomenclature
- by HEX (hexadecimal) code

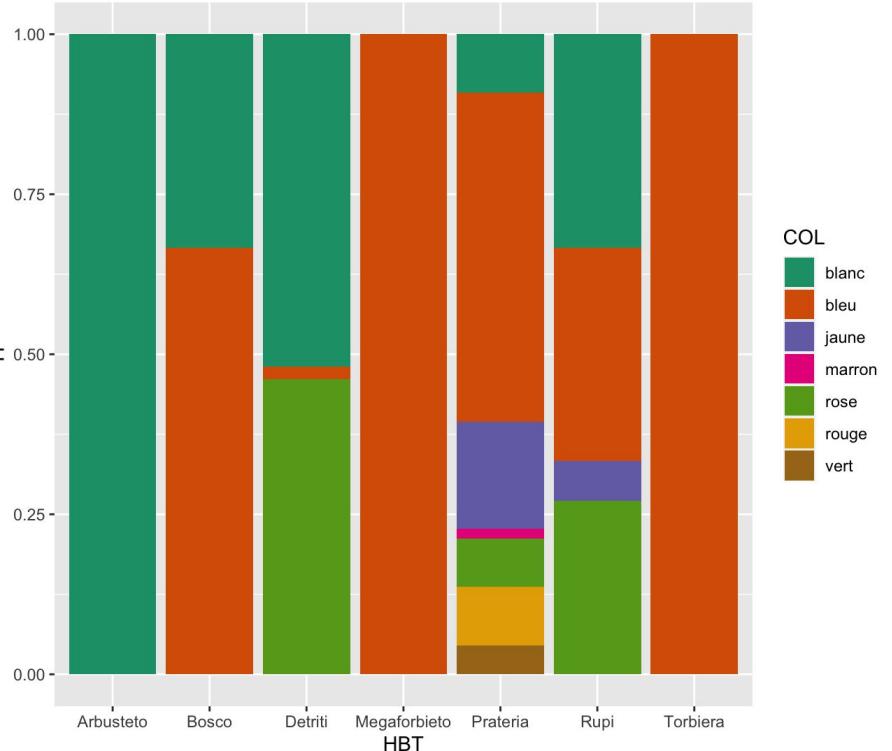
"blue" == "#0000ff"

... and so on ...

```
ggplot(flowers_all, aes(x=HBT, y = H, fill = COL)) +  
  geom_bar(stat = "identity", position = "fill") +  
  scale_fill_brewer(palette = "Dark2")
```



stacked barplots :: the palette

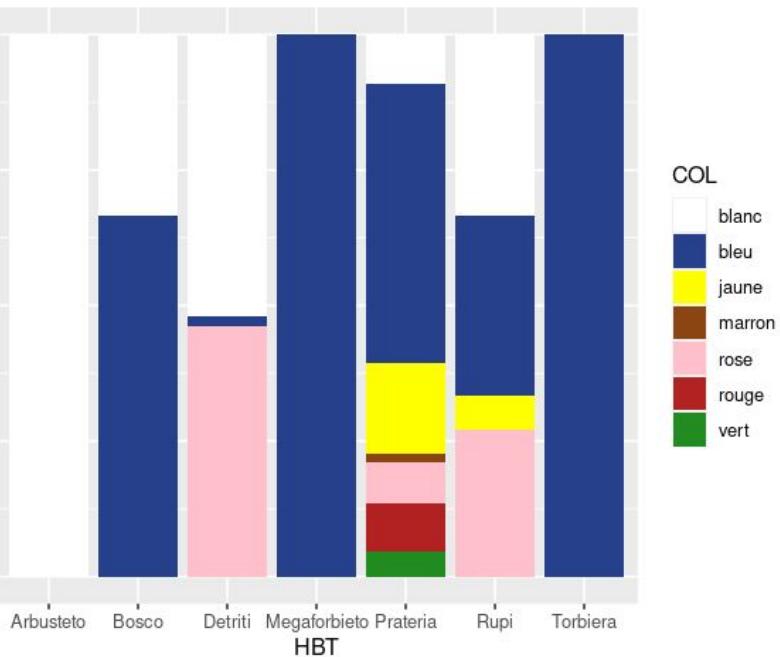


Exercise 2:

It is clear that this legend make no sense. Create a proper palette and apply it on the barplot!

COL	
blanc	green
bleu	orange
jaune	purple
marron	pink
rose	yellow
rouge	brown
vert	teal

stacked barplots :: the palette



Exercise (my) solution:

```
# create vector of color (ordered)
mypalette <- c(    "blanc"="white",
                 "rose"="pink",
                 "vert"="forestgreen", "bleu"="blue",
                 "jaune"="gold"
                 "marron"="chocolate4"
                 "rouge"="red")

# plot graph
ggplot(flowers_all, aes(x=HBT, y = H, fill = COL)) +
  geom_bar(stat = "identity", position="fill") +
  scale_fill_manual(values = mypalette)
```



pie charts

The long, but easier method

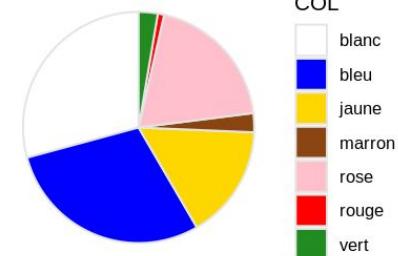
Pies are special type of barplots (1 column barplot with a round coordinate system) with only 1 aesthetic: **fill**

We need to generate a pie chart visualizing the proportions of flower colors among the species in our dataset.

```
# summarize color counts
```

```
> col_counts<-count(flowers_all, COL) %>%  
  mutate(prop=prop.table(n), label=str_c(round(prop.table(n)*100, 1), "%"))
```

```
> pie<-ggplot(col_counts, aes(x="", y=prop, fill=COL)) +  
  geom_bar(stat="identity", position=position_fill(), color="gray90") +  
  scale_fill_manual(values=mypalette)+  
  coord_polar("y", start=0) + theme_void()
```





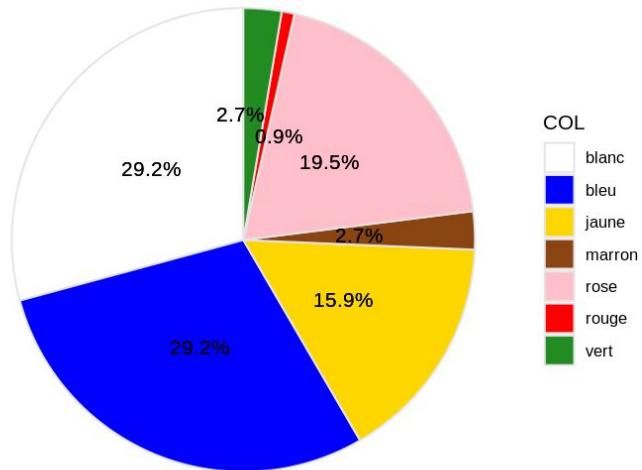
pie charts

The long, but easier method

#Add labels

```
library(ggrepel)
```

```
> pie<- pie + geom_text_repel(aes(label = label), position=position_fill(0.5), force = 0.002)
```



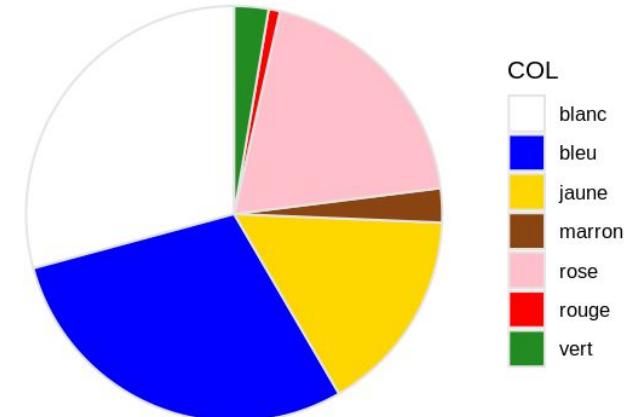


pie charts

The “on-the-fly” method

color counts as summarized directly by geom_bar() function using stat="count" (as for barplots)

```
> pie<-ggplot(flowers_all, aes(x="",fill=COL)) +  
  geom_bar(stat="count", position="fill", color="gray90") +  
  scale_fill_manual(values=mypalette)+  
  coord_polar("y", start=0) + theme_void()
```



seems easier...but if you need to put labels..you have to swear! Let's see..



