Lecture 2 - Basic plotting and tidy data

Make your paper figures professionally: Scientific data analysis and visualization with R

Julien Gagneur

Lecture overview

12 April	Basic Plots
	Tidy data

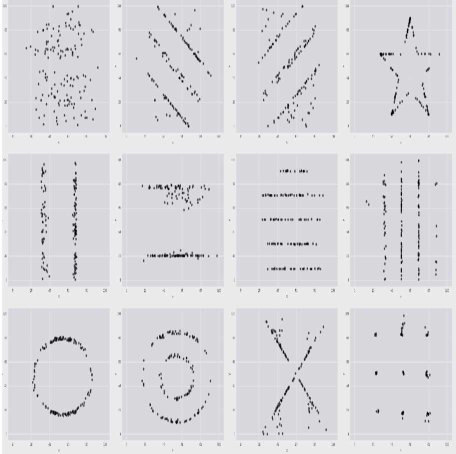
Motivation

Why plotting?

- To observe: Discover associations or patterns in the data (Scientific method)
- To communicate findings effectively

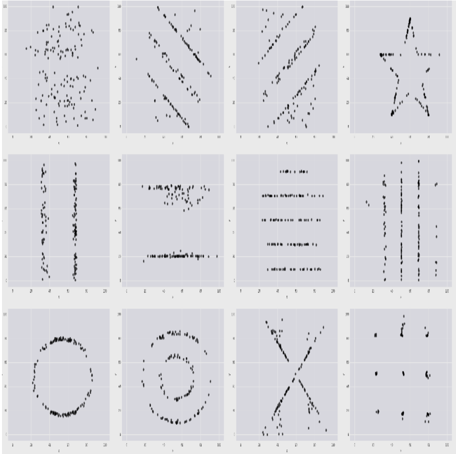
Summary statistics lose information

What have those 12 datasets in common?



Summary statistics lose information

What have those 12 datasets in common?

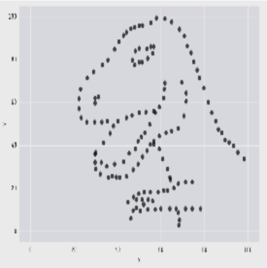


These statistics:

- X mean: 52.26
- Y mean: 47.83
- X standard deviation: 16.76
- Y standard deviation: 29.93
- Pearson correlation: -0.06

Summary statistics lose information

This dataset too..



See https://github.com/lockedata/datasauRus

These statistics:

- X mean: 52.26
- Y mean: 47.83
- X standard deviation: 16.76
- Y standard deviation: 29.93
- Pearson correlation: -0.06

Why plotting?

- To observe: Discover associations or patterns in the data (Scientific method)
- To communicate findings effectively
- Summary statistics lose information

Another motivating example

A vector containing 100 (hypothetical) height measurements for adults in Germany:

```
head(height_dt, n=5)
```

```
##    height
## 1:   1.79
## 2:   1.72
## 3:   1.79
## 4:   1.79
## 5:   1.75
```

We want to know their average height:

```
height_dt[, mean(height)]
```

```
## [1] 3.3635
```

Wait. . . what?

Quiz

Calculating the mean height returns the following output:

```
height_dt[, mean(height)]
```

```
## [1] 3.3635
```

What happened?

1. A. `mean()` is not the right function to assess what we want to know.
2. B. Adults in Germany are exceptionally tall
3. C. A decimal point error in one data point.
4. D. It's a multiple testing problem because we are looking at so many data points (n=100).

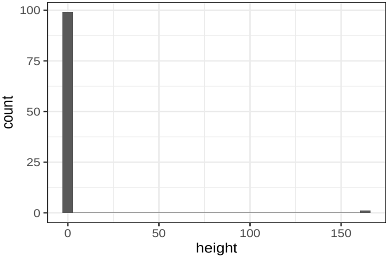
Solution

What happened?

- A. `mean()` is not the right function to assess what we want to know.
  - *No, the mean is exactly what we want.*
- B. Adults in Germany are exceptionally tall.
  - *OK, no. . .*
- **C. A decimal point error in one data point.**
  - *Yes, see next slide.*
- D. It's a multiple testing problem because we are looking at so many data points (n=100).
  - *This question was intentionally misleading, this does not have anything to do with multiple testing.*

Plotting helps to find outliers in the data

```
ggplot(height_dt , aes(height)) + geom_histogram() + mytheme
```



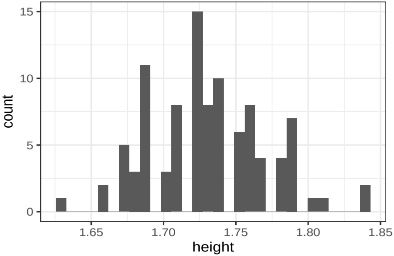
Removing outliers in the data

A quick way to fix our dataset is to remove our outlier.

```
height_dt <- height_dt[height < 3]
```

Removing outliers in the data

```
ggplot(height_dt , aes(height)) + geom_histogram() + mytheme
```




```
height_dt[, mean(height)]
```

```
## [1] 1.730808
```

Why plotting?

- To observe: Discover associations or patterns in the data (Scientific method)
- To communicate findings effectively
- Summary statistics lose information
- To find "bugs in the data" or bugs in your code

Our 3 visualization lectures

- Low dimensional visualizations
  - Grammar of graphics
  - 1-d and 2-d plots (e.g. boxplots, histograms, scatterplots)
- High-dimensional visualizations
  - Heatmaps
  - Clustering
  - Dimensionality reduction
- Graphically supported hypotheses
  - Descriptive vs. demonstrative plots
  - Confounding
  - Guidelines for data visualization and presentation

Grammar of graphics and ggplot2

Grammar of Graphics

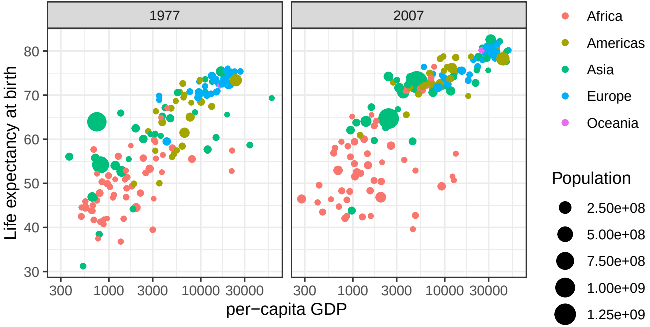
The Grammar of Graphics is a visualization theory developed by Leland Wilkinson in 1999.

- influenced the development of graphics and visualization libraries alike
- 3 key principles
    - Separation of data from aesthetics (e.g. x and y axis, color-coding)
    - Definition of common plot/chart elements (e.g. dot plots, boxplots, etc.)
    - Composition of these common elements (one can combine elements as layers)

Plotting with ggplot2

The library `ggplot2` is a powerful implementation of the grammar of graphics and has become widely used by R programmers.

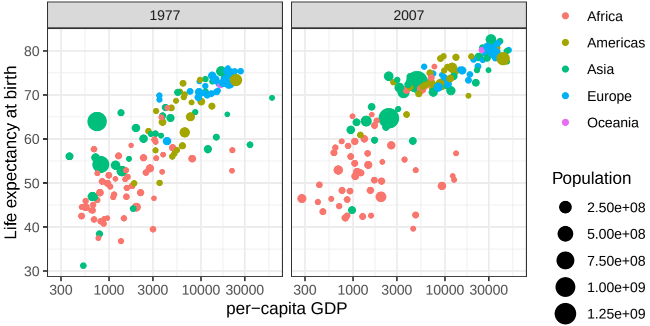
Here is a sophisticated example to compare the relationship between per-capita gross domestic product (GDP) and life expectancy at birth for the years 1997 and 2007:



Example of a first plot with ggplot2

The code for generating the plot. . .

```
library(gapminder)
gm_dt <- as.data.table(gapminder)[year %in% c(1977, 2007)]
ggplot(data = gm_dt, aes(x = gdpPercap, y = lifeExp)) + mytheme +
  geom_point(aes(color=continent, size=pop)) + facet_grid(~year) + scale_x_log10() +
  labs(y="Life expectancy at birth", x="per-capita GDP", size="Population")
```



But how do we create such a plot step by step?

We will have a look at its components and recreate it step by step. . .

Components of the layered grammar

- **Always**:

Data: `data.table` (or `data.frame`) object where columns correspond to variables

Aesthetics: mapping of data to visual characteristics - what we will see on the plot (`aes`) - position (x,y), color, size, shape, transparency

Geometric objects: geometric representation defining the type of the plot data (`geom_`) - points, lines, boxplots, ...