pie charts

The “on-the-fly” method

```
> pie +
```

```
  geom_text_repel(
```

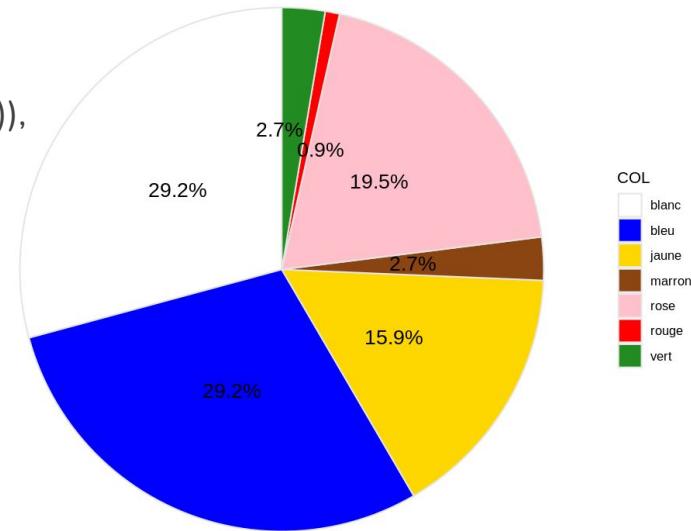
```
    stat='count',
```

```
    aes(y=after_stat(..count..),
```

```
    label=after_stat(scales::percent(..count../sum(..count..),accuracy=0.1))),
```

```
    position=position_fill(0.5), force = 0.002
```

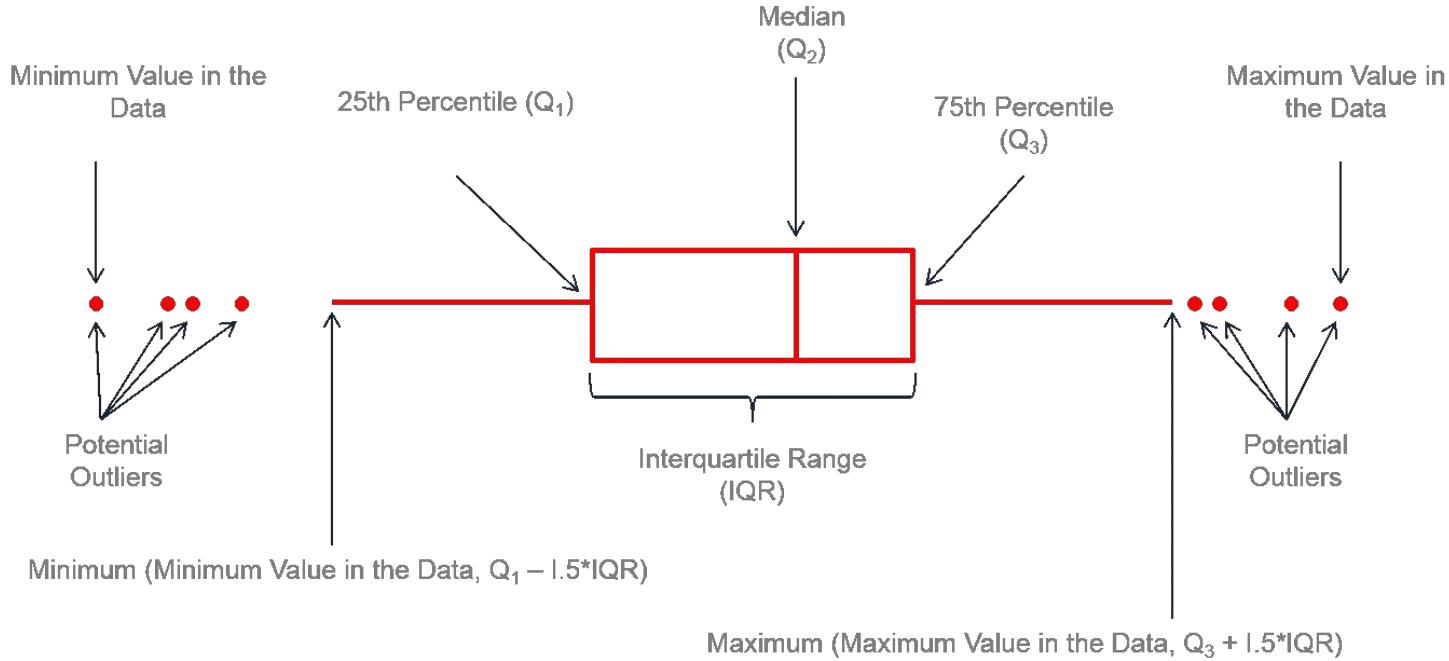
```
)
```



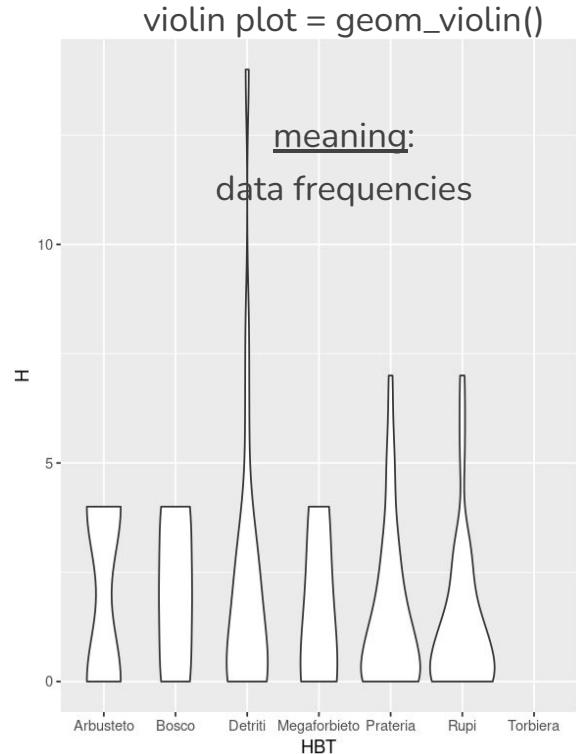
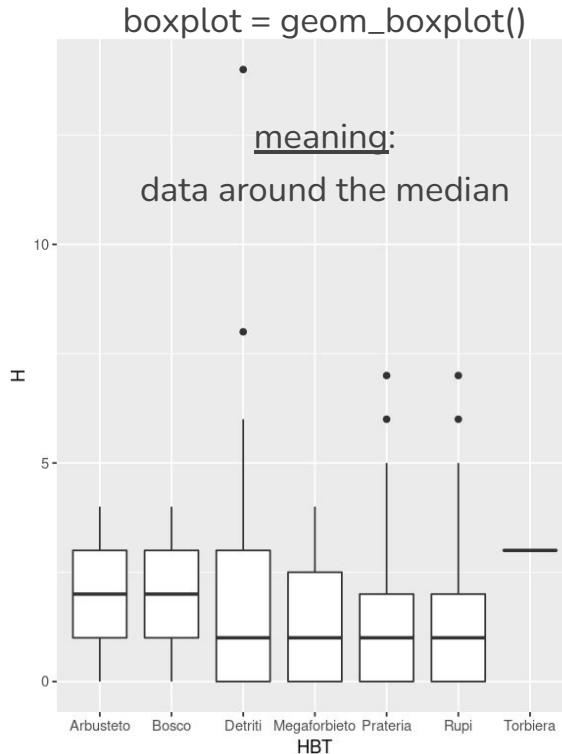
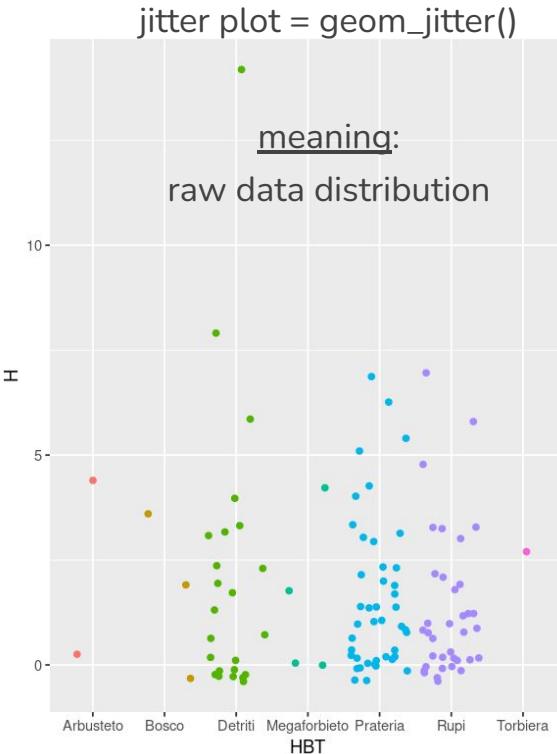


boxplot and friends

Boxplots are good tools to describe how data are distributed around the median value