- **Often**:

Scales: for each aesthetic, describes how visual characteristic is converted to display values (`scale_`) - log scales, color scales, size scales, shape scales, ...

Facets: describes how data is split into subsets and displayed as multiple sub graphs (`facet_`)

- **Useful, but with care**:

Stats: statistical transformations that typically summarize data (`stat`) - counts, means, medians, regression lines, ...

- **Domain-specific**:

Coordinate system: describes 2D space that data is projected onto (`coord_`) - Cartesian coordinates, polar coordinates, map projections, ...

ggplot Cheatsheet

https://raw.githubusercontent.com/rstudio/cheatsheets/master/datatable.pdf

Data

Let's have a quick look at our data

```
head(gm_dt[, .(country, continent, gdpPercap, lifeExp, year)])
```

```
##        country continent gdpPercap lifeExp year
## 1: Afghanistan      Asia  786.1134  38.438 1977
## 2: Afghanistan      Asia  974.5803  43.828 2007
## 3:     Albania    Europe 3533.0039  68.930 1977
## 4:     Albania    Europe 5937.0295  76.423 2007
## 5:     Algeria    Africa 4910.4168  58.014 1977
## 6:     Algeria    Africa 6223.3675  72.301 2007
```

Data layer

To start with the visualization we initiate a `ggplot` object loaded with the data. It generates an empty plot:

```
ggplot(gm_dt)
```

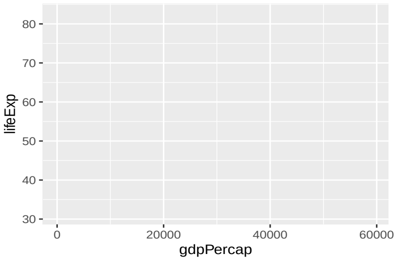
Aesthetics layer

Selecting variables to visualize and their rendering

We next add to the plot an "aesthetic mapping" layer with the function `aes()`, which defines which variables of the table will be visualized and how they map to so-called aesthetics: axis, colors, point shapes, etc.

For a scatter plot, we need at least to define the mapping to `x` and `y`. Now the axes are rendered:

```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp))
```



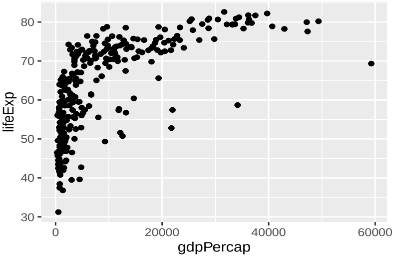
Geometric object layer

Defining the type of the plot

We next add to the plot a "geometric object" layer which defines the type of the plot.

A scatter plot can be plotted using the geometric object function `geom_point()`:

```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp)) + geom_point()
```



Advantage of a layered grammar

Storing `ggplot2` objects

One of the advantages of plotting with `ggplot2` is that it returns an object which can be stored (e.g. in a variable called `p`). The stored object can be further edited and inspected using the `names()` function:

```
p <- ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp)) + geom_point()
p <- p + labs(x='per-capita GDP')
names(p)
```

```
## [1] "data"        "layers"      "scales"      "mapping"     "theme"
## [6] "coordinates" "facet"       "plot_env"    "labels"
```

Advantage of a layered grammar

Saving `ggplot2` objects

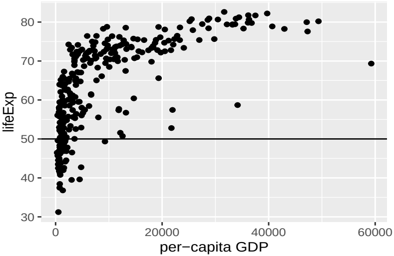
We can also save the `ggplot2` object into a file with the help of the function `saveRDS()`:

```
saveRDS(p, "../../extdata/my_first_plot.rds")
p <- readRDS("../../extdata/my_first_plot.rds")
```

Advantage of a layered grammar

Loading `ggplot2` objects Later, we can read the saved object with the help of the function `readRDS()`. We illustrate this by reading the previously saved plot into a variable `p` and adding a horizontal line at `y=50`:

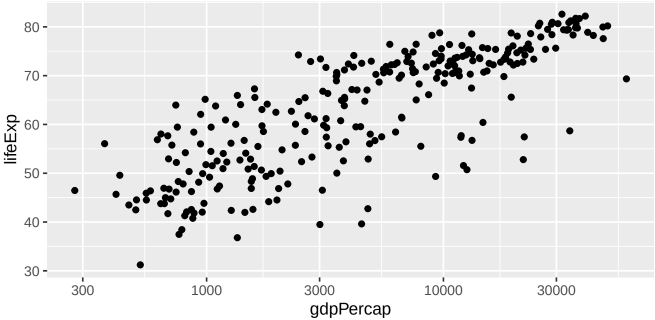
```
p <- readRDS("../../extdata/my_first_plot.rds")
p + geom_hline(yintercept=50)
```



Scale layer

For a better visualization of the data points, we apply log scaling (more details later).

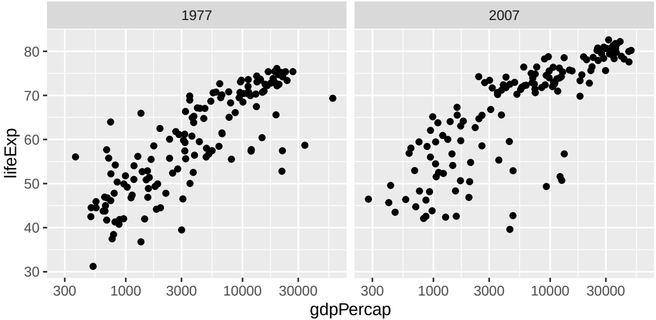
```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp)) +
  geom_point() + scale_x_log10()
```



Facets layer

To compare the data from the year 1977 with the data from 2007, we create a multi-panel plot with `facet_grid()`, which takes a formula as argument specifying by which variable(s) the plot should be split by.

```
ggplot(data = gm_dt, aes(x=gdpPercap, y=lifeExp)) +
  geom_point() + scale_x_log10() + facet_grid(~year) # "~year"  means "by year"
```

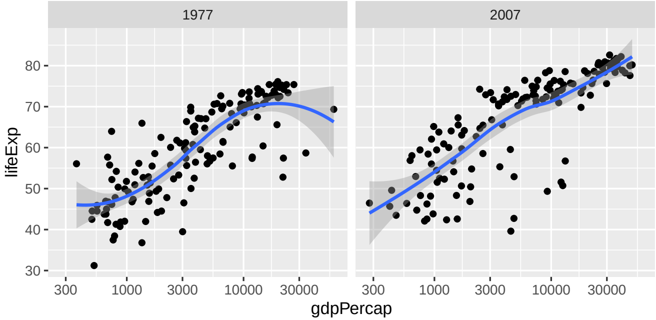


Stats layer

Stats layer are statistical transformations that typically summarize data. Examples are counts, means, medians, regression lines, smooth trends.

Here is an example of smooth trends:

```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp))+
  geom_point() + scale_x_log10() + facet_grid(~year) +
  stat_smooth()
```

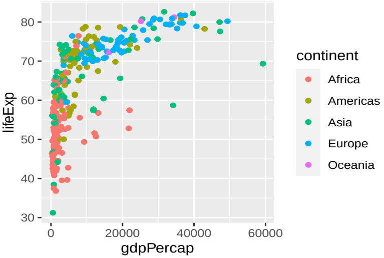


Stats layers can be helpful. However, we recommend to use regression lines an smooth trends **if they are well understood** as they can mislead the interpretation of a plot.