boxplot and friends

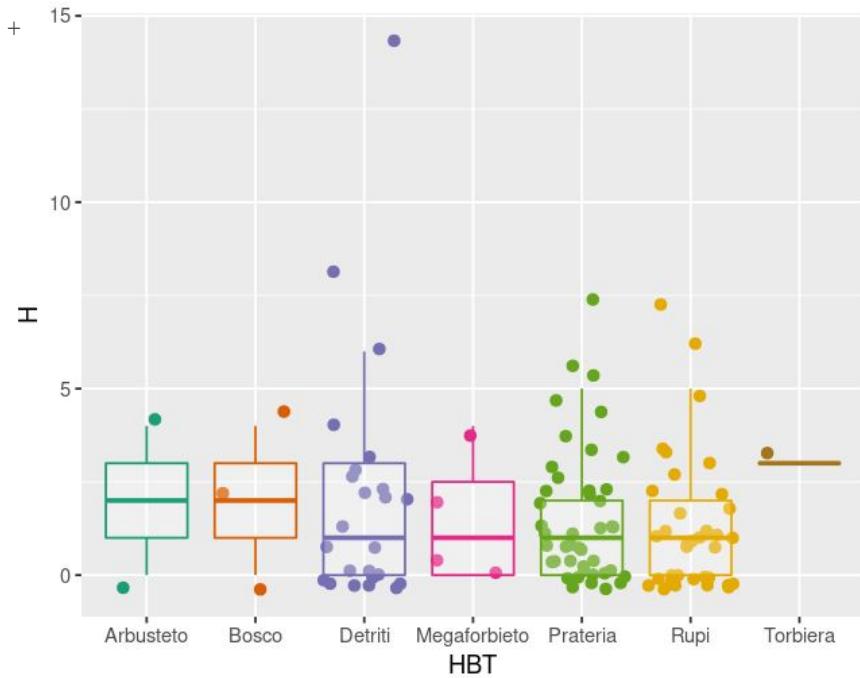




boxplot and friends

ggplot2 works with **layers**, so we can overlap several visualizations!

```
boxplot<-ggplot(flowers_all, aes(x=HBT, y = H, color = HBT)) +  
  geom_jitter(size = 2) +  
  geom_boxplot(alpha = 0.25, shape = 20, outlier.shape = NA) +  
  scale_color_brewer(palette = "Dark2") +  
  theme(legend.position = "none")
```

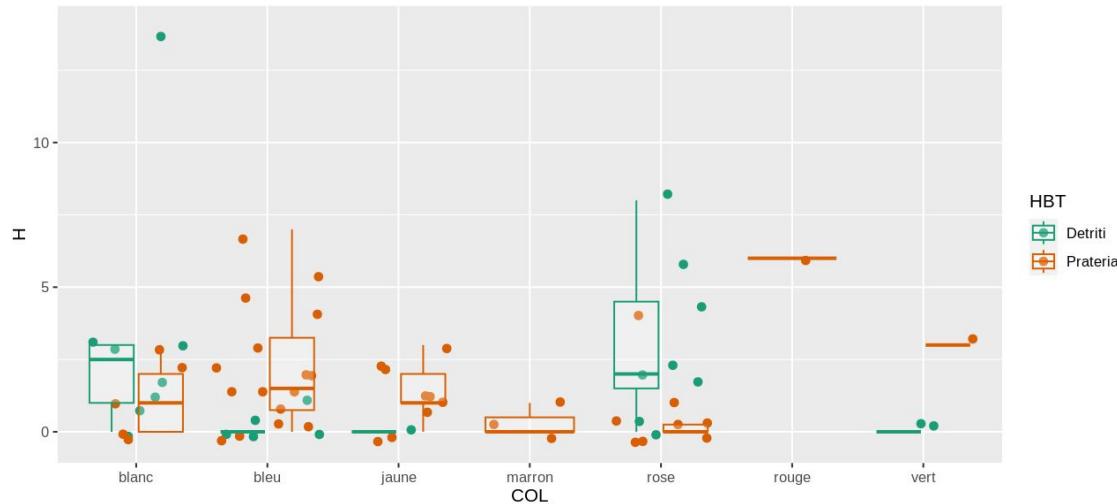




grouped boxplots

```
> ggplot(filter(flowers_all, HBT %in% c("Prateria", "Detriti")), aes(x=COL, y = H,  
colour=HBT)) +  
  
geom_jitter(size = 2) +  
  
geom_boxplot(alpha = 0.25, shape = 20, outlier.shape = NA, show.legend=T) +  
  
scale_color_brewer(palette = "Dark2")
```

What if color aesthetic is set as
the x?





Pre-loaded themes

theme_bw(): a variation on theme_grey() that uses a white background and thin grey grid lines.

theme_linedraw(): a theme with only black lines of various widths on white backgrounds, reminiscent of a line drawing.

theme_light(): similar to theme_linedraw() but with light grey lines and axes, to direct more attention towards the data.

theme_dark(): the dark cousin of theme_light(), with similar line sizes but a dark background. Useful to make thin coloured lines pop out.

theme_minimal(): a minimalist theme with no background annotations.

theme_classic(): a classic-looking theme, with x and y axis lines and no gridlines.

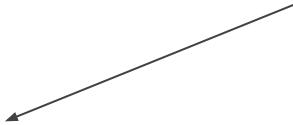
theme_void(): a completely empty theme.



manual set theme elements

```
plot + theme(element.name = element_function() )
```

specify what you wanna modify



plot.title

axis.title.y

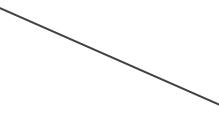
plot.background

panel.background

panel.grid.major

legend.position

they are legions!



FOUR basic element_function
types:

element_text()

element_line()

element_rect()

element_blank()

but also other
available!

element_markdown()

...



manual themes

```
plot + theme_ggplot  
  
theme_ggplot <- theme(  
  legend.position = "none",  
  axis.title = element_text(size = 12),  
  axis.text.x = element_text(size = 11),  
  axis.text.y = element_text(size = 11),  
  panel.grid = element_blank(),  
  plot.caption = element_text(size = 10, color = "gray50"),  
  plot.title = element_text(face="bold", size=12))
```

Elements to modify are many!

I use to create a standard ggplot theme that I apply to all my outputs

then I can do minor editing

waste time where worth!



other relevant theme-related functions

Function to append to ggplot

- `ggtitle()` adds title
- `xlab() / ylab()` modify x and y axes labels
- `scale_color_manual()` manually modify color palette
- `scale_fill_manual()` manually modify fill palette
- `scale_y_continuous` |
- `scale_x_continuous` | set axis graphical parameters
- `scale_y_discrete` | and adjust tick scale
- `scale_x_discrete` |



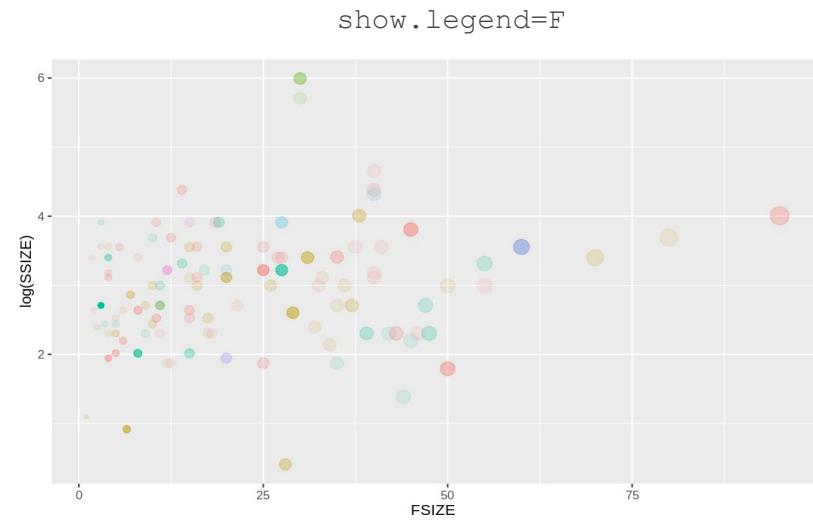
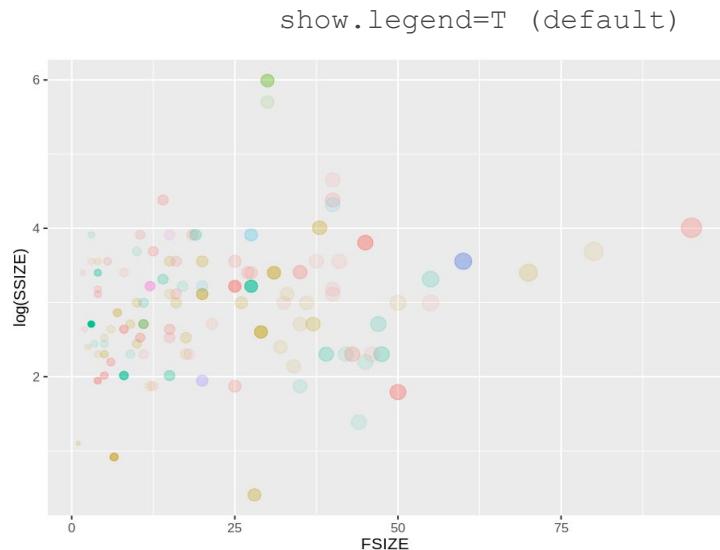
Working with legends

Legends always require customisation

You may need to remove legends for certain layers since they may be redundant.

```
# let's plot a simple flower plot
```

```
> plot_legend<-ggplot(flowers_all)+geom_point(aes(y = log(SSIZE), x = FSIZE, color=HBT,  
size=FSIZE, alpha=H), show.legend =T)
```