More aesthetics: color

We can easily map variables to different colors, sizes or shapes depending on the value of the specified variable using the aes() function:

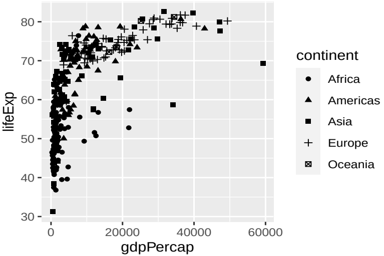
```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp, color=continent)) +
  geom_point()
```



More aesthetics: `shape`

To change the shape of our points we can override the `shape` argument of the `aes()` function:

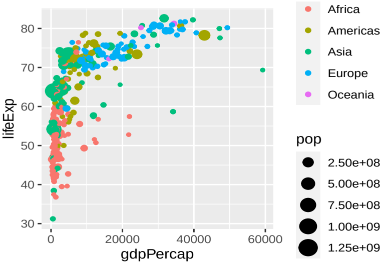
```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp, shape=continent)) +
  geom_point()
```



More aesthetics `size`

Additionally, we distinguish the population of each country by giving a size to the points in the scatter plot:

```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp, color=continent, size=pop)) +
geom_point()
```



Polishing: Axes labels

We can give better labels of the plot with `labs()`:

```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp, color=continent, size=pop))+
  geom_point() + scale_x_log10() + facet_grid(~year) +
  labs(x="per-capita GDP", y="Life expectancy at birth", size = 'Population')
```

Polishing: Themes

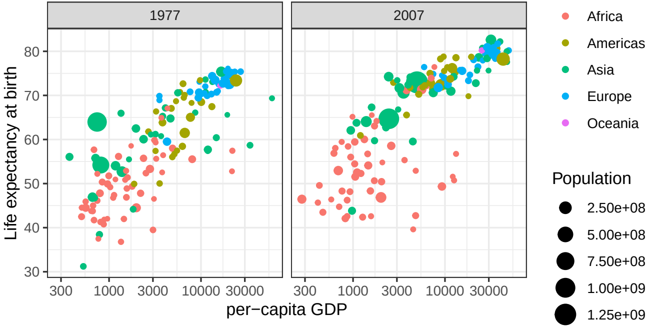
Some default settings can be stored in a so-called `theme`. Themes can be reused, giving your plots a uniform style across a document. Here we create a theme based on the black and white theme of ggplot2. See `?theme()` for details.

```
mysize <- 15 #font size
mytheme <- theme(
    axis.title = element_text(size=mysize),
    axis.text = element_text(size=mysize),
    legend.title = element_text(size=mysize),
    legend.text = element_text(size=mysize)
    ) + theme_bw() # ggplot2's black-and-white theme
```

Adding the theme

Et voilà:

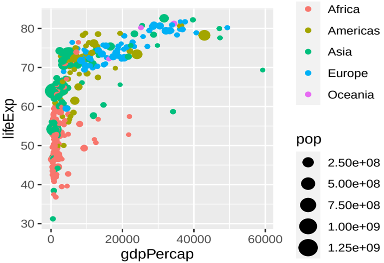
```
ggplot(data=gm_dt, aes(x=gdpPercap, y=lifeExp, color=continent, size=pop)) +
  geom_point(aes(color=continent, size=pop)) + scale_x_log10() + facet_grid(~year) +
  labs(x="per-capita GDP", y="Life expectancy at birth", size = 'Population') +
  mytheme
```



Global vs. individual mapping

Global mapping mapping is inherited by default to all layers, while mapping at individual layers is only recognized at that layer

```
ggplot(data = gm_dt, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(color = continent,size = pop))
```



Global versus individual mapping

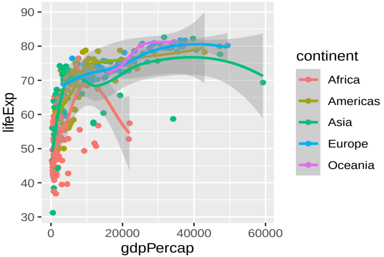
Local mapping cannot be recognized by other layers. For instance, adding another layer for smoothing with `stat_smooth()` like this does not work:

```
# this doesn't work as stat_smooth didn't know aes(x , y)
ggplot(data = gm_dt) + geom_point(aes(x = gdpPercap, y = lifeExp)) +
  stat_smooth()
```

Global versus individual mapping

Local mapping works like this but is too redundant:

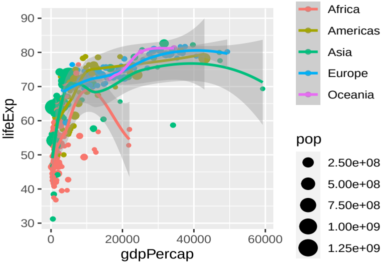
```
# this would work but too redundant
ggplot(data = gm_dt) + geom_point(aes(x = gdpPercap, y = lifeExp, color=continent))
  stat_smooth(aes(x = gdpPercap, y = lifeExp, color=continent))
```



Global versus individual mapping

Local mapping without redundancy:

```
# the common aes(x, y) shared by all the layers can be put in the ggplot()
ggplot(data = gm_dt, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(aes(size = pop)) + stat_smooth()
```



Quiz

What's the result of the following command?

`ggplot(data = mpg)`

1. Nothing happens
2. A blank figure will be produced
3. A blank figure with axes will be produced
4. All data in `mpg` will be visualized

Solution

`ggplot(data = mpg)`: a blank figure will be produced

Quiz

What's the result of the following command?

```
ggplot(data = mpg, aes(x = hwy, y = cty))
```

1. Nothing happens
2. A blank figure will be produced
3. A blank figure with axes will be produced
4. A scatter plot will be produced

Solution

`ggplot(data = mpg, aes(x = hwy, y = cty))`: A blank figure with axes will be produced

Quiz

What's the result of the following command?

```
ggplot(data = mpg, aes(x = hwy, y = cty)) + geom_point()
```

1. Nothing happens
2. A blank figure will be produced
3. A blank figure with axes will be produced
4. A scatter plot will be produced

Solution

`ggplot(data = mpg, aes(x = hwy, y = cty)) + geom_point()`: A scatter plot will be produced

Types of plots for discrete and continuous variables

Types of plots for low dimensional datasets

- In the previous examples, we had a look at scatter plots which are suitable for plotting the relationship between two continuous variables.
- However, there are many more types of plots (e.g. histograms, boxplots) which can be used for plotting in different scenarios.
- Mainly, we distinguish between plotting **one or two variables** and whether the variables are **continuous or discrete**.

Simple plot exercise

Plots for one continuous variable

For plotting **one continuous variable** we mainly use histograms, density plots, boxplots, violinplots and beanplots.

Let us prepare some data for plotting continuous variables using the Human Development Index (HDI) dataset:

```
head(ind, n=3)
```

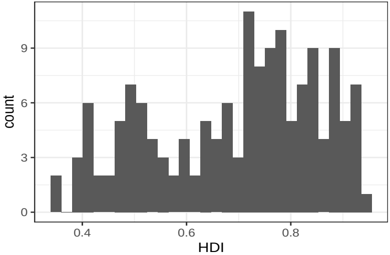
```
##    V1     country wbcode CPI   HDI            region
## 1:  1 Afghanistan    AFG  12 0.465      Asia Pacific
## 2:  2     Albania    ALB  33 0.733 East EU Cemt Asia
## 3:  3     Algeria    DZA  36 0.736              MENA
```

with columns V1: row index, country, wbcode: World Bank Code, CPI: Corruption Perception Index, HDI: Human Development Index, and world region.

Histograms

A histogram represents the frequencies of values of a continuous variable bucketed into ranges. For this, we can use the function `geom_histogram()`:

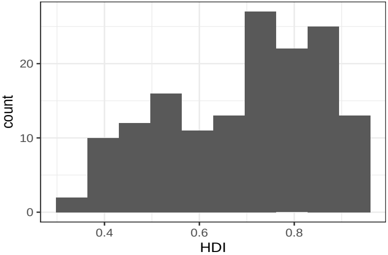
```
ggplot(ind, aes(HDI)) + geom_histogram() + mytheme
```



Histograms: setting the number of bins

By default, the number of bins in ggplot2 is 30. We can change this by defining the number of desired bins in the bins argument of the geom_histogram() function:

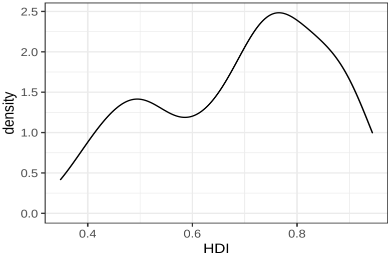
```
ggplot(ind, aes(HDI)) + geom_histogram(bins=10) + mytheme
```



Density plots

Sometimes densities are shown instead of histograms. In `ggplot2` we use the function `geom_density()`:

```
ggplot(ind, aes(HDI)) + geom_density() + mytheme
```



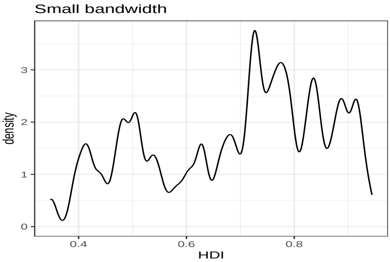
These smoothed distribution plot are typically obtained by kernel density estimation:
https://en.wikipedia.org/wiki/Kernel_density_estimation

Density plots: setting the bandwidth

The bw argument of the geom_density() function allows to tweak the bandwidth of a density plot manually.

Smaller bandwidth:

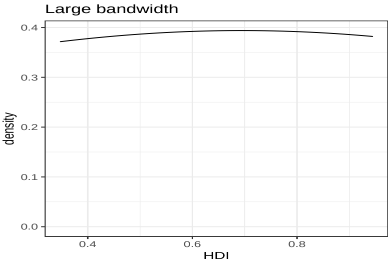
```
ggplot(ind, aes(HDI)) + geom_density(bw=0.01) + ggtitle('Small bandwidth') +
  mytheme
```



Density plots: setting the bandwidth

Larger bandwidth:

```
ggplot(ind, aes(HDI)) + geom_density(bw=1) + ggtitle('Large bandwidth') +
  mytheme
```



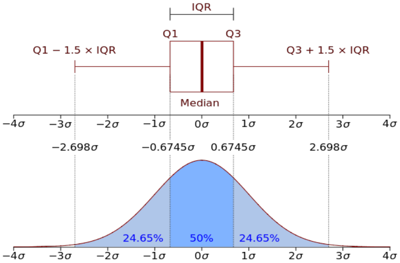
Be careful with density plots as the bandwidth can have strong visual effects. Histograms are not that bad and show the data 'raw'.

General principle: Show the data as raw as possible.

Histogram exercise

Boxplots

Box plots can give a good graphical insight into the distribution of the data.

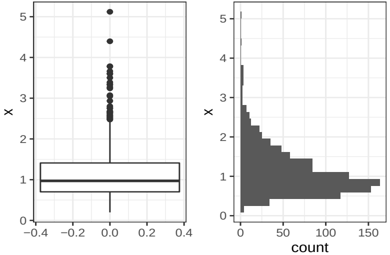


- Median: the center of the data, middle value of a sorted list, 50% quantile of the data
- First quantile (Q1) and the third quantile (Q3): 25% and 75% quantiles of the data
- Interquantile range (IQR): the distance between Q1 and Q3

Boxplots: example

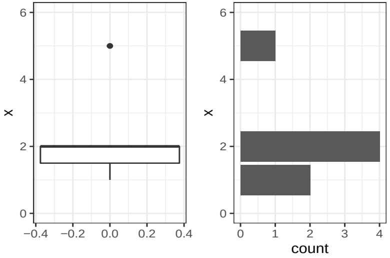
Here is an example of a boxplot with log-normal distribution, which is an asymmetric distribution. The corresponding histogram is shown for comparison.

```
dt <- data.table(x=rlnorm(1000,meanlog=0,sdlog=0.5)) # 1,000 random draws of log-nor
ggplot(dt, aes(x)) + geom_boxplot()
```



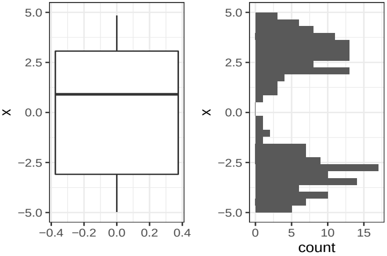
Boxplots are not appropriate for discrete data

For discrete data (e.g. number of gears of a car, rolls of a dice), a boxplot is not appropriate. Choose a barplot or histogram.



Boxplots are not appropriate for multi-modal distribution

Boxplot are misleading for multi-modal distributions, i.e. distribution with multiple peaks of densities, for instance made of two separate clusters of data. Choose a histogram.



Boxplot exercise

Plots for two variables: one continuous and one discrete variable

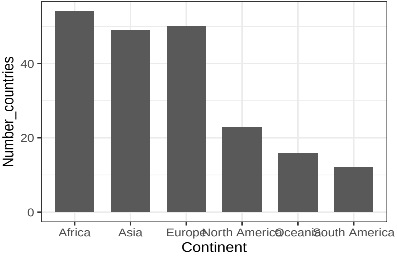
For plotting **one continuous variable** and **one discrete variable** we mainly use

- barplots,
- boxplots by category and
- violinplots by category.

Barplots

Barplots are used to highlight individual quantitative values per category. For creating a barplot with ggplot2 we can use the function geom_bar():

```
ggplot(countries_dt, aes(Continent, Number_countries)) +
  geom_bar(stat = 'identity', width = .7) + mytheme
```



Barplots with errorbars

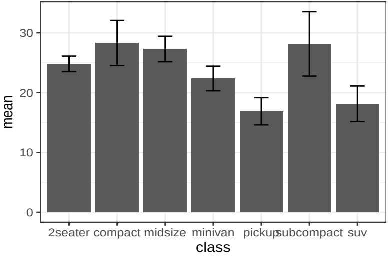
Visualizing uncertainty is important, otherwise, barplots with bars as a result of an aggregation can be misleading. One way to visualize uncertainty is with error bars.

As error bars, we can consider the standard deviation (SD) which can be computed in R with the function `sd()`.

Barplots with errorbars

In the following example, we plot the average highway miles per gallon `hwy` per vehicle class `class` including error bars computed as the average plus/minus standard devation of `hwy`:

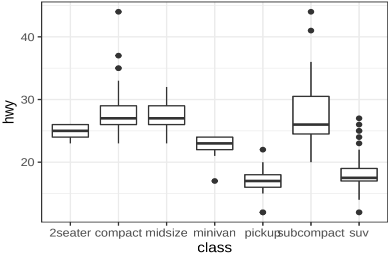
```
as.data.table(mpg) %>%  .[, .(mean = mean(hwy), sd = sd(hwy)), by = class] %>%
  ggplot(aes(class, mean, ymax=mean+sd, ymin=mean-sd)) +
  geom_bar(stat='identity') + geom_errorbar(width = 0.3) + mytheme
```



Boxplots by category

Boxplots are well suited to compare distributions, i.e. plotting distributions of a continuous variable with respect to some categories:

```
ggplot(mpg, aes(class, hwy)) + geom_boxplot() + mytheme
```

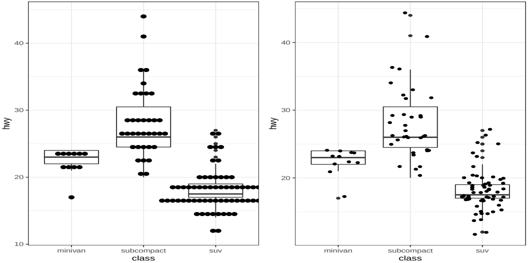


One favor boxplots over barplots for showing median as they show more data (IQR, outliers)
General principle Increase the data/ink ratio

Boxplots with dots

We can add dots (or points) to a box plot using the functions `geom_dotplot()` or `geom_jitter()`:

```
p <- ggplot(mpg_, aes(class, hwy)) + geom_boxplot() + mytheme
p1 <- p + geom_dotplot(binaxis='y', stackdir='center', dotsize=0.5)
p2 <- p + geom_jitter(width=0.3)
cowplot::plot_grid(p1, p2)
```

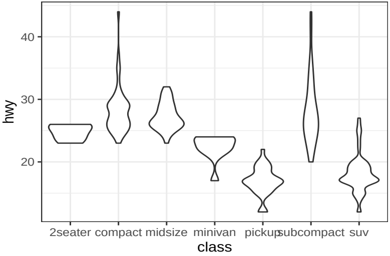


General principle Show the data as raw as possible ***General principle*** Increase the data/ink ratio

Violin plots

A violin plot is an alternative to the boxplot. An advantage of the violin plot over the boxplot is that it also shows the distribution of the data. This can be particularly interesting when dealing with multimodal data. For this, we use the `geom_violin()` function:

```
ggplot(mpg, aes(class, hwy)) + geom_violin() + mytheme
```



Quiz

For which type of data will boxplots produce meaningful visualizations? (2 possible answers)

1. For discrete data.
2. For bi-modal distributions.
3. For non-Gaussian, symmetric data.
4. For exponentially distributed data.

Solution

For which type of data will boxplots produce meaningful visualizations?

- **3. For non-Gaussian, symmetric data.**
- **4. For exponentially distributed data.**

Boxplots are bad for bimodal data since they only show one mode (the median), but are ok for both symmetric and non-symmetric data, since the quartiles are not symmetric.