Working with legends

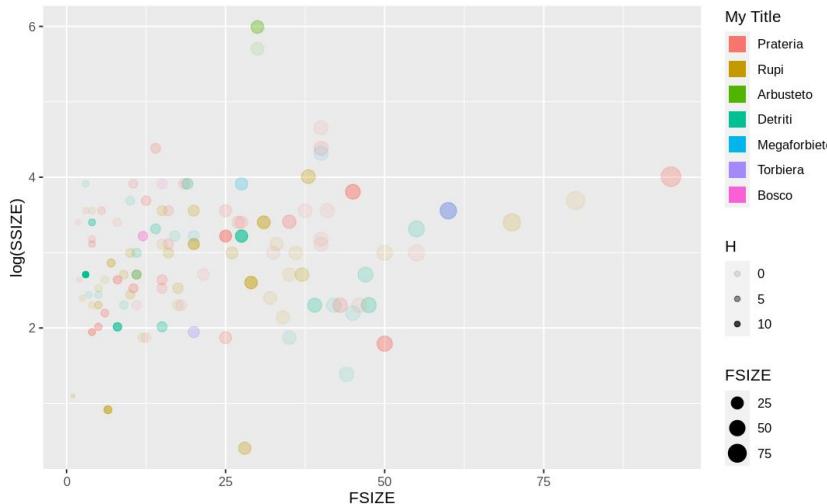
Legends always require customisation

Remove legend for some aesthetics. We need to use the **guides()** function

```
> plot_legend + guides(color="none")# removes legend for colors
```

guides() works with **guide_legend()** function

```
> plot_legend + guides(color= guide_legend(title="My Title", order=1,
                                             override.aes = list(size=5, shape=15)))
```





Working with legends

```
# Extract legends from plot  
  
> only_legend<-lemon::g_legend(plot_legend)      # extract legend from plot object  
  
> plot(only_legend)  
  
# Useful when assembling panels, you may need flexibility to organize legends
```



ggplot2: extension libraries

A number of additional libraries are available for **ggplot2** to extend its functionalities (new graphs type, graphic features, themes, color palette, statistics...):

Some examples:

ggbubr statistics and graphical parameters, simplified grammar

ggfortify simplified ggplot2 grammar

ggraph network visualization

ggtern implements ternary plots

lemon extended graphic functionalities

ggh4x The package extends 'ggplot2' facets through customisation

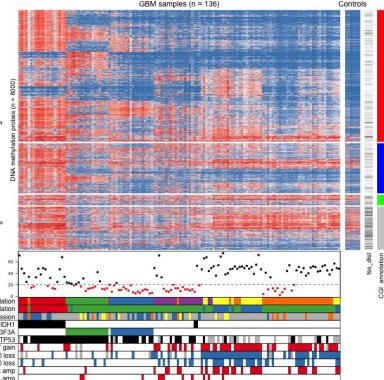
ggsci palette collections

Most of functions in these packages are well documented and use similar parameters to ggplot2



Other worth graphic libraries in R

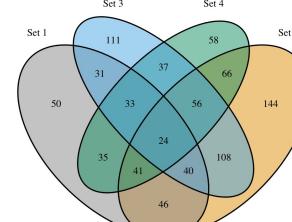
complexHeatmap state-of-the-art heatmap plotting



UpSetR upset plots (intersections)

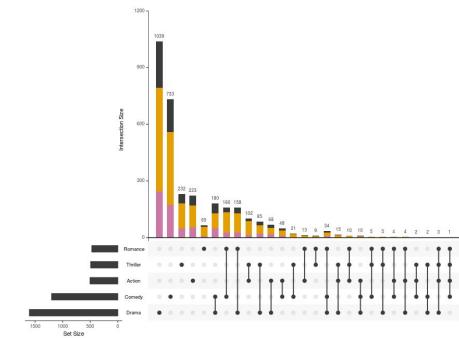
VennDiagram Venns

ggvenn Venns



ggVennDiagram Venns

ggalluvial alluvial plots

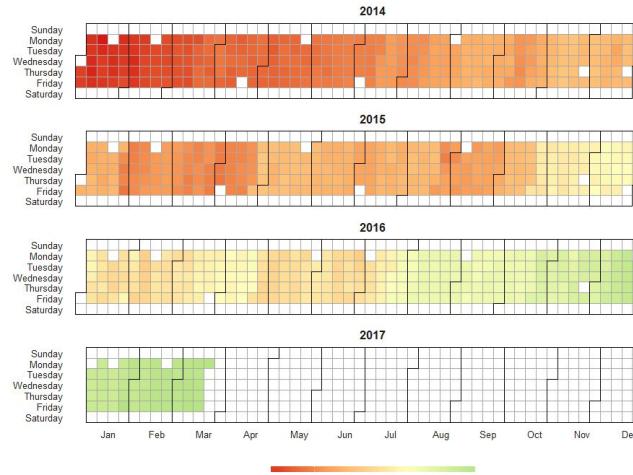
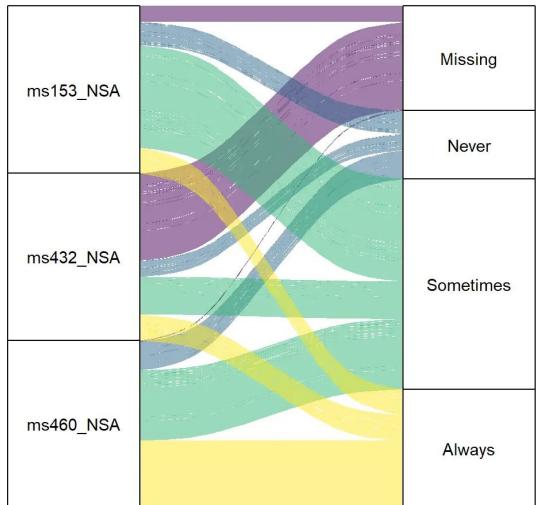