Plots for two continuous variables

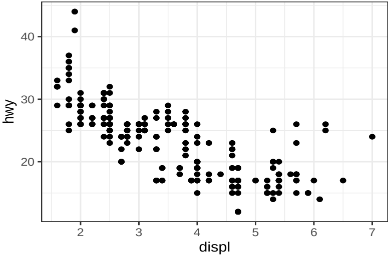
For plotting **two continuous variables** we mainly use

- scatterplots and
- line plots

Scatter plots

Scatter plots are a useful plot type for easily visualizing the relationship between two continuous variables. To make a scatter plot we use the geom_point() function:

```
ggplot(mpg, aes(displ, hwy)) + geom_point() + mytheme
```

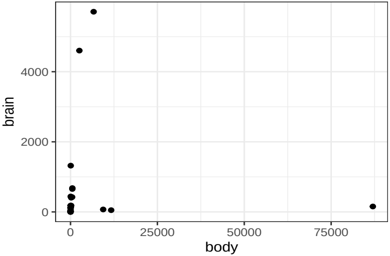


Scatter plots with logarithmic scaling

Consider this scatter plot showing the weights of the brain and body of different animals:

```
library(MASS) # to access Animals data sets
animals_dt <- as.data.table(Animals)
```

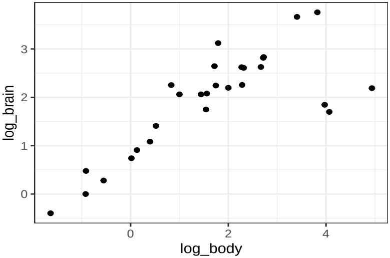
```
ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() + mytheme
```



We can clearly see that there are a few points which are notably larger than most of the points. This makes it harder to interpret the relationships between most of these points. In such cases, we can consider **logarithmic** transformations and/or scaling.

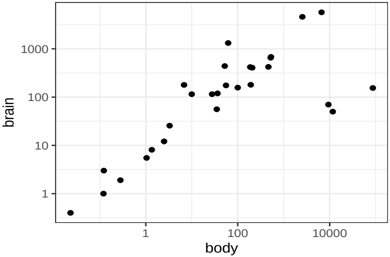
A first idea would be to manually transform the values into a logarithmic space and plot the transformed values instead of the original values:

```
animals_dt[, c('log_body', 'log_brain') := list(log10(body), log10(brain)) ]
ggplot(animals_dt, aes(x = log_body, y = log_brain)) + geom_point() + mytheme
```



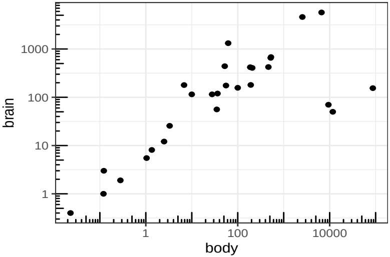
A better option is to logarithmic scales of the plot. The functions `scale_x_log10()` and `scale_y_log10()` allow appropriate scaling and labeling of the axes:

```
ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() +
  scale_x_log10() + scale_y_log10() + mytheme
```



Log-scale ticks on the axis using `annotation_logticks()` make very obvious we look at a logarithmic scale:

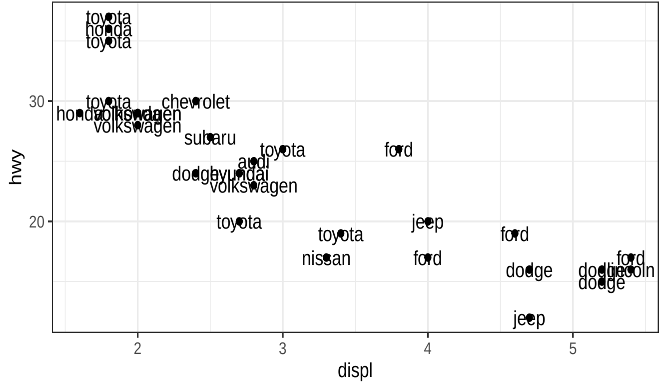
```
ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() +
  scale_x_log10() + scale_y_log10() +
annotation_logticks()  + mytheme
```



Scatter plots with text labeling

Labeling the individual points in a scatter plot may be useful in some applications. For this, ggplot2 offers the function geom_text(). However, these labels tend to overlap:

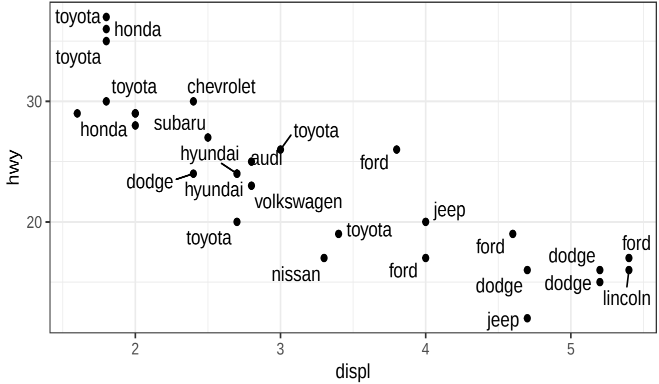
```
ggplot(mpg_subset, aes(displ, hwy, label=manufacturer)) + geom_point() +
  geom_text() + mytheme
```



Scatter plots with text labeling

For a better understanding of the labels we exchange the function `geom_text()` by `geom_text_repel()` from the library `ggrepel`:

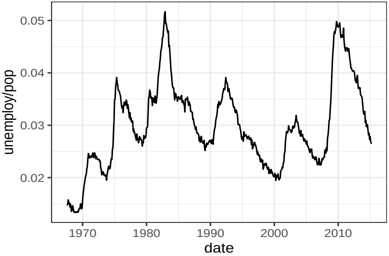
```
library(ggrepel)
ggplot(mpg_subset, aes(displ, hwy, label=manufacturer)) +
  geom_point() + geom_text_repel() + mytheme
```



Line plots

Line plots can be also used to plot two continuous variables. For this we use the function `geom_line()`:

```
ggplot(economics, aes(date, unemploy/pop)) + geom_line() + mytheme
```



Quiz

When to use a line plot?

A. To show a connection between a series of individual data points B. To show a correlation between two quantitative variables C. To highlight individual quantitative values per category D. To compare distributions of quantitative values across categories

Solution

When to use a line plot?

A. **To show a connection between a series of individual data points**

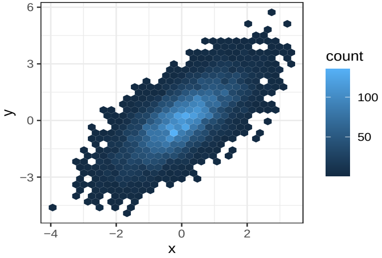
A line plot can be considered for connecting a series of individual data points or to display the trend of a series of data points. This can be particularly useful to show the shape of data as it flows and changes from point to point.

2D-Density plots

In scatter plots, we can not clearly see how many points are at a certain position. This is especially problematic with large datasets. A 2D density plot counts the number of observations within a particular area of the 2D space.

The function `geom_hex()` is particularly useful for creating 2D density plots in R:

```
x <- rnorm(10000); y=x+rnorm(10000)
data.table(x, y) %>% ggplot(aes(x, y)) +
  geom_hex() + mytheme
```

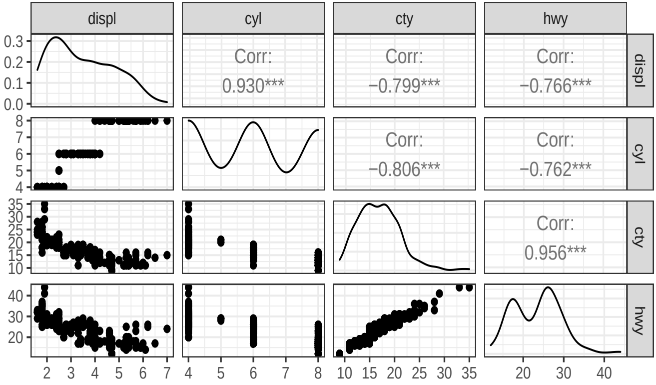


Further plots for low dimensional datasets

Scatterplot matrix

A scatter plot matrix is useful for exploring the distributions and correlations of a few variables in a matrix-like representation. We can use the function `ggpairs()` from the library `GGally` for constructing plot matrices:

```
library(GGally)
ggpairs(mpg, columns = c("displ","cyl","cty","hwy")) + mytheme
```

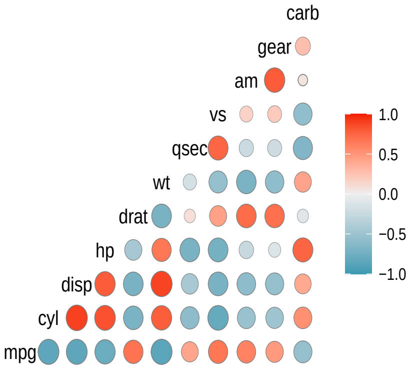


Pearson correlations are shown in the upper triangle and their statistical significance marked with "*" (later lectures).

Correlation plot

A correlation plot is a graphical representation of a correlation matrix. It is useful to highlight the most correlated variables in a dataset. As an example, we visualize the correlation between the variables of the dataset mtcars with the help of the function ggcorr() from the library GGally:

```
ggcorr(mtcars, geom = 'circle')
```



Scatter plot exercise

Summary

Conclusion - You should remember...