ggtern ternary plots

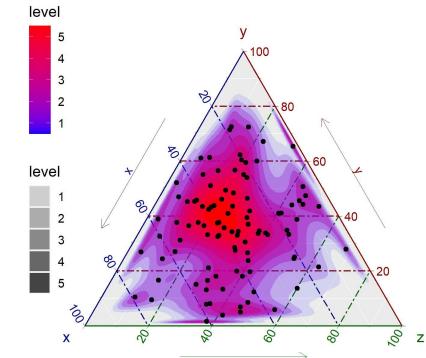
ggraph networks

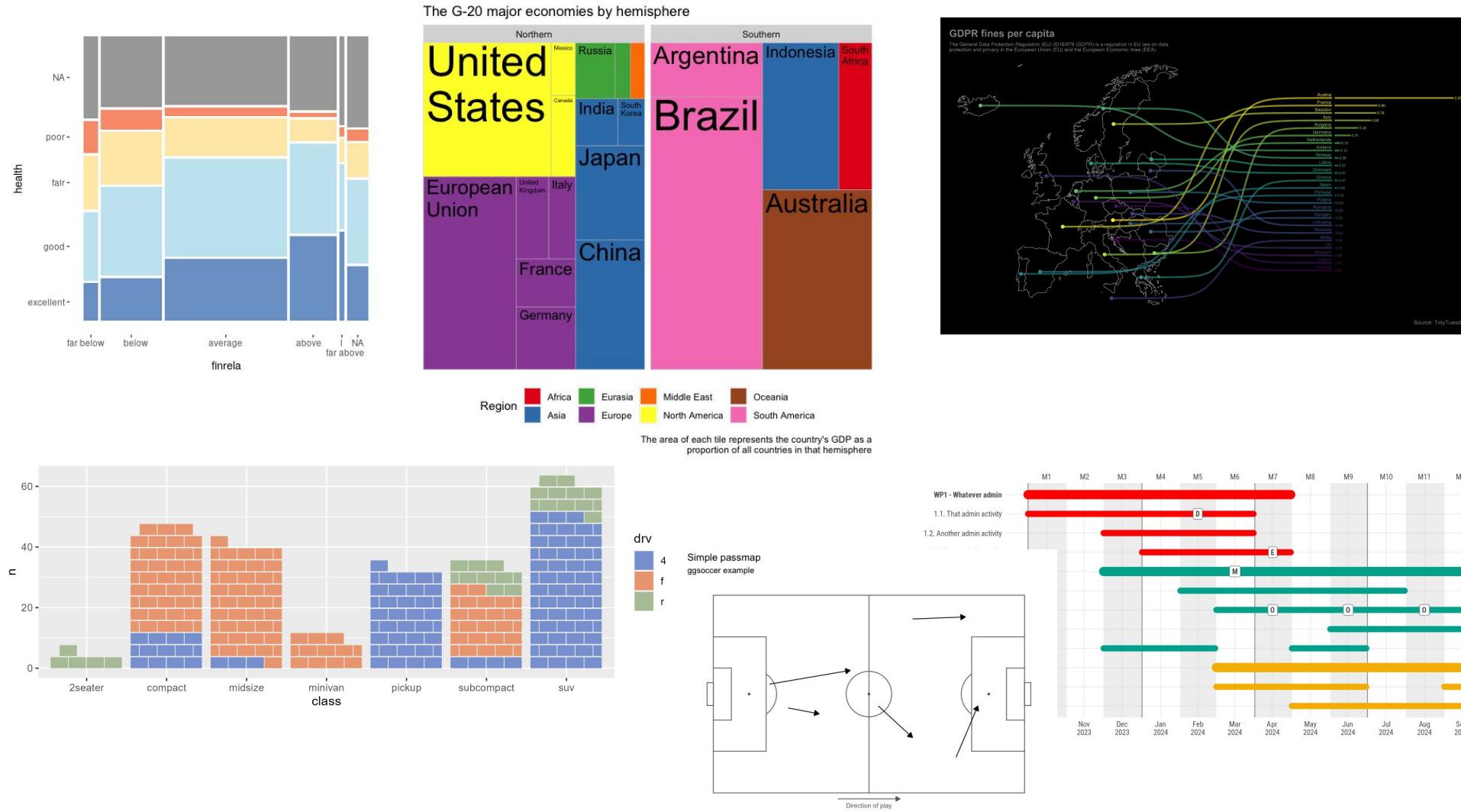
ggrepel manage overlapping text labels

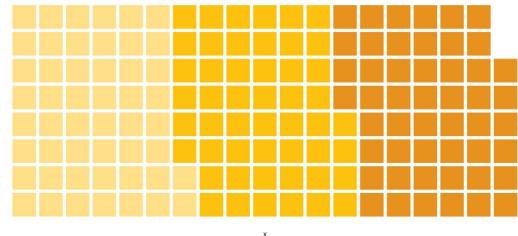
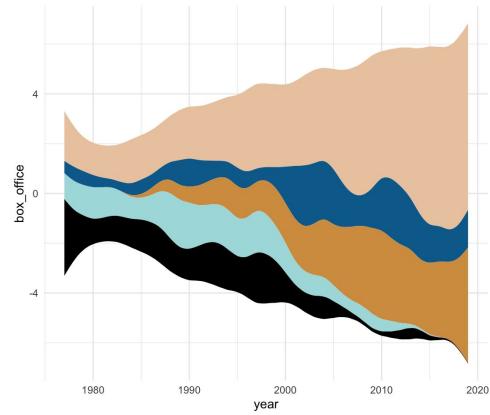
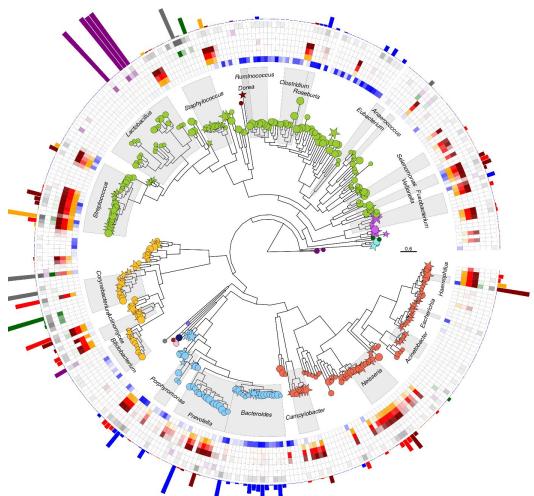
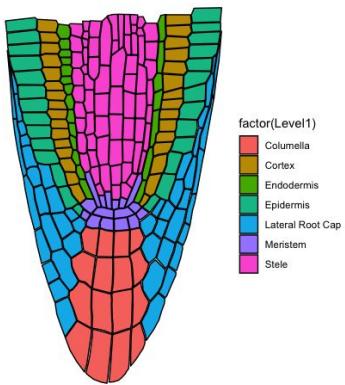
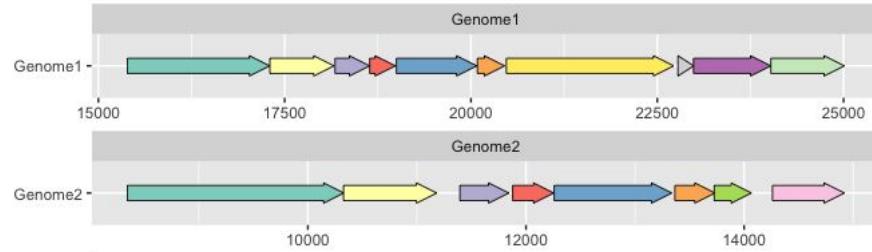
ggtree/ggtreeExtra complex phylogenetic trees



Example Density/Contour Plot







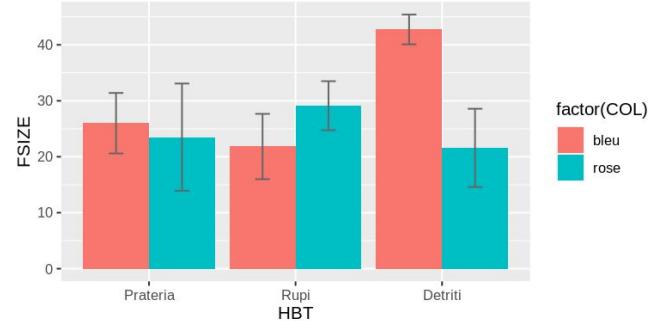


add statistics to a ggplot2

We create a plot containing only:

- blue and rose flowers
- species who live in Prateria, Rupi and Detriti
- species of the following biological forms: H scap, G bulb and H rose

```
>blue_rose<-ggplot(filter(flowers_all, COL %in% c("bleu", "rose") &
  HBT %in% c("Prateria", "Rupi", "Detriti") &
  FBIO %in% c("H scap", "G bulb", "H ros")),
  aes(HBT, FSIZE, fill = factor(COL), group=factor(COL))) +
  geom_bar(stat="summary", fun = "mean", position=position_dodge(width=0.9)) +
  stat_summary(fun.data=mean_se, geom="errorbar", position=position_dodge(width=0.9),
  color="gray40", width=0.2, na.rm = T)
```





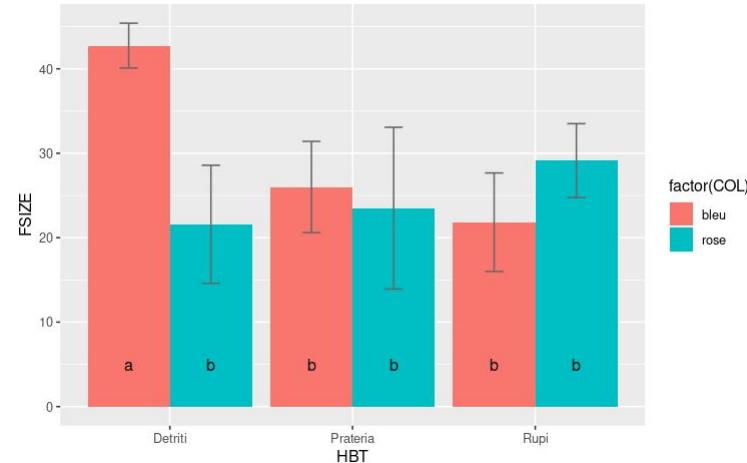
add statistics to a ggplot2 manually

```
# We create a data frame containing staticics values
```

```
> stat_FSIZE<-data.frame(HBT=c("Detriti", "Detriti", "Prateria", "Prateria", "Rupi", "Rupi"),
  COL=rep(c("bleu", "rose"),3),
  group=c("a", rep("b",5)))
```

```
# plot letters with geom_text()
```

```
> blue_rose+geom_text(data = stat_FSIZE, aes(HBT, y=5, group=COL, label=group),
position=position_dodge(width=0.9))
```





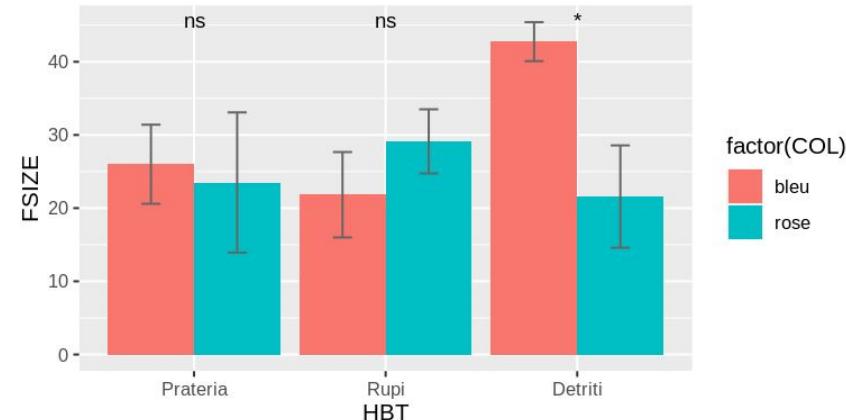
add statistics to a ggplot2 plot using ggpubr

Are blue flowers significantly bigger in those three habitats?

```
>library(ggpubr)
```

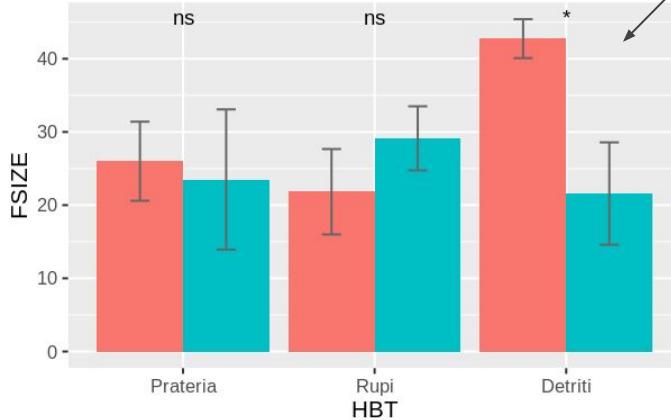
label "psignif" or "p.format"
method "t.test" or "kruskal.test"

```
> blue_rose + stat_compare_means(    label = "p.signif",
                                    method = "kruskal.test",
                                    hide.ns = F, na.rm=T,
                                    size=3.5,
                                    label.y.npc = c(0.8)    )
```





Exercise



add a title (centered)

change the theme and remove
the background grid

Modify the legend title

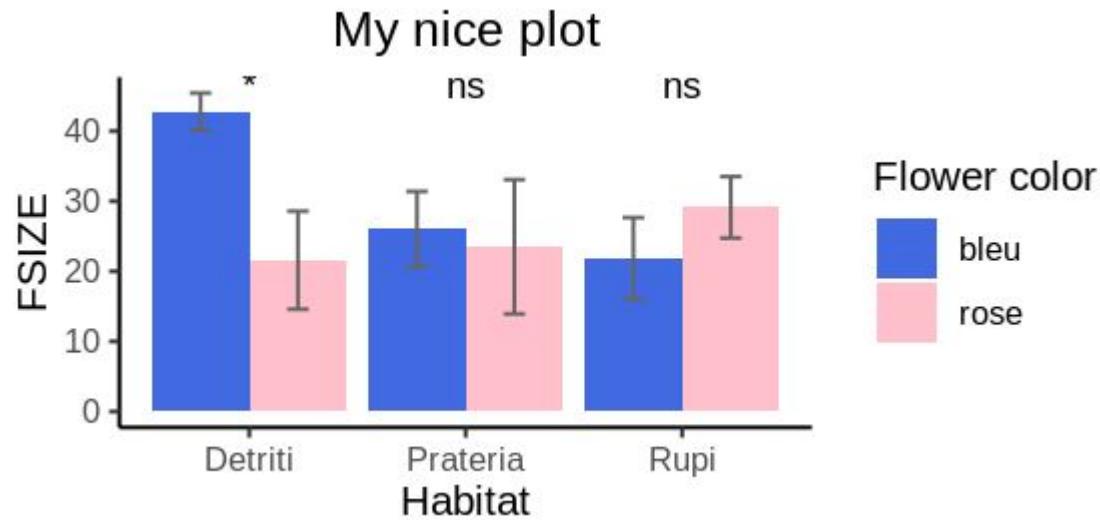
factor(COL)
bleu
rose

Use appropriate color
palette

change x-axis title
into "Habitat"



Exercise





Title position

```
> plot_stats <- plot_stats+  
  ggtitle("My nice plot") +  
  scale_fill_manual(values=c("royalblue", "pink")) +  
  guides(fill=guide_legend(title="Flower color")) +  
  xlab("Habitat") +  
  theme_classic() +  
  theme(plot.title = element_text(hjust=0.5))
```



Venn Diagrams

```
# First, you need to obtain vectors of categories you want to intersect  
  
# Most of the function which generate Venns in R require a list of vectors.  
  
# As an example we want to study the the overlap between plant orders (ORD) across  
"Prateria", "Rupi" and "Detriti" habitats (HBT)  
  
# Create a list  
  
venn_list<-list(    "Detriti" = flowers_all[flowers_all$HBT=="Detriti",]$ORD,  
  
                    "Prateria"= flowers_all[flowers_all$HBT=="Prateria",]$ORD,  
  
                    "Rupi"    = flowers_all[flowers_all$HBT=="Rupi",]$ORD )
```

You should also be able to use dplyr::**filter()**



Venn Diagrams

```
# First, you need to obtain vectors of categories you want to intersect  
  
# Most of the function which generate Venns in R require a list of vectors.  
  
# As an example we want to study the the overlap between plant orders (ORD) across  
“Prateria”, “Rupi” and “Detriti” habitats (HBT)  
  
# Create a list  
  
venn_list<-list(    "Detriti" = filter(flowers_all, HBT=="Detriti")$ORD,  
  
                    "Prateria"= filter(flowers_all, HBT=="Prateria")$ORD,  
  
                    "Rupi"= filter(flowers_all, HBT=="Rupi")$ORD )
```



Venn Diagrams

Now you're almost ready to create your Venn Diagram

Exercise!

Create your first Venn using the `ggvenn()` function from package `ggvenn`

Tips

- 1) This is the first time you view this function, if you have doubts on how to use it, please read the [function documentation!](#) Most of the time the solution is simpler than it seems.

- 2) First, you can skip to set all the arguments of a function. Most of them have already **defaults** that allow you to use the function out-of-the-box.



Venn Diagrams

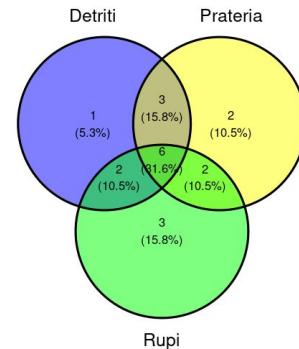
Now you're almost ready to create your Venn Diagram

Exercise!

Create your first Venn using the **ggvenn()** function from package ggvenn

Actually the solution is quite simple:

```
> ggvenn(venn_list)      # this simple command already creates your Venn!
```





Venn Diagrams

Exercise!

Now it's time to customize your graph:

- Remove the circle “**stroke**”
- remove **percentage** of intersections shown

Tips

- 1) Now you have to come back to the RStudio help tab and have a look to the function arguments



Venn Diagrams

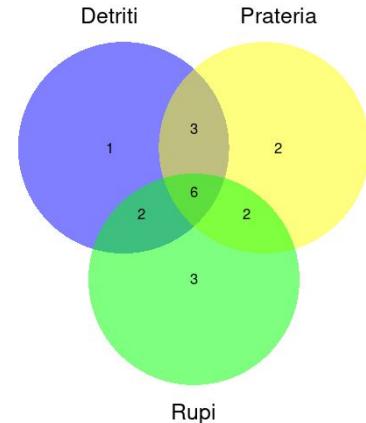
Exercise!

Now it's time to customize your graph:

- Remove the circle “**stroke**”
- remove **percentage** of intersections shown

This will make the job!

```
> ggvenn(venn_list, show_percentage = F, stroke_size = NA)
```





Venn Diagrams

Exercise!

- Add a centered **blue title** with a text **size** bigger than the other text lables in the venn
- Change the color of circles (Rupi must be blue, Prateria must be red and Detriti orange)

The mission looks impossible since there are no **title** arguments in the **ggvenn()**

BUT...

- 1) **ggvenn** is based on **ggplot2** you can further customize graphical stuff using ggplot2 grammar (ggplot2 function can be add to the existing function using “+” operator

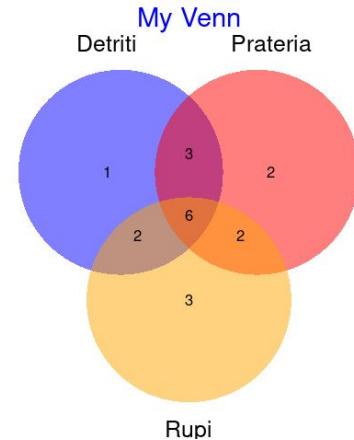


Venn Diagrams

Solution!

```
> ggvenn(venn_list, show_percentage = F, stroke_size = 0, , fill_color = c("blue", "red", "orange")) +  
  ggtitle("My Venn") +  
  theme(plot.title = element_text(hjust=0.5, color="blue", size=20))
```

YES! You have to follow this workflow each time you need to use an unknown function!





Export Figures

Your figures are now ready to be published!

Different approach to export them into a file

- 1) By the RStudio plot interface

PRO: it is quick. It can be useful for share intermediate figures that you want to share with someother or that you do not need to save again. The figures of this presentation has been exported in this way!

CONS: Your figures are not reproducible! If a reviewer ask you to modify a figure you already have submitted to a journal, you cannot obtain the same **figure size**, unless you know how big was your plot interface.



Export Figures

Your figures are now ready to be published!

Different approach to export them into a file

2) Use and ad-hoc function to export figure

PRO: Your figure will look also the same style and the plot the same size each time you re-run your code!

CONS: It requires a little bit more of time to set-up and debug the code.

Some of the most useful functions

`tiff()` Today we'll see this in great detail. Highest quality for mixed line-art and pictures.