- Visualization helps to
  - explore data
  - communicate observations
  - find “bugs” in the data
- The layers of grammar of graphics
- Which plots to use for univariate and bivariate situations
- Interpret elements of a boxplot
- Plotting guiding principles
  - Show the data as raw as possible
  - Increase data/ink ratio

References

- H. Wickham, A Layered Grammar of Graphics, Journal of Computational and Graphical Statistics, 2010 https://vita.had.co.nz/papers/layered-grammar.pdf
- See also Udacity's Data Visualization and D3.js

Tidy data

Now you will learn. . .

- The notion of tidy data
- Operations for tidying up messy data tables
- How to concatenate tables with the same format
- How to merge tables by common variables
- There is no single tidy representation of a dataset

This will set us ready for data visualization and analytics.

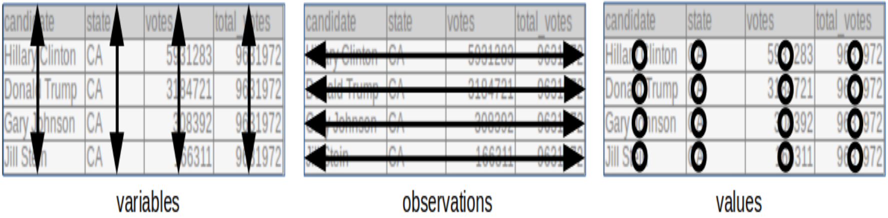
Tidy Data

Motivation

- Without good practices, much of the time of a data analyst can be wasted in data wrangling rather than visualization or analysis.
- The concept of **tidy data** (Wickham, 2014) offers a standard representation of data, that is easy to manipulate, model and visualize.

Advanced data.table cheatsheet
https://raw.githubusercontent.com/rstudio/cheatsheets/master/datatable.pdf

Definition of tidy data



We say that a data table is in *tidy* format if:

1. Each **variable** has its own **column**.
2. Each **observation** has its own **row**.
3. Each **value** has its own **cell**.

Example dataset

Data from 2016 US presidential vote[1] is an example of a tidy dataset:

```
##          candidate state   votes total_votes
## 1: Hillary Clinton    CA 5931283     9631972
## 2:    Donald Trump    CA 3184721     9631972
## 3:    Gary Johnson    CA  308392     9631972
## 4:      Jill Stein    CA  166311     9631972
## 5:  Gloria La Riva    CA   41265     9631972
## 6:    Donald Trump    FL 4605515     9386750
```

Each row represents a state and a candidate with each of the four values related to these states stored in the four variables: candidate, state, votes, and total_votes.

[1] https://www.kaggle.com/stevepalley/2016uspresidentialvotebycounty?select=pres16results.csv

Advantages of tidy data

Organizing data in a tidy fashion reduces the burden to frequently reorganize the data.

In particular, the advantages are:

- Easier manipulation using `data.table` commands such as
  - sub-setting by rows and columns
  - by operations
- Vectorized operations become easier to use
- Many other tools work better with tidy data, including:
  - plotting functions
  - hypothesis testing functions
  - modeling functions such as linear regression

Common signs of untidy datasets

Often, untidy datasets can be identified by one or more of the following issues:

- Column headers are values, not variable names
- Multiple variables are stored in one column
- Variables are stored in both rows and columns
- A single observational unit is stored in multiple tables
- *(Multiple types of observational units are stored in the same table)*

H. Wickham, Journ. Stat. Software, 2014

Column headers are values, not variable names

A common sign of untidy data is that columns are not descriptive enough to tell what they contain.

In this example (Religion dataset, Wickham 2014), we see that the columns refer to some amount of money, but what exactly: income, donations, etc.? The column headers are themselves values of a variable not explicitly mentioned.

Untidy:

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Also, avoid when possible special characters in column names!

Column headers are values, not variable names

In this tidy version it is easy to understand what each column and value contain.

Tidy:

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Multiple variables are stored in one column

This table is an example of multiple variables are stored in one column. Untidy:

```
##          candidate state      proportion
## 1: Hillary Clinton    CA 5931283/9631972
## 2:    Donald Trump    CA 3184721/9631972
## 3:    Gary Johnson    CA  308392/9631972
```

The column 'proportion' is stored as a `character` containing two values. Further data manipulation (splitting the column, converting to `numeric`) is needed to compute or do some plotting with it.

Variables are stored in both rows and columns

In the following table (Weather dataset, Wickham, 2014):

- the variable *date* is stored across multiple columns (year, month, d1-d31)
- the `element` column is not a variable; it stores the names of variables

Untidy:

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

A single observational unit is stored in multiple tables

A subset of the World Health Organization Global Tuberculosis Report (`tidyr` package):

Untidy

```
## $Afghanistan
##    year cases population
## 1: 1999   745   19987071
## 2: 2000  2666   20595360
##
## $Brazil
##    year cases population
## 1: 1999 37737  172006362
## 2: 2000 80488  174504898
##
## $China
##    year  cases population
## 1: 1999 212258 1272915272
## 2: 2000 213766 1280428583
```

It is hard to work across multiple tables. One would rather work with a single concatenated table with an extra column "country".

Quiz: Is the following dataset tidy?

Dataset: `mtcars` (Motor Trend Car Road Tests)

- mpg: Miles/(US) gallon
- cyl: Number of cylinders
- hp: Gross horsepower
- wt: Weight (1000 lbs)

```
##                 model  mpg cyl  hp    wt
## 1:          Mazda RX4 21.0   6 110 2.620
## 2:      Mazda RX4 Wag 21.0   6 110 2.875
## 3:         Datsun 710 22.8   4  93 2.320
## 4:     Hornet 4 Drive 21.4   6 110 3.215
## 5: Hornet Sportabout 18.7   8 175 3.440
## 6:            Valiant 18.1   6 105 3.460
```

1. Yes
2. No

Quiz: Is the following dataset tidy?

Dataset: `mtcars` (Motor Trend Car Road Tests)

- mpg: Miles/(US) gallon
- cyl: Number of cylinders
- hp: Gross horsepower
- wt: Weight (1000 lbs)

```
##                 model  mpg cyl  hp    wt
## 1:          Mazda RX4 21.0   6 110 2.620
## 2:      Mazda RX4 Wag 21.0   6 110 2.875
## 3:         Datsun 710 22.8   4  93 2.320
## 4:     Hornet 4 Drive 21.4   6 110 3.215
## 5: Hornet Sportabout 18.7   8 175 3.440
## 6:            Valiant 18.1   6 105 3.460
```

Answer: Yes, each row (observation) is a car model and all columns contain different variables, therefore, it is tidy.

Quiz: Is the following dataset tidy?

```
##         country   1999   2000
## 1: Afghanistan    745   2666
## 2:      Brazil  37737  80488
## 3:       China 212258 213766
```

1. Yes
2. No

Quiz: Is the following dataset tidy?

```
##        country   1999   2000
## 1: Afghanistan    745   2666
## 2:      Brazil  37737  80488
## 3:       China 212258 213766
```

Answer: No, the column names are values not variable names.

Tidy data exercise

Tidying up single data tables

Tidying up single data tables

- melt tables (wide to long)
- cast tables (long to wide)
- separate columns
- unite columns

From wide to long: column headers are values, not variable names

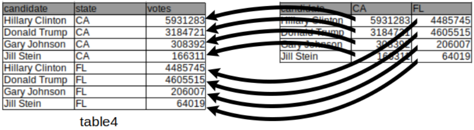
One of the most used operations to obtain tidy data is to transform a **wide** table into a **long** table. This operation is called **melting**, by analogy with melting a piece of metal.

As an example, consider the table below which reports vote counts for two US states, California and Florida. In this untidy table, the column names CA and FA are values of a variable *state*.

```
table4
```

```
##          candidate      CA      FL
## 1: Hillary Clinton 5931283 4485745
## 2:    Donald Trump 3184721 4605515
## 3:    Gary Johnson  308392  206007
## 4:      Jill Stein  166311   64019
```

From wide to long: column headers are values, not variable names



Melting the previous table can be achieved by using the **data.table** function `melt()`:

```r
melt(table4, id.vars = "candidate", measure.vars = c("CA", "FL"),
     variable.name = "state", value.name = "votes") %>% head(n=6)
```

```
##          candidate state   votes
## 1: Hillary Clinton    CA 5931283
## 2:    Donald Trump    CA 3184721
## 3:    Gary Johnson    CA  308392
## 4:      Jill Stein    CA  166311
## 5: Hillary Clinton    FL 4485745
## 6:    Donald Trump    FL 4605515
```

Understanding the melting function

```r
melt(table4, id.vars = "candidate", measure.vars = c("CA", "FL"),
     variable.name = "state", value.name = "votes") %>% head(n=6)
```

```
##          candidate state   votes
## 1: Hillary Clinton    CA 5931283
## 2:    Donald Trump    CA 3184721
## 3:    Gary Johnson    CA  308392
## 4:      Jill Stein    CA  166311
## 5: Hillary Clinton    FL 4485745
## 6:    Donald Trump    FL 4605515
```

When melting, all values in the columns specified by the `measure.vars` argument are **gathered into one column** with name specified in the `variable.name` argument.