`png()`

`pdf()` A good alternative for vectorial-only graphics. Uncomfortable to be used in presentations

`jpeg()`

`svg()`



Export Figures

The Workflow is simple, at least three line of code required

```
> tiff(filename="", etc....)      # this will create a new tiff file on your folder with features set in the function  
  
> your_plot                      # either a plot object or a plot function  
  
> other plot layers               # if required, let's see later..  
  
> dev.off()                     # this function save and closes the file on your folder. It is mandatory to run it,  
                                otherwise you'll see nothing.
```

If the path is not specified your figure
will be saved in your active working
directory.



Export Figures

Let's export the boxplot figure we created before

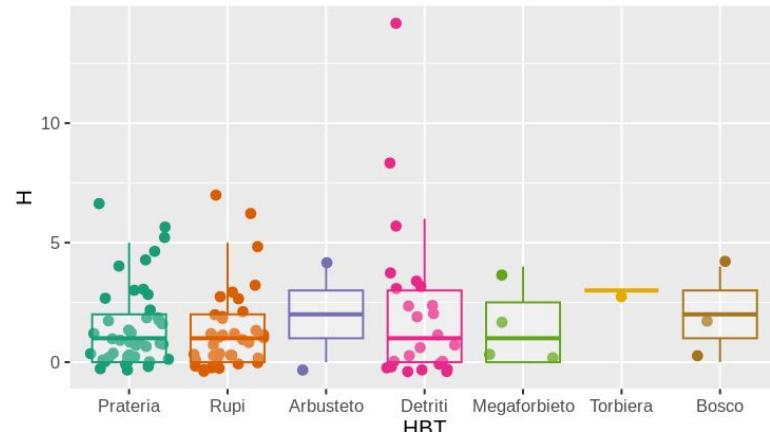
```
> tiff(filename = "Figure 1_v1.tiff", width = 16, height = 10,  
       units="cm", res=600, compression="lzw")
```

```
> boxplot
```

```
> dev.off()
```

png(), **pdf()**, **jpeg()**, **svg()** work in a similar way

Each format has its own specific parameters..



Unit: you can set your preferred one: px, mm, inch, ...

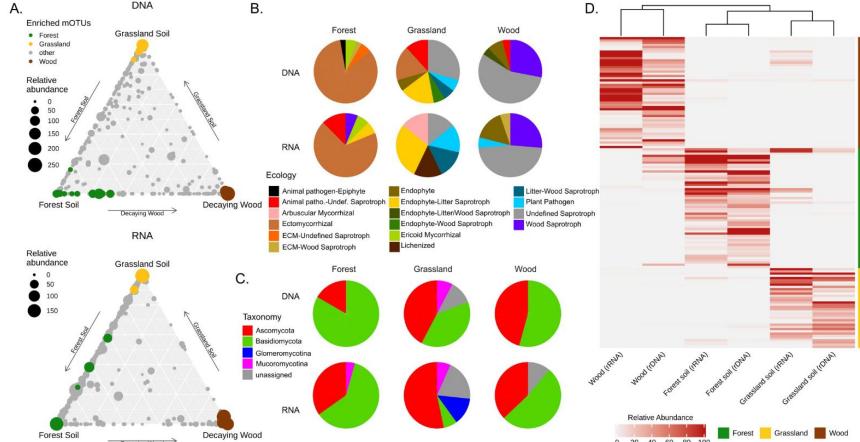
Note: “**lzw**” compress the figures without lowering the quality (lossless compression)

NB. 1) If the path is not specified your figure will be saved in your active working directory.

NB. 2) If you re-run the code without replacing the file name, the existing file will be overwritten

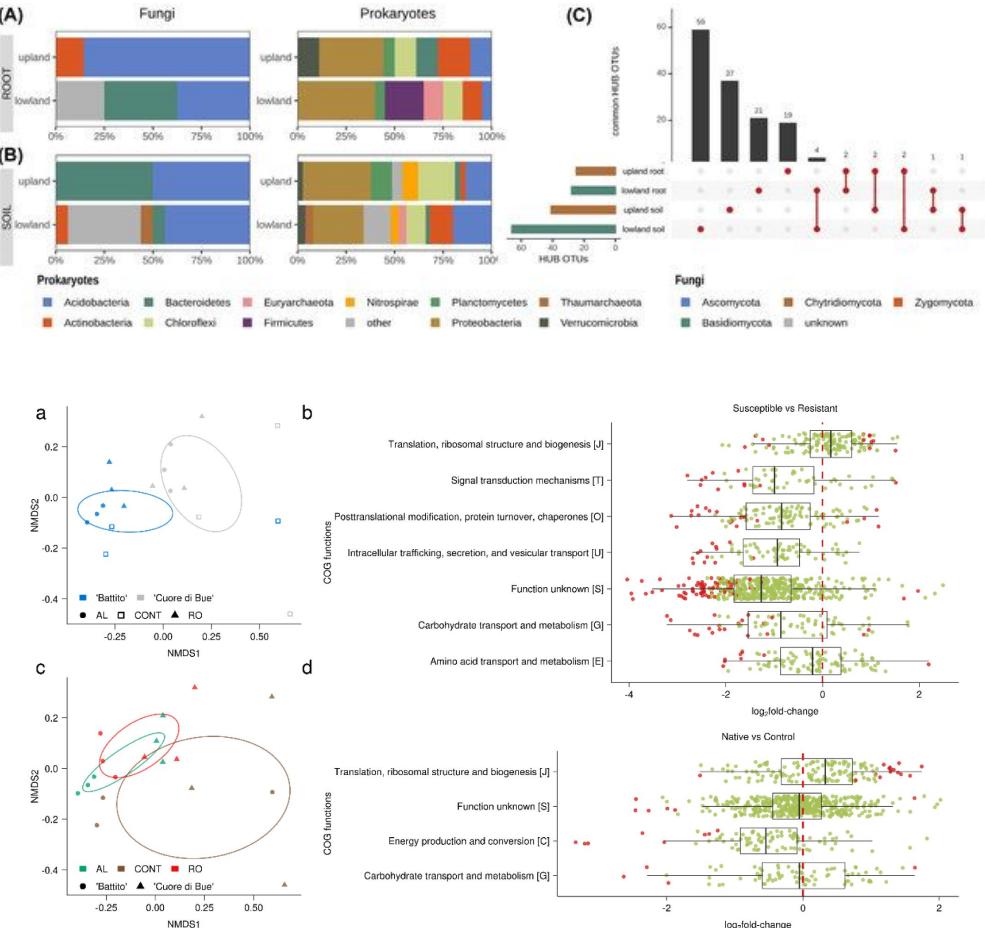
Compose panels

The science ends, the graphics begins.....



We can mainly mix:

- graphs
- legends
- annotation text
- pictures





Compose panels

Some good reasons to NOT use photoshop and assemble composite figure panels in R

- 1) You have to lost LOT of time if you need to redo or update your figure
once your code is ready figure editing is far more quick.
- 2) It is not reproducible
putting together figures in an ad-hoc software can led to different outputs
if the work has to be re-done. Simply by sharing your code anyone can
reproduce your figure.



Compose panels

some libraries we need

```
> library(gridExtra)  
> library(grid)  
> library(lemon)
```

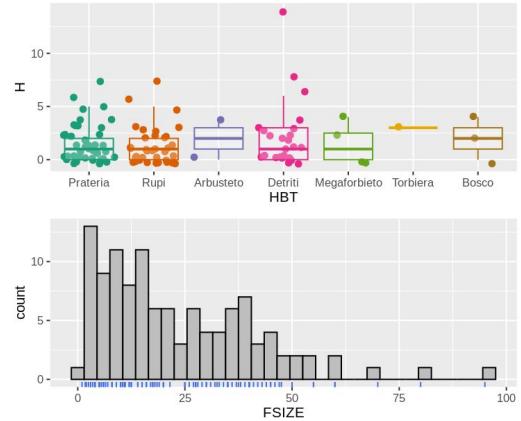
We use the `grid.arrange()` function to merge plots in the same figure



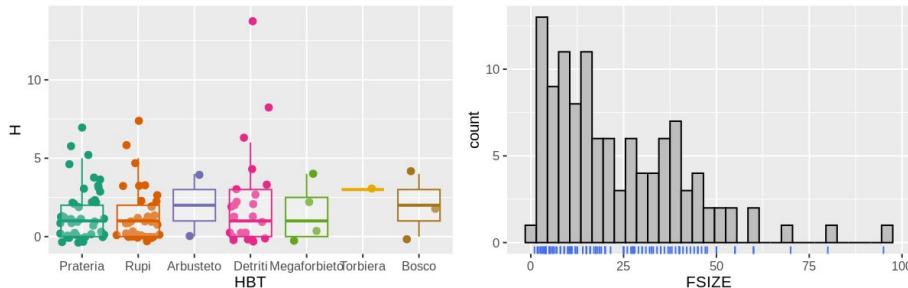
Compose panels

A very simple use case: merge two figures: **boxplot** and **histogram**

```
> grid.arrange(boxplot, histogram) # by default it works on 1 column
```



```
> grid.arrange(boxplot, histogram, ncol=2, nrow=1) # you can specify columns and rows number
```