Additionally, a new column with name in the `value.name` argument is created **containing all values** which were previously stored in the `measure.vars` **column names**.

After melting, we have a table in a tidy format where a row represents the number of votes for a candidate in a state. The new table also makes clear that the quantities are numbers of votes thanks to the column name.

From long to wide: multiple variables are stored in one column

The other way around also happens frequently.

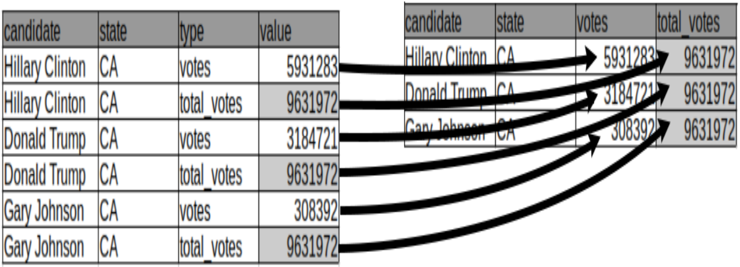
In the table below, multiple values, namely the number of votes for a candidate and the total number of votes, are reported in one column:

```
table2
```

```
##          candidate state        type   value
## 1: Hillary Clinton    CA       votes 5931283
## 2: Hillary Clinton    CA total_votes 9631972
## 3:    Donald Trump    CA       votes 3184721
## 4:    Donald Trump    CA total_votes 9631972
## 5:    Gary Johnson    CA       votes  308392
## 6:    Gary Johnson    CA total_votes 9631972
```

It is not easy to compute the percentage of votes given to a candidate in this format. To tidy up this table we have to **separate** those values **into two columns**.

From long to wide: multiple variables are stored in one column



Casting transforms a long table into a wide table and is achieved with the `dcast()` function whose most frequent usage is:

```
dcast(data, formula, value.var = guess(data))
```

- `formula` specifies which column contains the categories by which the new columns should be created
- `value.var` refers to which column the values have to be extracted from

```
dcast(table2, ... ~ type,  value.var = "value")
```

```
##          candidate state total_votes   votes
## 1:    Donald Trump    CA     9631972 3184721
## 2:    Gary Johnson    CA     9631972  308392
```

Melt/cast exercise

Separating columns

This untidy data stores multiple variables into one column:

```
table3
```

```
##           candidate state      proportion
## 1: Hillary Clinton    CA 5931283/9631972
## 2:    Donald Trump    CA 3184721/9631972
## 3:    Gary Johnson    CA  308392/9631972
```

We split up the `proportion` column into two columns, one containing the votes and the other one containing the total votes with the function `separate()` from the `tidyr` package. By default, columns are separated by any non-alphanumerical character (such as ",", ";", "/",...):

```
separate(table3, col = proportion,
         into = c("votes", "total_votes"))
```

```
##           candidate state   votes total_votes
## 1: Hillary Clinton    CA 5931283     9631972
## 2:    Donald Trump    CA 3184721     9631972
## 3:    Gary Johnson    CA  308392     9631972
```

Uniting columns

Here the first and last names are separated columns without a real need for it (we will not compute any stats over all Hillary's):

```
table5
```

```
##       name surname state   votes total_votes
## 1: Hillary Clinton    CA 5931283     9631972
## 2:  Donald   Trump    CA 3184721     9631972
## 3:    Gary Johnson    CA  308392     9631972
## 4:    Jill   Stein    CA  166311     9631972
## 5:  Gloria La Riva    CA   41265     9631972
```

We unite multiple variables into a single variable with the function `unite()` from the `tidyr`:

```
unite(table5, col = candidate, name, surname, sep = " ")
```

```
##          candidate state   votes total_votes
## 1: Hillary Clinton    CA 5931283     9631972
## 2:    Donald Trump    CA 3184721     9631972
## 3:    Gary Johnson    CA  308392     9631972
## 4:      Jill Stein    CA  166311     9631972
## 5:  Gloria La Riva    CA   41265     9631972
```

The sep argument defines the separating character(s) used to unite the different column values into one.

Quiz: What transformations are required to tidy the following data?

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35

1. cast
2. melt
3. cast and melt
4. unite

Quiz: What transformations are required to tidy the following data?

Answer: 2. melt only. Tidy form:

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Quiz: What transformations are required to tidy the following data?

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—

1. separate
2. unite
3. melt and cast
4. melt, unite and cast

Quiz: What transformations are required to tidy the following data?

Answer: 4. melt, unite and cast. Tidy form:

id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

Concatenating tables

Concatenating tables

- Goal: Concatenate (i.e. append) tables of same format
- Example: Daily COVID-19 data

1. Get all file names of the directory into a list called `files`:

```
files <- list.files('path_to_your_directory')
```

```
head(files)
```

```
## [1] "../../extdata/cov_concatenate//covid_cases_01_03_2020.csv"
## [2] "../../extdata/cov_concatenate//covid_cases_02_03_2020.csv"
## [3] "../../extdata/cov_concatenate//covid_cases_03_03_2020.csv"
## [4] "../../extdata/cov_concatenate//covid_cases_04_03_2020.csv"
## [5] "../../extdata/cov_concatenate//covid_cases_05_03_2020.csv"
## [6] "../../extdata/cov_concatenate//covid_cases_06_03_2020.csv"
```

Concatenating tables

2. load all file contents with fread using lapply

```
# name the list elements by the filenames
names(files) <- basename(files)

# read all files at once into a list of data.tables
tables <- lapply(files, fread)
```

3. View first table:

```
head(tables[[1]])
```

```
##    cases deaths countriesAndTerritories geoId countryterritoryCode popData2019
## 1:    54      0                 Germany    DE                  DEU    83019213
## 2:   240      8                   Italy    IT                  ITA    60359546
##    continentExp Cumulative_number_for_14_days_of_COVID-19_cases_per_100000
## 1:       Europe                                                  0.1156359
## 2:       Europe                                                  1.8638311
```

Concatenating tables

To combine the different tables into one, we can use the data.table function rbindlist() which gives us the option to introduce a new column idcol containing the list names (filenames):

```
# bind all tables into one using rbindlist,
# keeping the list names (the filenames) as an id column.
dt <- rbindlist(tables, idcol = 'filename')
head(dt, n=3)
```

```
##                     filename cases deaths countriesAndTerritories geoId
## 1: covid_cases_01_03_2020.csv    54      0                 Germany    DE
## 2: covid_cases_01_03_2020.csv   240      8                   Italy    IT
## 3: covid_cases_02_03_2020.csv    18      0                 Germany    DE
##    countryterritoryCode popData2019 continentExp
## 1:                  DEU    83019213       Europe
## 2:                  ITA    60359546       Europe
## 3:                  DEU    83019213       Europe
##    Cumulative_number_for_14_days_of_COVID-19_cases_per_100000
## 1:                                                  0.1156359
## 2:                                                  1.8638311
## 3:                                                  0.1373176
```

Merging tables

Merging tables

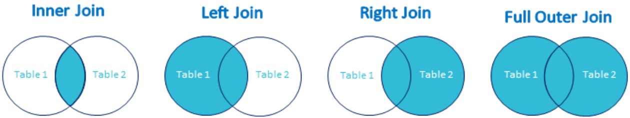
Merging two data tables into one by common column(s) is frequently needed. This can be achieved using the `merge` function whose core signature is:

```
merge(
  x, y,                                    # tables to merge
  by = NULL, by.x = NULL, by.y = NULL,     # by which columns
  all = FALSE, all.x = all, all.y = all    # types of merge
)
```

Types of merges

The four types of merges (also commonly called joins) are:

- **Inner (default)**: consider only rows with matching values in the `by` columns.
- **Outer or full (all)**: return all rows and columns from `x` and `y`. If there are no matching values, return NAs.
- **Left (all.x)**: consider all rows from `x`, even if they have no matching row in `y`.
- **Right (all.y)**: consider all rows from `y`, even if they have no matching row in `x`.



[2]

[2] https://documentation.mindsphere.io/resources/html/predictive-learning/en-US/Types_of_Joins.htm

Merging examples

We provide examples of each type using the following made up tables:

```
dt1 <- data.table(p_id = c("G008", "F027", "L051"),
                  value = rnorm(3))
dt1
```

```
##    p_id     value
## 1: G008 0.5260630
## 2: F027 0.4841822
## 3: L051 1.6767429
```

```
dt2 <- data.table(p_id = c("G008", "F027", "U093"),
                  country = c("Germany", "France", "USA"))
dt2
```

```
##    p_id country
## 1: G008 Germany
## 2: F027  France
## 3: U093     USA
```

Inner merge

An inner merge returns only rows with matching values in the *by* columns and discards all other rows:

```
# Inner merge, default one, all = FALSE
m <- merge(dt1, dt2, by = "p_id", all = FALSE)
m
```

```
##    p_id     value country
## 1: F027 0.4841822  France
## 2: G008 0.5260630 Germany
```

Inner merge

Note that the column order got changed after the merging in the previous example.

To prevent this and, therefore, to keep the original ordering we can use the argument `sort` and set it to `FALSE`:

```
m <- merge(dt1, dt2, by = "p_id", all = FALSE, sort = FALSE)
m
```

```
##    p_id     value country
## 1: G008 0.5260630 Germany
## 2: F027 0.4841822  France
```

Note that the column order got changed after the merging.

Outer (full) merge

An outer merge returns all rows and columns from `x` and `y`. If there are no matching values, it yields missing values (`NA`):

```
# Outer (full) merge, all = TRUE
merge(dt1, dt2, by = "p_id", all = TRUE)
```

```
##    p_id     value country
## 1: F027 0.4841822  France
## 2: G008 0.5260630 Germany
## 3: L051 1.6767429    <NA>
## 4: U093        NA     USA
```

Left merge

Returns all rows from x, even if they have no matching row in y. Rows from x with no matching in y lead to missing values (NA).

```r
# Left merge, all.x = TRUE
merge(dt1, dt2, by = "p_id", all.x = TRUE)
```

```
##    p_id     value country
## 1: F027 0.4841822  France
## 2: G008 0.5260630 Germany
## 3: L051 1.6767429    <NA>
```

Right merge

Returns all rows from `y`, even if they have no matching row in `x`. Rows from `y` with no matching in `x` lead to missing values (`NA`).

```
# Right, all.y = TRUE
merge(dt1, dt2, by = "p_id", all.y = TRUE)
```

```
##    p_id     value country
## 1: F027 0.4841822  France
## 2: G008 0.5260630 Germany
## 3: U093        NA     USA
```

Merging by more than one column

Merging can also be done using more than two columns. Here are two made up tables to illustrate this use case:

```
dt1 <- data.table(firstname = c("Alice", "Alice", "Bob"),
                  lastname = c("Coop", "Smith", "Smith"), x=1:3)
dt1
```

```
##    firstname lastname x
## 1:     Alice     Coop 1
## 2:     Alice    Smith 2
## 3:       Bob    Smith 3
```

```
dt2 <- data.table(firstname = c("Alice", "Bob", "Bob"),
                  lastname = c("Coop", "Marley", "Smith"),
                  y=LETTERS[1:3])
dt2
```

```
##    firstname lastname y
## 1:     Alice     Coop A
## 2:       Bob   Marley B
## 3:       Bob    Smith C
```

Merging by more than one column

We merge now dt1 and dt2 by first name and last name:

```
merge(dt1, dt2, by=c("firstname", "lastname"))
```

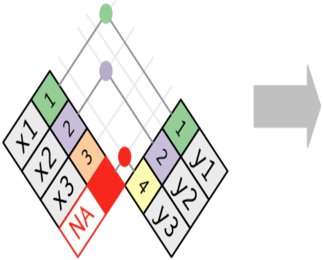
```
##    firstname lastname x y
## 1:     Alice     Coop 1 A
## 2:       Bob    Smith 3 C
```

Note that merging by first name only gives a different result (as expected):

```
merge(dt1, dt2, by="firstname")
```

```
##    firstname lastname.x x lastname.y y
## 1:     Alice       Coop 1       Coop A
## 2:     Alice      Smith 2       Coop A
## 3:       Bob      Smith 3     Marley B
## 4:       Bob      Smith 3      Smith C
```

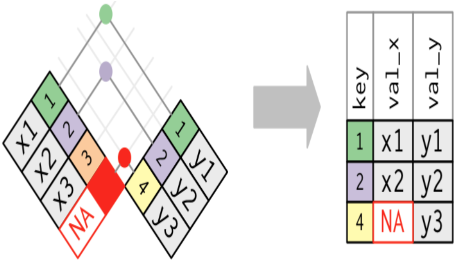
Quiz: How do you perform the data table merge pictured here?



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

1. Inner, all = FALSE
2. Full, all = TRUE
3. Left, all.x = TRUE
4. Right, all.y = TRUE

Quiz: How do you perform the data table merge pictured here?

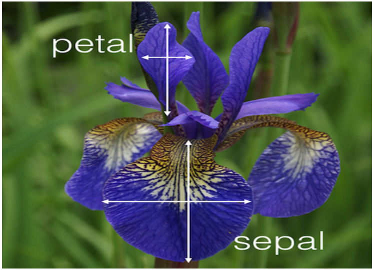


Answer: 4. Right, all.y = TRUE

There is no single tidy representation of a dataset

Fisher's Iris dataset

- 150 iris flowers from 3 different iris species with measurements of 4 different attributes.
- R. Fisher (1936). See https://en.wikipedia.org/wiki/Iris_flower_data_set



The tidy representation depends on what the observation is

We can have different tidy representations depending on what we consider to be an observation, and what the goal of our analysis is.

Here each row represents one flower. Tidy:

```
# Iris dataset, usual representation
iris_dt[1:3,]
```

```
##    Flower Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1:    F_1          5.1         3.5          1.4         0.2  setosa
## 2:    F_2          4.9         3.0          1.4         0.2  setosa
## 3:    F_3          4.7         3.2          1.3         0.2  setosa
```

Here each row represents one measurement. Also tidy:

```
# Another tidy representation
iris_melt[1:3,]
```

```
##    Flower Species    Attribute value
## 1:    F_1  setosa Sepal.Length   5.1
## 2:    F_2  setosa Sepal.Length   4.9
## 3:    F_3  setosa Sepal.Length   4.7
```

On multiple types of observational units in the same table

Remember the typical hallmarks of messy datasets (Wickham, 2014):

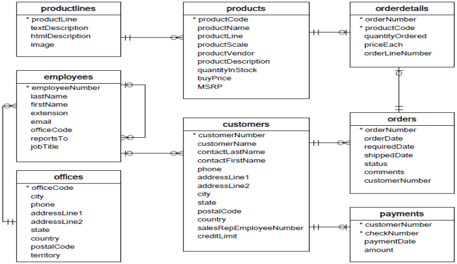
- Column headers are values, not variable names
- Multiple variables are stored in one column
- Variables are stored in both rows and columns
- A single observational unit is stored in multiple tables
- **(Multiple types of observational units are stored in the same table)**

Wickham (2014) mentions the latter may be not so bad. . .

Data tables are typically normalized in databases

Normalized databases ensure that no multiple types of observational units are stored in the same table.

Database example (back-end)



Data analyst tables do not have to be normalized

As data analysts, we are not interested in maintaining a database (back-end), rather in having the desired data in a ready-to-use format which depends on our needs (front-end).

R script example (front-end)

A table combining product and customer data, even if product or customer data are replicated, can be useful:

```
##    productCode quantOrdered priceEach customerNumber customerName state
## 1:        p018            1       450           c001        Smith    CA
## 2:        p030            2       600           c001        Smith    CA
## 3:        p018            1       450           c002        Lewis    AZ
```

Then we can perform easily operations like:

```
prod_dt[, totalPrice := quantOrdered * priceEach]
prod_dt[, N_prod := .N, by = state]
```

Front-end vs. back-end: An analogy

Back-end

Front-end

Summary

Summary

By now, you should be able to:

- define what a tidy dataset is
- recognize untidy data
- perform the operations of melting and casting
- perform the operations of uniting and splitting
- append tables with the same format by rows
- understand and perform the 4 merging operations

Resources

H. Wickham, Journal of Statistical Software, 2014, Volume 59, Issue 10
https://www.jstatsoft.org/v59/i10/paper

https://r4ds.had.co.nz/tidy-data.html

https://cran.r-project.org/web/packages/data.table/vignettes/datatable-reshape.html

Cheatsheets

ggplot cheatsheet https://raw.githubusercontent.com/rstudio/cheatsheets/master/datatable.pdf

Advanced data.table cheatsheet
https://raw.githubusercontent.com/rstudio/cheatsheets/master/datatable.pdf