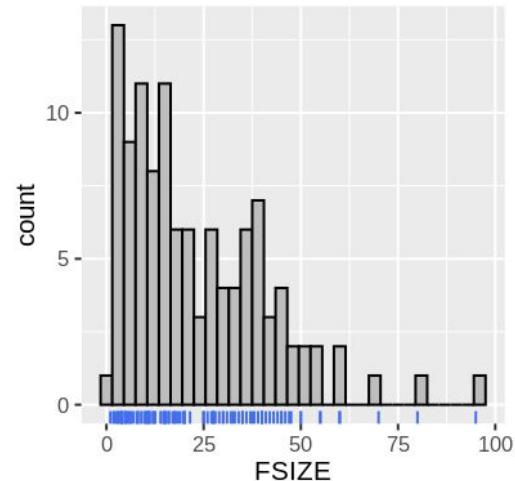
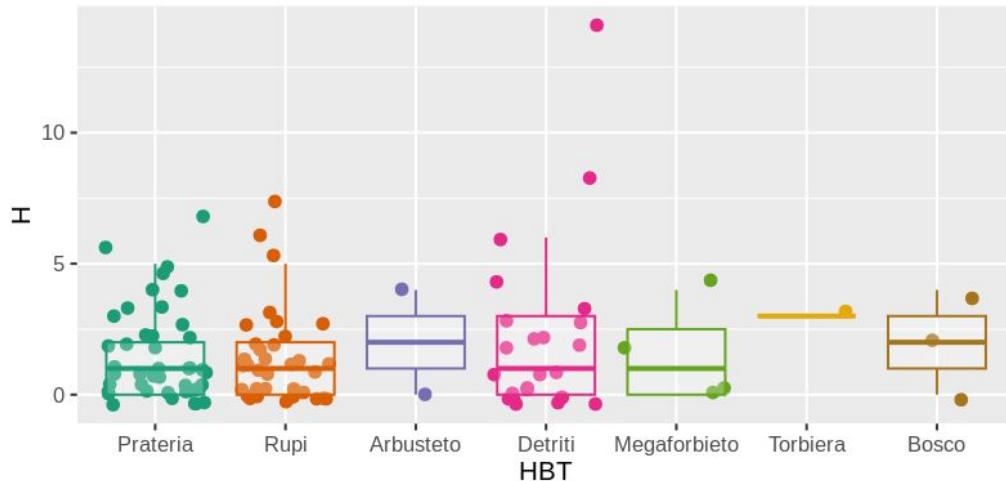


Compose panels

you can change heights or widths of each column/row

```
> grid.arrange(boxplot, histogram, ncol=2, nrow=1, widths=c(1,0.5))
```

Note: here values are relative to their sums: boxplot is double than histogram in width





Compose panels

```
# We need to arrange more plots with complex grid layout
```

```
# We need to use layout_matrix argument!
```

```
> grid.arrange(  boxplot,           # this is number 1  
                 histogram,        # this is number 2  
                 pie,              # this is number 3  
                 barplot,          # this is number 4  
                 layout_matrix=lay )
```

```
# object “lay” should contain your layout: each of the plot will be a number (see above)
```

```
> lay<-rbind(    c(1,1,2),    c(3, 4, 4)  )  
  
> lay<-cbind(    c(1,3), c(1,4), c(2,4)      )      # the same but with cbind()  
  
> lay  
  
 [,1] [,2] [,3]  
[1,]    1    1    2  
[2,]    3    4    4
```



Compose panels

> lay

[,1] [,2] [,3]

[1,] 1 1 2

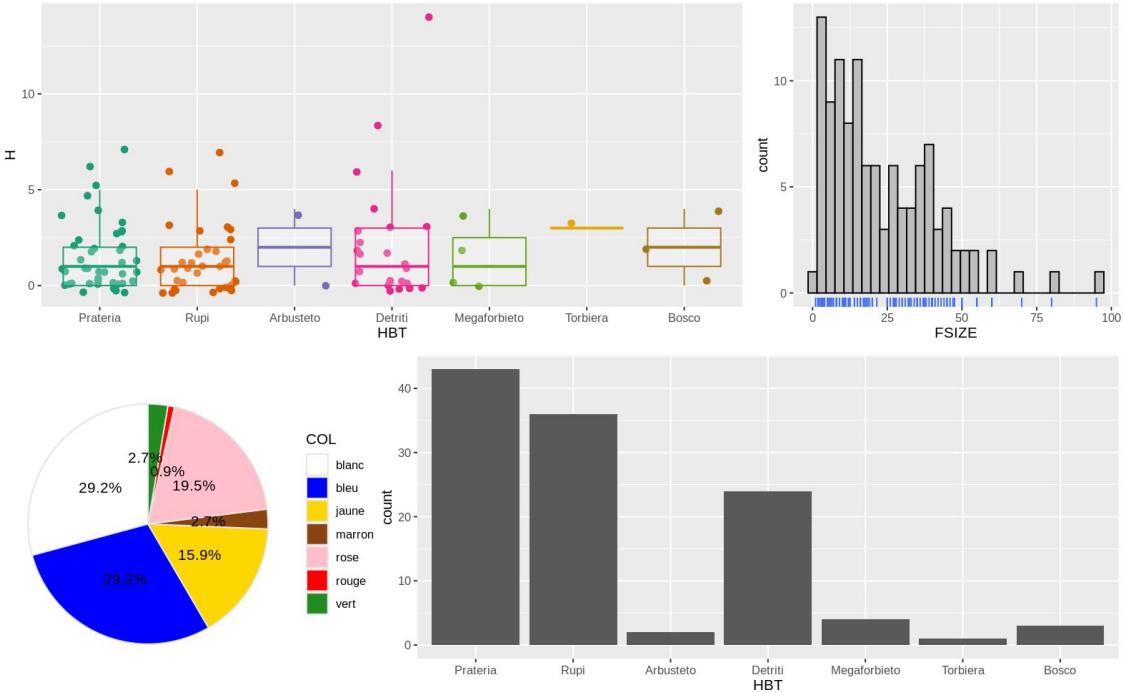
[2,] 3 4 4

you can still apply

heights and **widths** arguments

heights: a vector of 2 elements

widths: a vector of 3 elements





Compose panels

we can add annotation text with **grid.text()** function (from **grid** library)

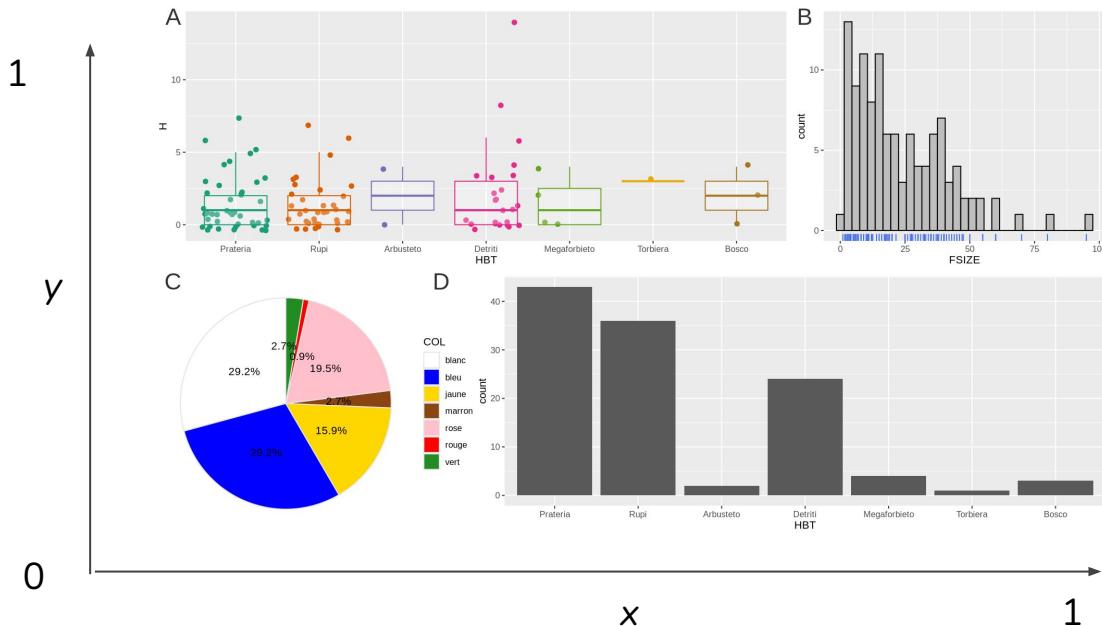
```
> grid.arrange(boxplot, histogram, pie, barplot, layout_matrix=lay)
> grid.text(c("A", "B", "C", "D"), y=c(0.98, 0.98, 0.48, 0.48), x=c(0.02, 0.68, 0.02, 0.3),
  gp=gpar(col="grey20", cex=2))
```

grid.text() needs:

- a vector of characters to plot
- x and y values for each element in the vector. It looks at the plot as a cartesian plane

It always plot over an existing plot!

It can be used to export images using **tiff()** or other image functions





Exercise

Compose the figure panel as in the image below:

