CeDoSIA SS2020 - Make your paper figures professionally: Scientific data analysis and visualization with R

Vangelis Theodorakis, Xueqi Cao, Daniela Andrade Salazar, Julien Gagneur

23 June, 2020

Contents

Chapter 1

Foreword

The aim of these series of lectures is to provide you with all the knowledge needed in order to make state of the art plots using the R programming language and Grammar of Graphics. The lectures cover the basic concepts of R, how to manipulate data and how to generate clear and meaningful plots for your target audience, using ggplot2 an implementation of Grammar of Graphics for R.

Chapter 2

R basics

In this lecture we will cover the basic concepts of the R programming language: data types, data operations, data structures, control flow and function

2.1 R studio

There are only few things needed to start your data science journey. Those are:

- a computer with an appropriate editor installed
- a programming language of your choice
- and some data (obviously)

You can download R for your particular OS from the r-project site.

And RStudio from the official RStudio site.

The basic editor used by R users is the RStudio. It consists of various sections such as:

- the main script section, for writing scripts [top left section] (Ctrl+1 to focus)
- the console tab, for typing R commands directly [bottom left section as tab] (Ctrl+2 to focus)
- the terminal tab, for direct access to your system shell [bottom left section] (Shift+Alt+T to focus)
- the plot tab, where you see the last plot generated [bottom right section as tab]
- the help tab, with useful documentation of R functions [bottom right section as tab] (F1 on the name of a function or Ctrl+3)
- the history tab, with the list of the R commands used [top right section as tab]
- the environment tab, with the created variables and functions loaded [top right section as tab]
- and the packages tab, with the available/loaded R packages [bottom right section as tab]

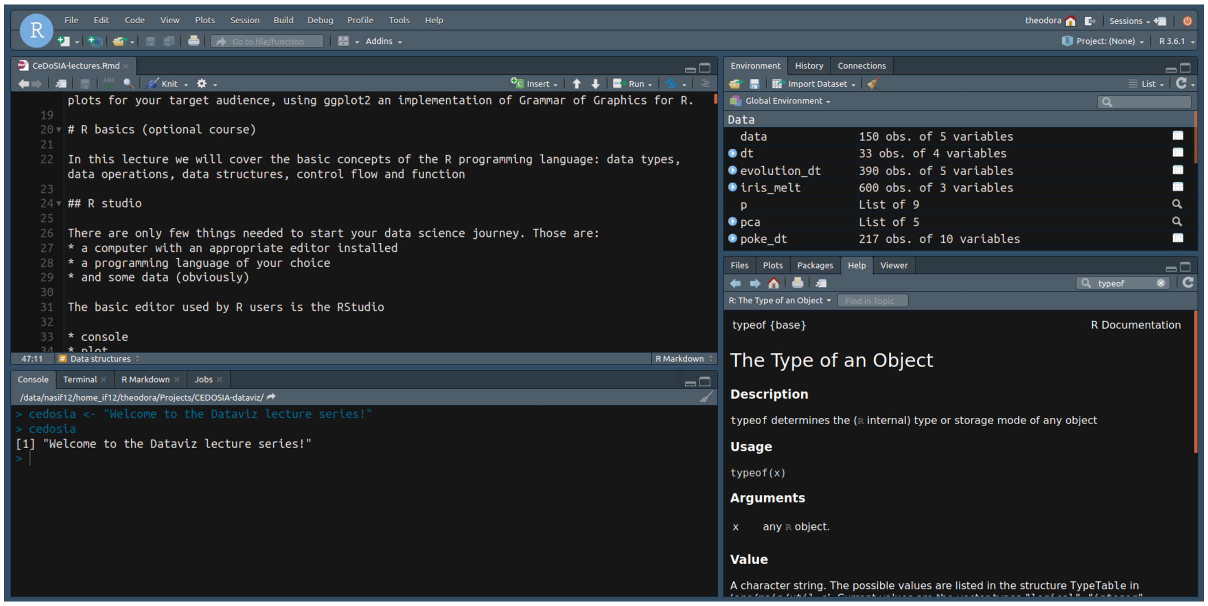
Check the View menu to find out the rest of useful shortcuts!

2.2 Data structures

R has various data structures and types that we will see shortly. From simple scalar variables to tables with multiple features and observation, each layer builds on top of its previous.

Notes:

- Scalars are just vectors of length one
- Use `str()` to see the structure of an object
- All more complex objects, like *S3*, *S4* or *Reference classes* are build from this structures
- Additionally R uses *attributes* to store metadata about an object



```
plots for your target audience, using ggplot2 an implementation of Grammar of Graphics for R.

# R basics (optional course)

In this lecture we will cover the basic concepts of the R programming language: data types,
data operations, data structures, control flow and function

## R studio

There are only few things needed to start your data science journey. Those are:
* a computer with an appropriate editor installed
* a programming language of your choice
* and some data (obviously)

The basic editor used by R users is the RStudio

* console
* plot
```

```
/data/nasif12/home_if12/theodora/Projects/CEDOSIA-dataviz/
> cedosia <- "Welcome to the Dataviz lecture series!"
> cedosia
[1] "Welcome to the Dataviz lecture series!"
>
```

```
Data
  data          150 obs. of 5 variables
  dt            33 obs. of 4 variables
  evolution_dt  390 obs. of 5 variables
  iris_melt     600 obs. of 3 variables
  p             List of 9
  pca           List of 5
  poke_dt       217 obs. of 10 variables
```

typeof {base} R Documentation

The Type of an Object

Description

typeof determines the (R internal) type or storage mode of any object

Usage

typeof(x)

Arguments

x any R object.

Value

A character string. The possible values are listed in the structure TypeTable in

Figure 2.1: RStudio

2.3 Data types

The usual types for an variable can be logical, numeric, double, character or complex (e.g. matrix, data.frame). It is quite handy to know your data types as then you know which operations/functions are available and meaningful to use and it saves you time when debugging

typeof	mode	storage.mode
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character

In practice no distinction is made between doubles and integers. They are just “numerics”.
Numeric are by default doubles:

```
typeof(1)
```

```
## [1] "double"
```

```
typeof(1L)
```

```
## [1] "integer"
```

2.4 Assignments

Assignments in R are preferably of this form:

```
objectName <- value
```

It is also possible to assign with the *equal* sign.

```
objectName = value
```

```
x <- 5  # Both methods have the same outcome
y = 5
```

```
x
```

```
## [1] 5
```

```
y
```

```
## [1] 5
```

However, the “equal” sign is used for argument passing to functions. Thus if nesting, the equal sign will be interpreted as a argument assignment and might throw an error:

```
## We want to measure the running time of the inner product of a large vector and
## assign the outcome of the function to a variable simultaneously
system.time(a <- t(1:1e+06) %*% (1:1e+06))
```

```
##    user  system elapsed
##   0.046   0.003   0.049
```

```
system.time(a = t(1:1e+06) %*% (1:1e+06))  ## This would cause an error
```

```
## Error in system.time(a = t(1:1e+06) %*% (1:1e+06)): unused argument (a =
t(1:1e+06) %*% (1:1e+06))
```

Right Alt + - gives a quick shortcut to add **<-** ;)

2.5 Vectors

Vectors are 1-dimensional data structures which can contain one or more variables, regardless of their type.

Usually created with `c()` (from *concatenation*):

```
c(1, 5, 8, 10)
```

```
## [1]  1  5  8 10
```

```
str(c(1, 5, 8, 10))
```

```
##  num [1:4] 1 5 8 10
```

```
length(c(1, 5, 8, 10))
```

```
## [1] 4
```

```
c("a", "B", "cc")
```

```
## [1] "a"  "B"  "cc"
```

```
c(TRUE, FALSE, c(TRUE, TRUE))
```

```
## [1]  TRUE FALSE  TRUE  TRUE
```

```
c(1, "B", FALSE)
```

```
## [1] "1"     "B"     "FALSE"
```

There are multiple ways to create a numeric sequence depending on the desired result.

Usually for an integer sequence *from:to* is enough. For different results we may need to use `seq()` function utilizing its arguments to increase by a *step* of our choice or to split the range depending on the desired length of the output.

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(from = 1, to = 10, by = 0.3)
```

```
##  [1]  1.0  1.3  1.6  1.9  2.2  2.5  2.8  3.1  3.4  3.7  4.0  4.3  4.6  4.9  5.2
## [16]  5.5  5.8  6.1  6.4  6.7  7.0  7.3  7.6  7.9  8.2  8.5  8.8  9.1  9.4  9.7
## [31] 10.0
```

```r
seq(from = 1, to = 10, length.out = 12)
```

```
##  [1]  1.000000  1.818182  2.636364  3.454545  4.272727  5.090909  5.909091
##  [8]  6.727273  7.545455  8.363636  9.181818 10.000000
```

Keep in mind that sometimes different approaches, give different data types as results so when weird things happen, always check the documentation (devil is in the details)!

For non numeric values e.g. logical, character, `rep()` function comes also in handy.

The *replicate* function `rep()` replicates a vector a certain number of *times* and concatenates them:

```r
rep(c(TRUE, FALSE), times = 5)
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

To replicate *each* entry of the input vector at the time:

```r
rep(c(TRUE, FALSE), each = 5)
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Vector elements are accessed via the `[` operator:

```r
## Create an A,B,C,D,E vector
x <- LETTERS[1:5]
x
```

```
## [1] "A" "B" "C" "D" "E"
```

```r
## access the third entry
x[3]
```

```
## [1] "C"
```

```r
## modify the third entry
x[3] <- "Z"
x
```

```
## [1] "A" "B" "Z" "D" "E"
```

Elements in vectors can have names. Using names instead of index to access entries in a vector make code more robust to re-ordering, sub-setting, and changes in data input.
Names can be created at initialization or set afterwards with `names()`.

```r
x <- c(a = 1, b = 2, c = 3)
x
```

```
## a b c
## 1 2 3
```

```r
names(x) <- c("A", "B", "C")
x
```

```
## A B C
## 1 2 3
```

Names don't have to be unique, but should preferably be, as sub-setting by names will only return the first match

```r
x <- c(a = 1, a = 2, b = 3)
x["a"]
```

```
## a
## 1
```

Not all elements need to have names.

```r
c(a = 1, 2, 3)
```

```
## a
## 1 2 3
```

2.6 Working with missing values

Missing values are specified with `NA` (Not Available). They are placeholders for the specific type and as such are something like an unspecified value. If not taken care of, it can break computation, e.g. What is the sum of [1, 3, 5, x] if you do not know x?

```r
## Define a vector containing a missing value (NA)
v <- c(1, 3, 5, NA)
```

Now we try to compute its mean:

```r
mean(v)
```

```
## [1] NA
```

For many functions including `mean()`, a `na.rm` parameter allows ignoring missing values:

```r
mean(v, na.rm = TRUE)
```

```
## [1] 3
```

2.7 Factors

```r
x <- factor(c("red", "yellow", "red", "green", "green"))
x
```

```
## [1] red    yellow red    green  green
## Levels: green red yellow
```

- A factor is a is used to store categorical data. The distinct categories are called ‘levels’.
- They belong to a special `class()`, *factor*, which makes them behave differently from regular integer vectors.
- Factors cannot be altered in the same way as vectors. Only their levels can be altered.
- Factors are built on top of integer vectors: the values are integer indexes in the dictionary of levels.
- They occupy less space in memory than characters, since they are stored as integers.

Level	Integer
green	1
red	2
yellow	3

- Factors are typically constructed with `factor()`. By default the levels are the unique values, sorted by alphanumerical order.

```
x <- factor(c("red", "yellow", "red", "green", "green"))
x
```

```
## [1] red    yellow red    green  green
## Levels: green red yellow
```

`levels()` gives the levels in ascending order of the underlying encoding integers.

```
levels(x)
```

```
## [1] "green"  "red"    "yellow"
```

The order of the levels can be forced:

```
x <- factor(c("red", "yellow", "red", "green", "green"), levels = c("green", "yellow",
    "red"))
x
```

```
## [1] red    yellow red    green  green
## Levels: green yellow red
```

```
levels(x)
```

```
## [1] "green"  "yellow" "red"
```

The order of the levels is then used for all function requiring comparisons, e.g. sorting

```
sort(x)
```

```
## [1] green  green  yellow red    red
## Levels: green yellow red
```

Only level values can be used:

```
x <- factor(c("red", "yellow", "red", "green", "green"))
x[2] <- "blue"
x
```

```
## [1] red   <NA>  red   green green
## Levels: green red yellow
```

Be aware that R does not prevent you from combining factors:

Do not try to combine factors, especially if levels are not the same!

```
c(x, factor("blue"))
```

```
## [1]  2 NA  2  1  1  1
```

2.8 Math

R supports various mathematical and logical operations between scalars, vectors etc.

- Basic mathematical operations: +, *, -, /
- Additional mathematical operations:
  - %* Matrix multiplication, for vectors: inner product (dot product)
  - **%%** Modulo
  - **%/%** Integer division
  - **%o%** Outer product
  - ˆ Exponentiation
- Boolean operations:
  - **&** element-wise AND
  - **&&** AND left to right until the result is determined

  - | element-wise OR
  - || OR left to right until the result is determined
  - **!** NOT
  - **xor(x,y)** element-wise XOR (exclusive OR)

2.9 Binary comparison

- Element-wise Binary comparison return a vector of same length as the input
  - **==** element-wise equality
  - **!=** element-wise inequality
  - **<**, **<=** element-wise smaller (or equal)
  - **>**, **>=** element-wise greater (or equal)

```
1:5 == 1:5
```

```
## [1] TRUE TRUE TRUE TRUE TRUE
```

- Binary comparison with a single Boolean statement as output
  - `identical(x, y)` exact equality
  - `all(x)` are all TRUE?

```
identical(1:5, 1:5)
```

```
## [1] TRUE
```

2.10 Matrices

2-dimensional structures that can be created by `matrix()`. By default 2D structures are populated column-wise.

```
mat <- matrix(data = 1:12, nrow = 2, ncol = 6, byrow = FALSE)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

```
mat <- matrix(data = 1:12, nrow = 2, ncol = 6, byrow = TRUE)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    7    8    9   10   11   12
```

```
# work the same as matrix(data = 1:12, nrow = 2, ncol = 6, byrow = FALSE)
mat <- matrix(1:12, 2, 6)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

```
dim(mat)
```

```
## [1] 2 6
```

Vector attributes generalizes differently for matrices as dimensions change:

`length()` becomes `nrow()` and `ncol()`

```r
nrow(mat)
```

```
## [1] 2
```

```r
ncol(mat)
```

```
## [1] 6
```

`names()` becomes `rownames()` and `colnames()`

```r
colnames(mat) <- c("A", "B", "C", "D", "E", "F")
rownames(mat) <- c("a", "b")
mat
```

```
##   A B C D  E  F
## a 1 3 5 7  9 11
## b 2 4 6 8 10 12
```

`c()` becomes `cbind()` (column-bind)

```r
mat <- matrix(c(1, 2, 4, 5), 2, 2)
cmat <- matrix(7:8, 2, 1)
mat <- cbind(mat, cmat)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
```

and `rbind()` (row-bind)

```r
rmat <- matrix(c(3, 6, 9), 1, 3)
mat <- rbind(mat, rmat)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Operations between a vector and a matrix is done column-wise:

```r
mat <- matrix(1, 3, 2)
```

```r
vec <- 1:3
vec
```

```
## [1] 1 2 3
```

```r
mat * vec
```

```
Can you guess what will happen?
```

Operations between a vector and a matrix is done column-wise:

```r
mat <- matrix(1:9, 3, 3)
```

```r
vec <- 1:3
vec
```

```
## [1] 1 2 3
```

```r
mat * vec
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    4   10   16
## [3,]    9   18   27
```

- This behavior comes from the fact that all objects in R are basically vectors, which are just “folded” column-wise
- Elements of the shorter vector are constantly reused.

```
as.vector(mat) * vec
```

```
## [1]  1  4  9  4 10 18  7 16 27
```

```
mat * 1:2
```

```
##      [,1] [,2] [,3]
## [1,]    1    8    7
## [2,]    4    5   16
## [3,]    3   12    9
```

- Elements of the shorter vector are constantly reused. If the shorter vector is a multiple of the longer vector, the computation succeeds. Otherwise it will run until the end and throw a warning.

```
mat * 1:4
```

```
##      [,1] [,2] [,3]
## [1,]    1   16   21
## [2,]    4    5   32
## [3,]    9   12    9
```

2.11 Lists

Lists are heterogeneous vectors, that is the elements can be of *any type*, including lists (recursive). Construct lists by using `list()`:

```
x <- list(c(1, 2, 3), "some text", list(c(TRUE, FALSE)))
x
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "some text"
##
## [[3]]
## [[3]][[1]]
## [1]  TRUE FALSE
```

`typeof` list is *list*:

```
typeof(x)
```

```
## [1] "list"
```

List elements are accessed via the `[[` operator:

```
## Create an A,B,C list
x <- as.list(LETTERS[1:3])
x
```

```
## [[1]]
## [1] "A"
##
## [[2]]
## [1] "B"
##
```

```
## [[3]]
## [1] "C"
```

```
## access the third element with `[[`
x[[3]]
```

```
## [1] "C"
```

```
## modify the third element
x[[3]] <- "Z"
x[[3]]
```

```
## [1] "Z"
```

```
## accessed with only a single `[` we get a list of the given index
x[3]
```

```
## [[1]]
## [1] "Z"
```

2.12 Data Frames

- *Data frames* are the most common way to store data
- It's a list of equal-length vectors
- Because of this 2-dimensional structure it shares properties of both a *list* and a *matrix*
- So a data frame has `rownames()` and `colnames()` equivalent to `names()`
- It has also `nrow()` and `ncol()` equivalent to `length()`
- `length()` actually returns the number of *list elements*, which is the column dimension
- Data frames are created with `data.frame()` and named vectors (also lists) or matrices as input

```
df <- data.frame(number = 1:4, letter = letters[1:4])
df
```

```
##   number letter
## 1      1      a
## 2      2      b
## 3      3      c
## 4      4      d
```

Matrices and lists can be without names, but data frames not. If an unnamed vector or matrix is provided, names are set by default. The functions `colnames` and `rownames` can be used to view and set the names.

```
rownames(df)
```

```
## [1] "1" "2" "3" "4"
```

```
rownames(df) <- paste0("row_", nrow(df):1)
rownames(df)
```

```
## [1] "row_4" "row_3" "row_2" "row_1"
```

- Merging works with `cbind()` and `rbind()` like in matrices and with `c()` like in lists.
- If combining column-wise, the number of rows must match

```
df <- data.frame(number = 1:2, letter = letters[1:2])
cbind(df, data.frame(cont = 5:6))
```

```
##   number letter cont
## 1      1      a    5
## 2      2      b    6
```

- If combining row-wise, **the column names have to be identical**.

```
rbind(df, data.frame(n = 3, l = letters[3]))  # This will cause and Error
```

```
## Error in match.names(clabs, names(xi)): names do not match previous names
```

```
rbind(df, data.frame(number = 3, letter = letters[3]))
```

```
##   number letter
## 1      1      a
## 2      2      b
## 3      3      c
```

2.13 Sub-setting

- There are three accessing methods
  - **[** For vectors
  - **[[** For list-based structures
  - **$** Similar to the prior
- Six ways of sub-setting
  - By positive integers
  - By negative integers
  - By logical vectors
  - Empty sub-setting
  - By Zero
  - Character vector

2.13.1 Atomic vectors

```
x <- LETTERS[1:5]
x
```

```
## [1] "A" "B" "C" "D" "E"
```

```
## By positive integers
x[c(1, 3)]
```

```
## [1] "A" "C"
```

```
## By negative integers
x[-c(1, 3)]
```

```
## [1] "B" "D" "E"
```

```
## By logical vectors
x[c(TRUE, TRUE, TRUE, FALSE, FALSE)]
```

```
## [1] "A" "B" "C"
```

```
## By logical vectors
x[x <= "C"]
```

```
## [1] "A" "B" "C"
```

`NA`s in the input vector will always produce an `NA` in the output:

```
x[c(1, 2, NA)]
```

```
## [1] "A" "B" NA
```

We can also sub-setting by character vectors, if names are present:

```
names(x) <- letters[1:5]
x[c("a", "b")]
```

```
##   a   b
## "A" "B"
```

2.13.2 Lists

Lists behave just like atomic vectors. The output is a list

```
myList <- list(char = c("a", "b"), int = 1:5, logic = TRUE)
## By positive integers
myList[[1]]
```

```
## [1] "a" "b"
```

```
myList[1]
```

```
## $char
## [1] "a" "b"
```

```
## Character vector
myList["int"]
```

```
## $int
## [1] 1 2 3 4 5
```

```
## By logical vectors
myList[c(FALSE, TRUE, FALSE)]
```

```
## $int
## [1] 1 2 3 4 5
```

```
## By negative integers
myList[-c(1, 3)]
```

```
## $int
## [1] 1 2 3 4 5
```

2.13.3 Matrices

```
mat <- matrix(1:9, 3, 3)
colnames(mat) <- LETTERS[1:3]
mat
```

```
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
```

Possible to mix different sub-setting styles

```
mat[c(2, 3), c(1, 2)]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[-1, -3]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[c(2, 3), c("A", "B")]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[c(FALSE, TRUE, TRUE), c(TRUE, TRUE, FALSE)]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[-1, 1:2]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[c(FALSE, TRUE, TRUE), c("A", "B")]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

2.13.4 Data Frames

Data frames share properties of lists and matrices. sub-setting by a single vector will behave as in lists. Note the number of elements in a list is the number of columns in the data frame. sub-setting by two vectors will behave as in matrices.

```r
df <- data.frame(matrix(1:6, 2, 2))
names(df) <- letters[1:2]
df[1:2]
```

```
##   a b
## 1 1 3
## 2 2 4
```

```r
df[-1, c("a", "b")]
```

```
##   a b
## 2 2 4
```

2.13.5 By accessors:

The `[[` and `$` operators behave more like accessors to objects. They return only one object and as such can take generally only one input, so they are useful for lists and data frames. The `$` is a shortcut for `x[[name, exact = FALSE]]`

```r
names(df) <- c("Ak", "B")
df
```

```
##   Ak B
## 1  1 3
## 2  2 4
```

```
df[[2]]
```

```
## [1] 3 4
```

```
df[["B"]]
```

```
## [1] 3 4
```

```
What will the following return?
```

```
df$A
```

```
What will the following return?
```

```
df$A
```

```
## [1] 1 2
```

```
df[["A", exact = TRUE]]
```

```
## NULL
```

2.13.6 Sub-setting and assignment

All sub-setting operators can be combined with assignments to modify a subset of values.

```
x <- c(a = 1, b = 2, c = 3)
x[2] <- 10
x
```

```
##  a  b  c
##  1 10  3
```

```
df <- data.frame(a = 1:2, b = 3:4)
df
```

```
##   a b
## 1 1 3
## 2 2 4
```

```
df$b <- c("first", "second")
df
```

```
##   a      b
## 1 1  first
## 2 2 second
```

2.14 Functions

Function are a nice way to break down a big problem (bake a pizza) to smaller, manageable sub-problems (make the dough, make the sauce, add toppings, bake the pizza). They offer code re-usage (save time for important things, like wine drinking), easier testing of functionality and cleaner code!

The syntax to write your own functions is the following:

```
myFunction <- function(args1, args2 = default, ...) {
    body
    return(result)
}
```

- If the `return` statement is missing, the result of the last line will be returned
- The parameters can be assigned default values
- If the default value is a vector, the first value will be used by default
- The *three dots* (`...`) in the function arguments are called **ellipsis** and are used as a placeholder for any arguments passed to a function call inside. This is useful when you don't want to push all parameters from embodied function to your functions or allow for generic functions without pre-specifying argument names.

```
myFunction <- function(args1, args2 = c("option1", "option2"), ...) {
    result <- doSomething(args1, ...)
    return(result)
}
myFunction(args1, args2, param1)
```

2.15 Conditional Statements

A typical way to control the flow of the execution of your code is to have conditional statements.

The typical setup is:

```
if (statement_to_test) {
    print("Statement is true")
} else {
    print("Statement is false")
}
```

Long version:

```
if (1 < 2) {
    print("TRUE")
} else {
    print("FALSE")
}
```

```
## [1] "TRUE"
```

Short version if the To-Do step fits into one line:

```
{
    if (1 > 2)
        print("TRUE") else print("FALSE")
}
```

```
## [1] "FALSE"
```

The curly brackets around the if-else statement are only needed in a script to tell R to which *if* the *else* belongs to.

You can also have multiple statement tests. If more than one statements can be satisfied the first that got evaluated will be executed:

```
x <- 1
if (x > 0) {
    print("X is 0, or maybe not?")
} else if (x == 1) {
    print("X is 1")
} else {
    print("X is something else")
}
```

```
## [1] "X is 0, or maybe not?"
```

You can also chain *if-else* statements:

```r
x <- 5
if (x > 0) {
    if (x > 3) {
        if (x == 5) {
            cat("X is 5")
        } else {
            cat("X is greater than 3")
        }
    } else {
        cat("X positive and less than 3")
    }
} else {
    cat("X less than 0")
}
```

R also supports switch statements.

2.16 For loops

R supports looping through the traditional *for* and *while* statements and with the use of the *apply family* functions.

Syntax:

```
for(index in sequence) {
  do something
}
```

Again, if the To-Do step is a one-liner, curly brackets can be dropped

```r
a <- NULL
for (i in 1:10) a <- c(a, sum(1:i))
a
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

A sequence can be any vector, including lists

```r
myList <- list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
a <- integer(10)
for (i in myList) a[i] <- sum(1:i)
a
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

However, usually looping using *for/while* statements is more time consuming compared to the the functions of the *apply family*.

2.17 The Apply family

- For loops condition each following step on the previous step. That means that after each step the environment is updated.
- This leads to long running times (for the already slow looping mechanism of R).
- In case the single steps are independent of each other, a particular function can be applied to the elements of the steps.

```r
myList <- list(1:10, 11:20, 5:14)
sapply(myList, sum)
```

```
## [1]  55 155  95
```

- Also lambda functions can be applied

```
sapply(myList, function(y) {
    sum(y)/length(y)
})
```

```
## [1]  5.5 15.5  9.5
```

2.18 R Markdown

- This is an R Markdown presentation. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.
- Simply go to File –> New File –> R Markdown
- Select PDF and you get a template.
- You most likely won't need more commands than in on the first page of this cheat sheet.
- When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

2.19 Useful references

- These slides are mostly based on Hadley Wickham's book Advanced R
- **Everything else** from Hadley Wickham
- In-depth documentations:
  - Introduction to R
  - R language definition
  - R Internals
- Last but not least:
  - Stackoverflow

Chapter 3

Simple Data Manipulation and Visualization

3.1 Why plotting?

Data in scientific publications are shown as plots because on their raw form they cannot give us the full picture. Other times, plotting the data can give us hints about bugs in our code (or even in the data!) and can help us to state-of-the-art methods. But even plotting on its one is not enough. One need but the statistical and the visualization part.

Do you see a pattern in these plots?

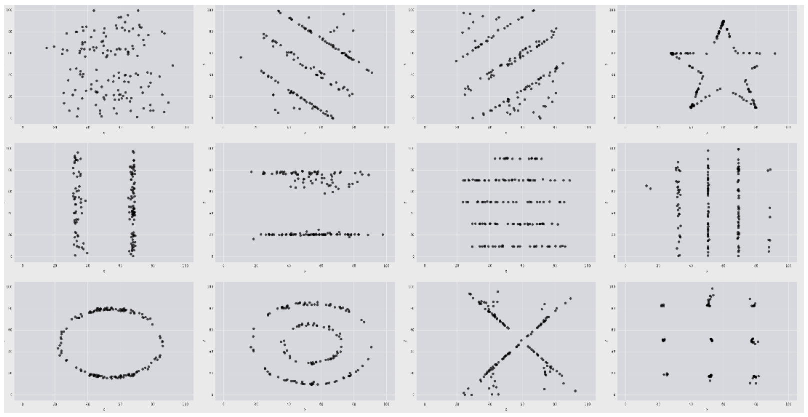


Figure 3.1: Random datasets

Maybe now?

Well all those plots, including the infamous datasaurus, share the same statistics!

So this back and forth is a constant procedure that one has to go through when doing data science.

Here is a realistic to warm up. A vector containing (hypothetical) height measurements for adults in Germany:

```r
length(height)
```

```
## [1] 500
```

```r
head(height, n = 20)
```

```
##  [1] 1.786833 1.715319 1.789841 1.787259 1.748659 1.660702 1.688214 1.716738
```

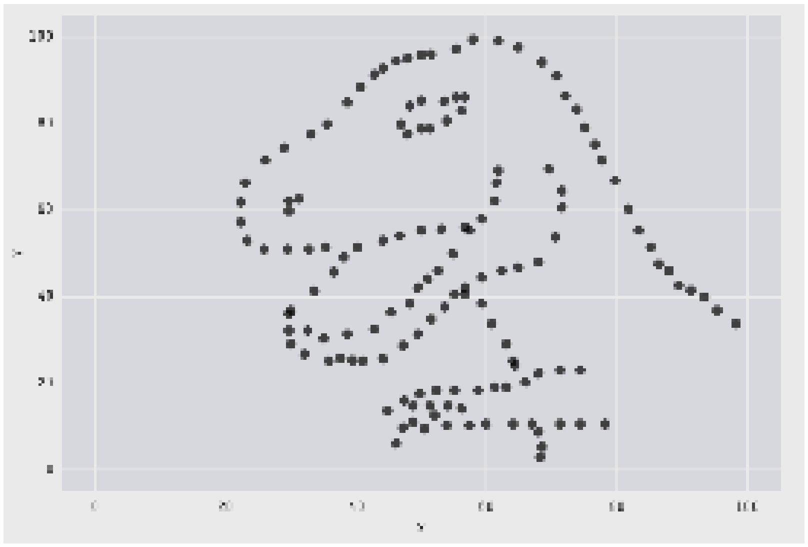
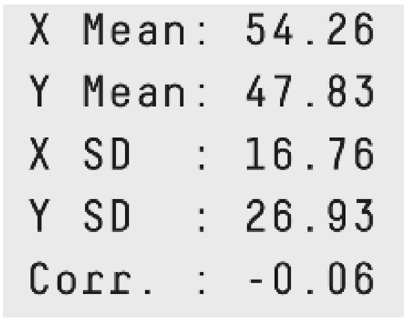


Figure 3.2: Datasaurus tex



```
X Mean: 54.26
Y Mean: 47.83
X SD  : 16.76
Y SD  : 26.93
Corr. : -0.06
```

Figure 3.3: Same statistics, different datasets

```
##  [9] 1.729740 1.838209 1.764362 1.694045 1.678355 1.716974 1.716535 1.711482
## [17] 1.741350 1.689864 1.749606 1.674311
```

Calculating the mean height:

```
mean(height)
```

```
## [1] 2.056583
```

Wait... what?

What happened?

A) `mean()` is not the right function to assess what we want to know.

B) Adults in Germany are exceptionally tall

C) A decimal point error in one data point.

D) It's a multiple testing problem because we are looking at so many data points (n=500).

Solution

```
mean(height)
```

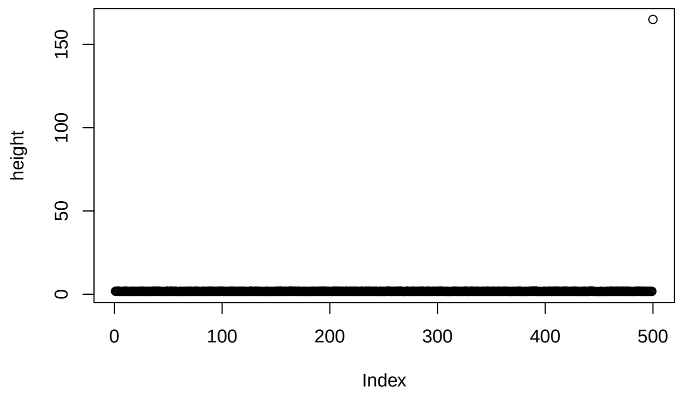
```
## [1] 2.056583
```

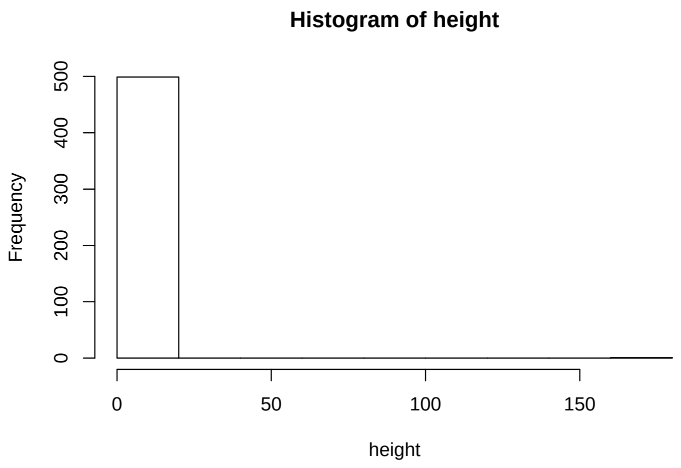
What happened?

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. **Correct!**
D) It's a multiple testing problem because we are looking at so many data points (n=500). *This question was intentionally misleading, this does not have anything to do with multiple testing.*

Let's see what will happen if we try to plot the data...

```
plot(height)
hist(height)
```





Interestingly, there is an outlier in our data! One particular person seems to have a height above 150 meters that it is obviously wrong! As a result, it inflates our mean giving us the false impression that Germans are exceptionally tall!

```
mean(height)
```

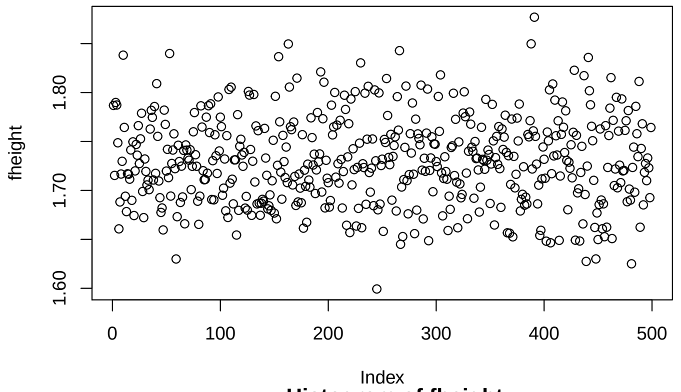
```
## [1] 2.056583
```

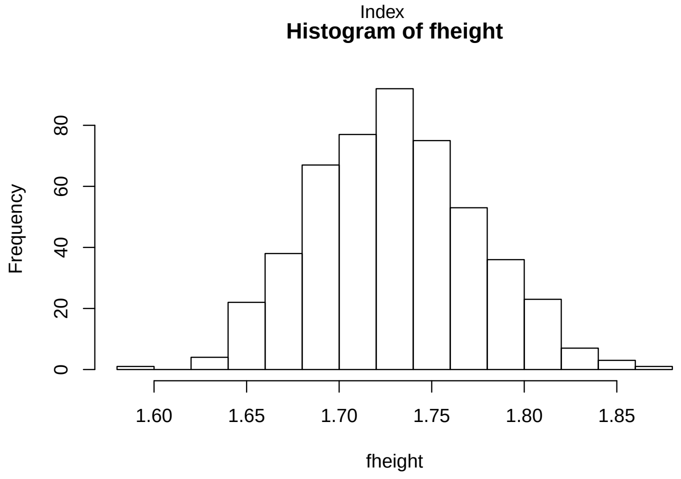
A quick way to fix our dataset is to remove our outlier.

```
fheight <- height[height < 3]
```

Now our data seem more realistic.

```
plot(fheight)
hist(fheight)
```





```
mean(fheight)
```

```
## [1] 1.730043
```

This is how the broken dataset was generated:

```
height <- c(rnorm(499, mean = 1.73, sd = 0.045), 165)
```

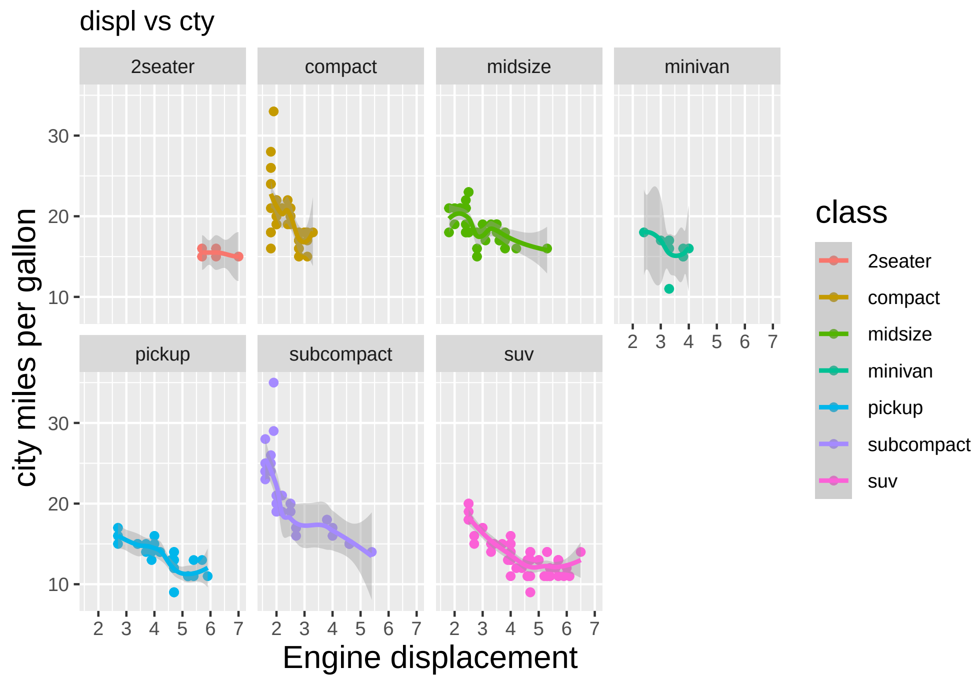
3.2 Grammar of Graphics

The Grammar of Graphics is a visualization theory developed by Leland Wilkinson in 1999. It was influenced the development of graphics and visualization libraries alike and base d on 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, box-plots, etc.)
- Composition of these common elements (one can combine elements as layers)

Now let's try to create a sophisticated example.

```
ggplot(mpg, aes(x = displ, y = cty, colour = class)) + geom_point() + facet_wrap(~class,
    ncol = 4) + theme(axis.title = element_text(size = 15), legend.title = element_text(size = 15)) +
    labs(title = "displ vs cty", x = "Engine displacement", y = "city miles per gallon") +
    stat_smooth()
```



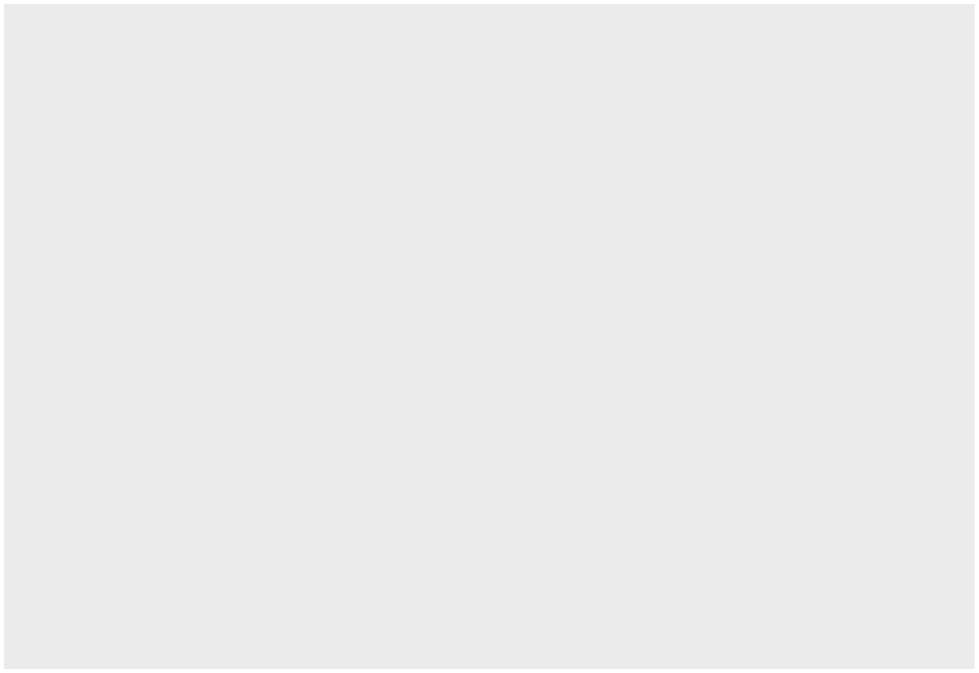
This plot shows the relationship of between the number of miles per gallon inside the city and the engine displacement for each type of car. One can use such graph to help him or her decide which car to purchase for cheap and environmental friendly rides inside the city.

How to create such a sophisticated plot step by step?

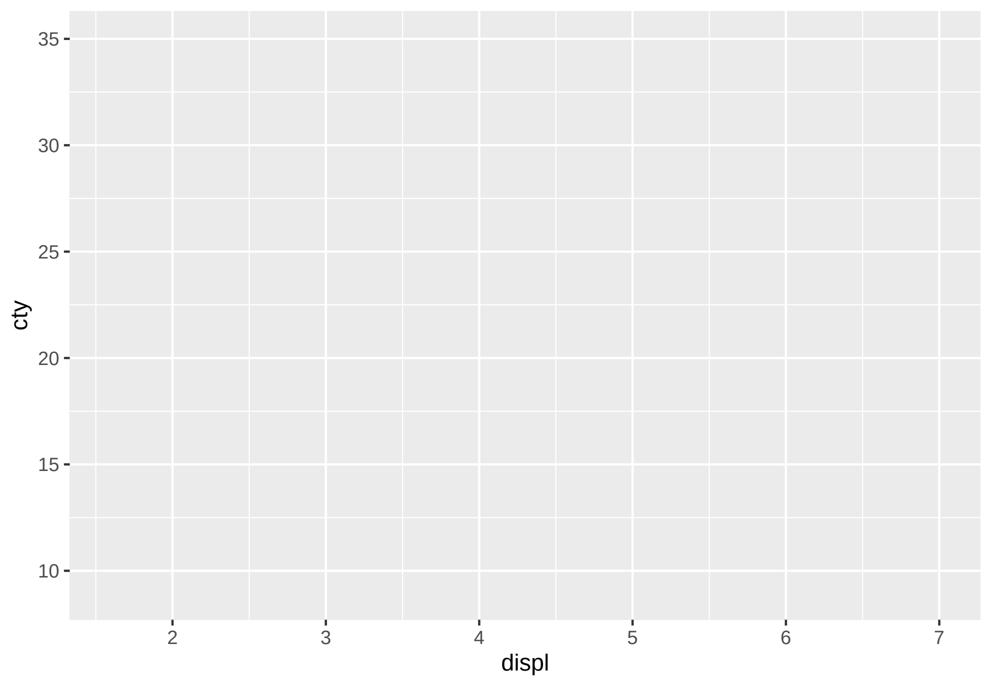
```
# quick look at the data, we want to plot cty against displ
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model    displ  year   cyl trans      drv     cty   hwy fl    class
##    <chr>        <chr>    <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
##  1 audi         a4         1.8  1999     4 auto(l~    f        18    29 p     comp~
##  2 audi         a4         1.8  1999     4 manual~    f        21    29 p     comp~
##  3 audi         a4         2     2008     4 manual~    f        20    31 p     comp~
##   4 audi         a4         2    2008     4 auto(a~ f         21    30 p     comp~
##   5 audi         a4         2.8  1999     6 auto(l~ f         16    26 p     comp~
##   6 audi         a4         2.8  1999     6 manual~ f         18    26 p     comp~
##   7 audi         a4         3.1  2008     6 auto(a~ f         18    27 p     comp~
##   8 audi         a4 quat~   1.8  1999     4 manual~ 4         18    26 p     comp~
##   9 audi         a4 quat~   1.8  1999     4 auto(l~ 4         16    25 p     comp~
##  10 audi         a4 quat~   2    2008     4 manual~ 4         20    28 p     comp~
## # ... with 224 more rows
```

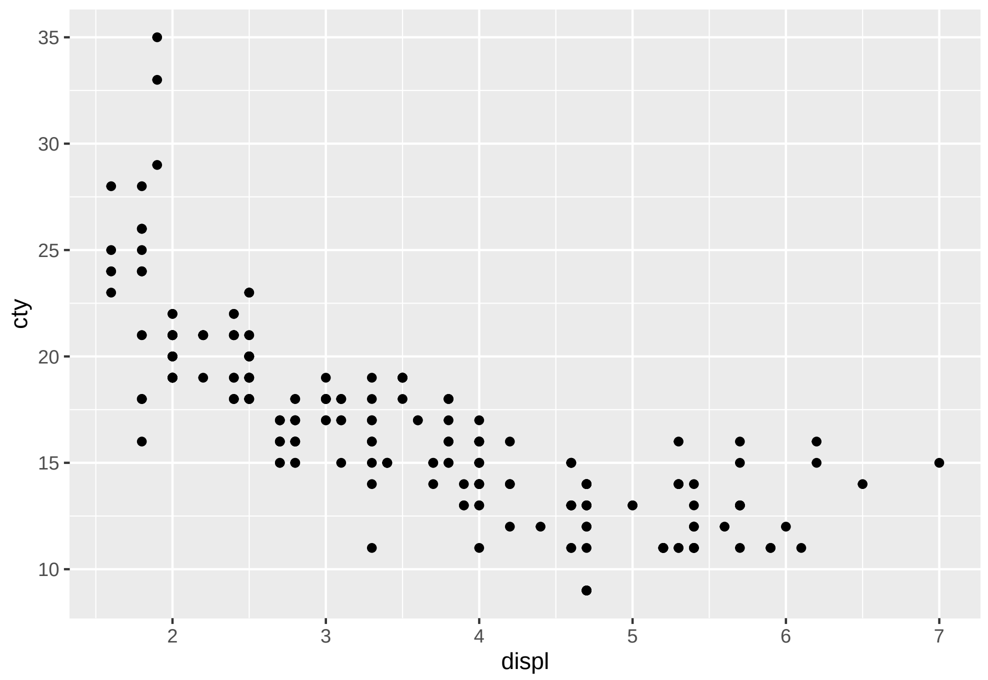
```
# only background
ggplot()
```



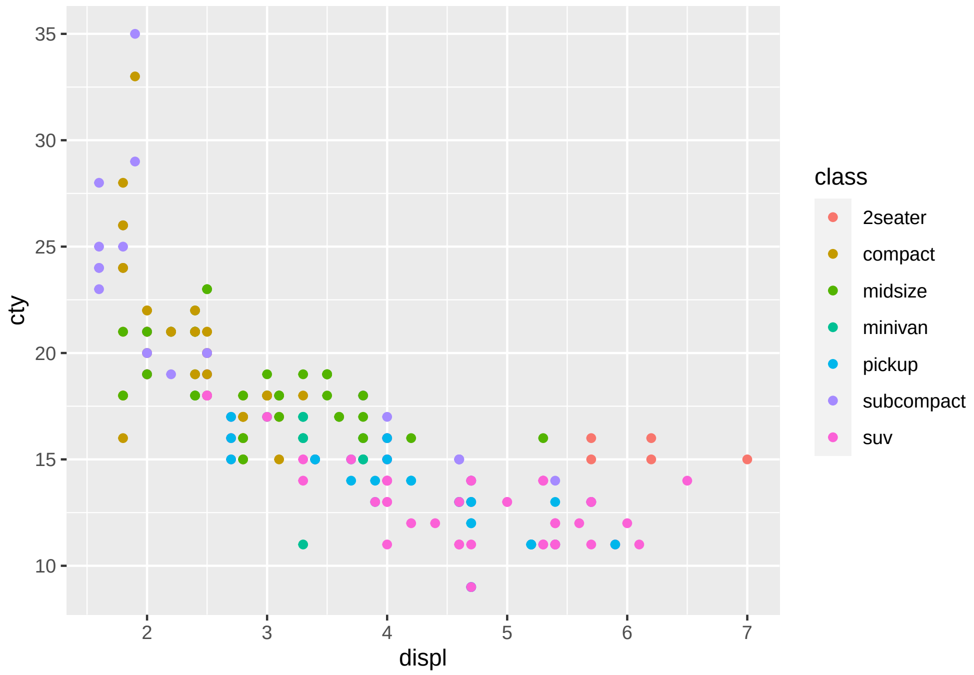
```
# blank plot

# data: how variables in the data are mapped to aesthetic attributes
ggplot(data = mpg, aes(x = displ, y = cty, colour = class))
```



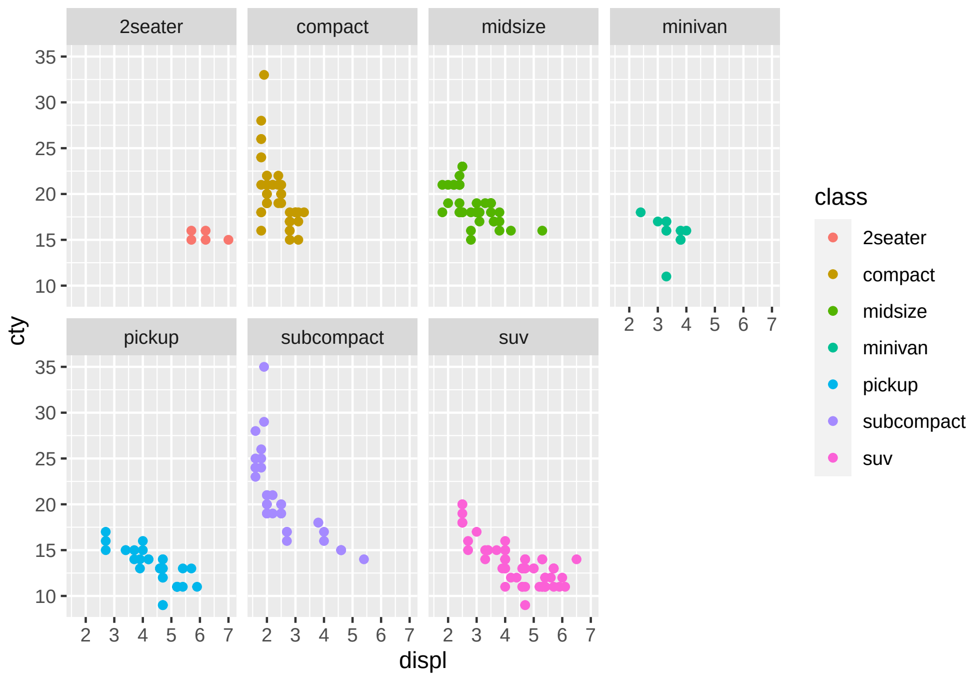
```r
# add layer of point

# layers: made up of geometric elements and statistical transformation.
ggplot(data = mpg, aes(x = displ, y = cty)) + geom_point()
```



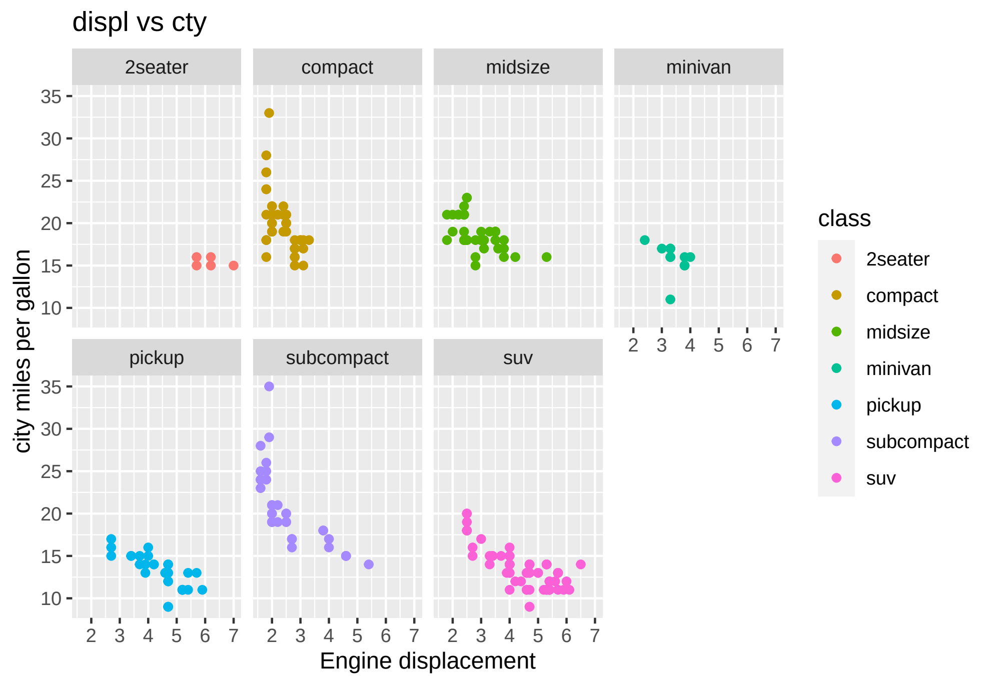
```r
# add color to the point layer
ggplot(data = mpg, aes(x = displ, y = cty, colour = class)) + geom_point()
```



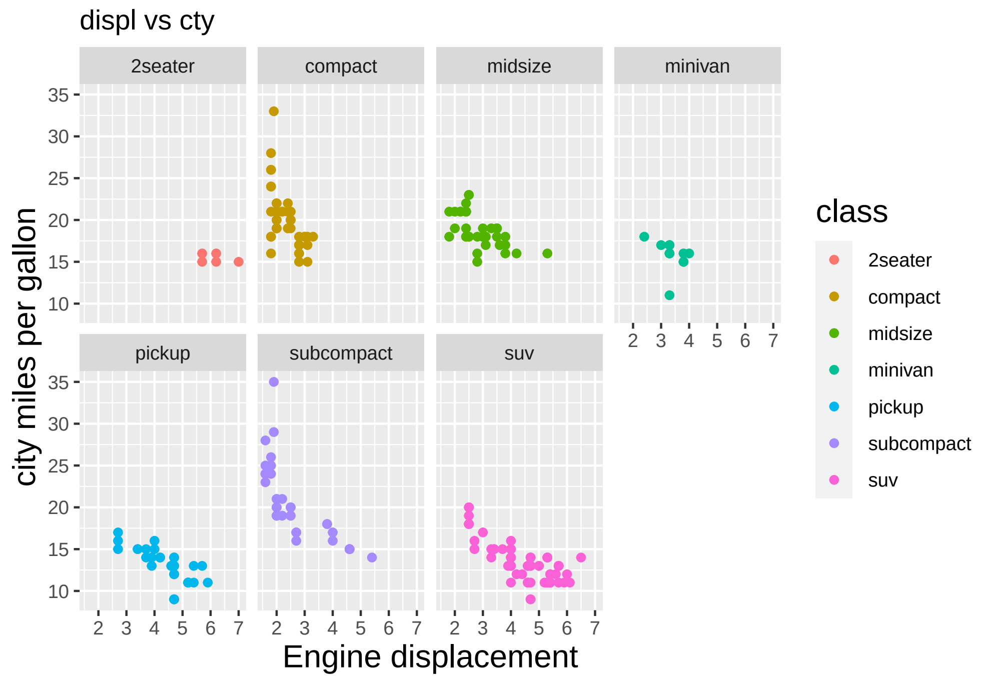
```r
# add facet
ggplot(data = mpg, aes(x = displ, y = cty, colour = class)) + geom_point() + facet_wrap(~class,
    ncol = 4)
```



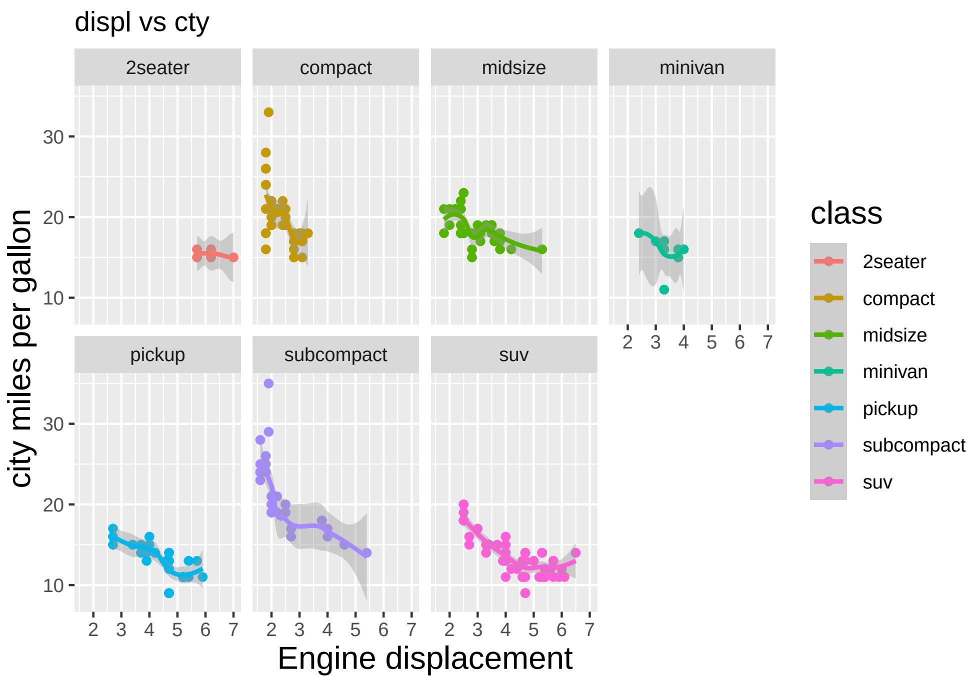
```r
# add labels
ggplot(data = mpg, aes(x = displ, y = cty, colour = class)) + geom_point() + facet_wrap(~class,
    ncol = 4) + labs(title = "displ vs cty", x = "Engine displacement", y = "city miles per gallon")
```



```
# make the font size larger
ggplot(data = mpg, aes(x = displ, y = cty, colour = class)) + geom_point() + facet_wrap(~class,
    ncol = 4) + labs(title = "displ vs cty", x = "Engine displacement", y = "city miles per gallon") +
    theme(axis.title = element_text(size = 15), legend.title = element_text(size = 15))
```



```
# add some statistics
ggplot(data = mpg, aes(x = displ, y = cty, colour = class)) + geom_point() + facet_wrap(~class,
    ncol = 4) + theme(axis.title = element_text(size = 15), legend.title = element_text(size = 15)) +
    labs(title = "displ vs cty", x = "Engine displacement", y = "city miles per gallon") +
    stat_smooth()
```



3.3 Grammar Defines Components of Graphics

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

Aesthetics: describes visual characteristics that represent data (`aes`) - for example: position, size, color, shape, transparency, fill

Layers: made up of geometric objects that represent data (`geom_`) - for example: points, lines, polygons, . . .

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

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

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

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

Here is a simple example: Human Development versus Corruption Perception

```r
ind <- fread("../../extdata/CPI_HDI.csv")
ind
```

```
##       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
##   4:   4      Angola    AGO  19 0.532               SSA
##   5:   5   Argentina    ARG  34 0.836          Americas
##  ---
## 147: 147     Uruguay    URY  73 0.793          Americas
## 148: 148  Uzbekistan    UZB  18 0.675 East EU Cemt Asia
## 149: 149       Yemen    YEM  19 0.498              MENA
## 150: 150      Zambia    ZMB  38 0.586               SSA
## 151: 151    Zimbabwe    ZWE  21 0.509               SSA
```

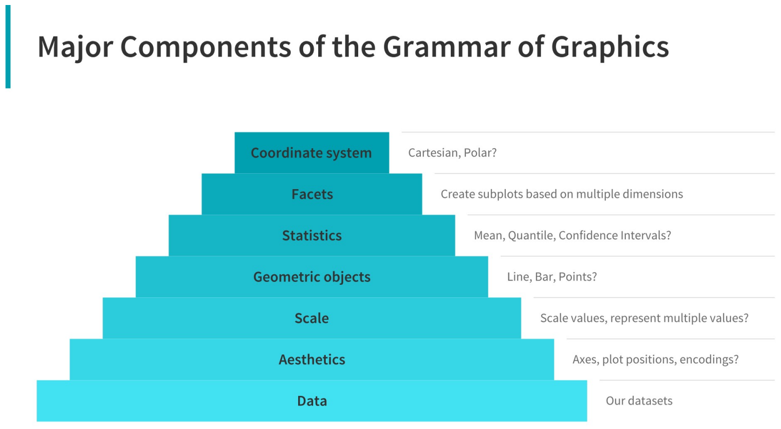

Major Components of the Grammar of Graphics
Coordinate system
Cartesian, Polar?
Facets
Create subplots based on multiple dimensions
Statistics
Mean, Quantile, Confidence Intervals?
Geometric objects
Line, Bar, Points?
Scale
Scale values, represent multiple values?
Aesthetics
Axes, plot positions, encodings?
Data
Our datasets

Figure 3.4: A representation of the abstraction of grammar of graphics

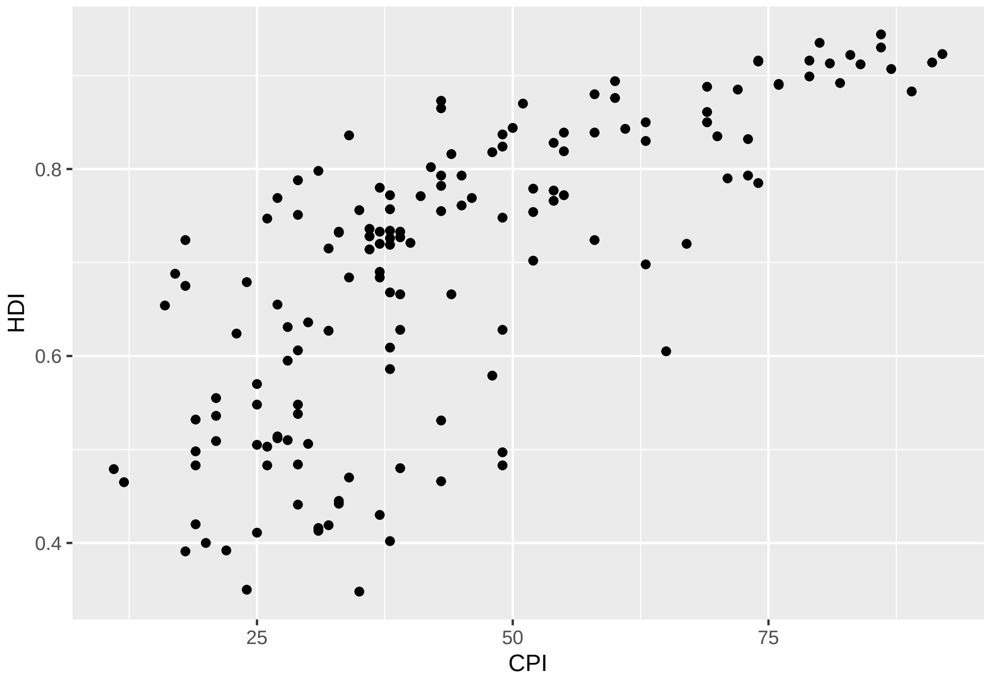
CPI: Corruption Perceptions Index (http://www.transparency.org/)

HDI: Human Development Index (http://hdr.undp.org/)

Year: 2014

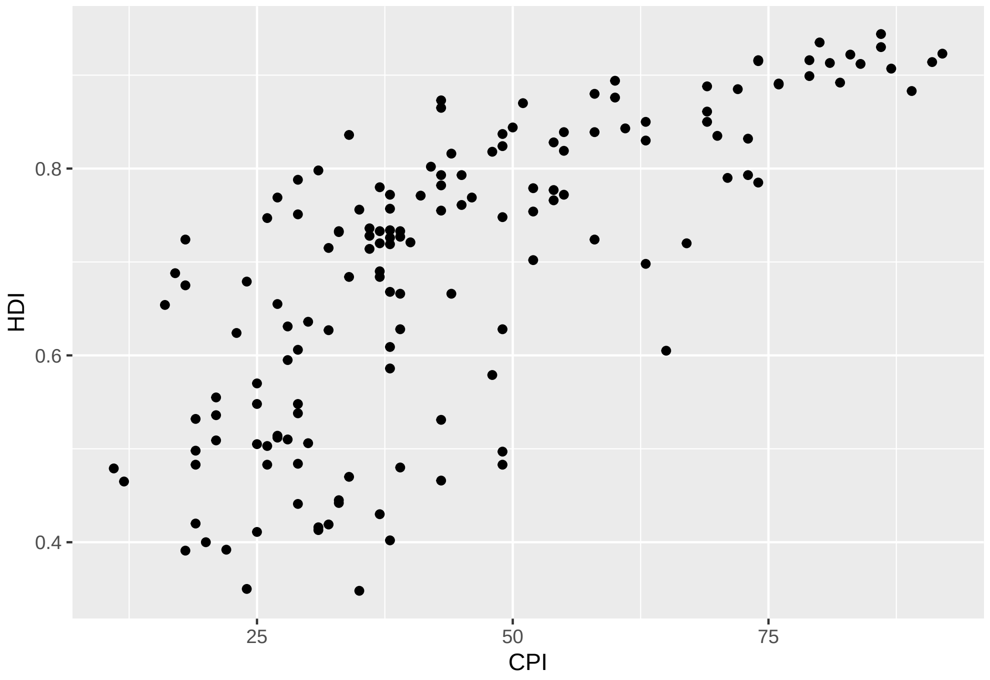
Lets start with a simple scatter plot

```r
ggplot(ind, aes(CPI, HDI)) + geom_point()
```



ggplot returns an object which can be stored and further edited

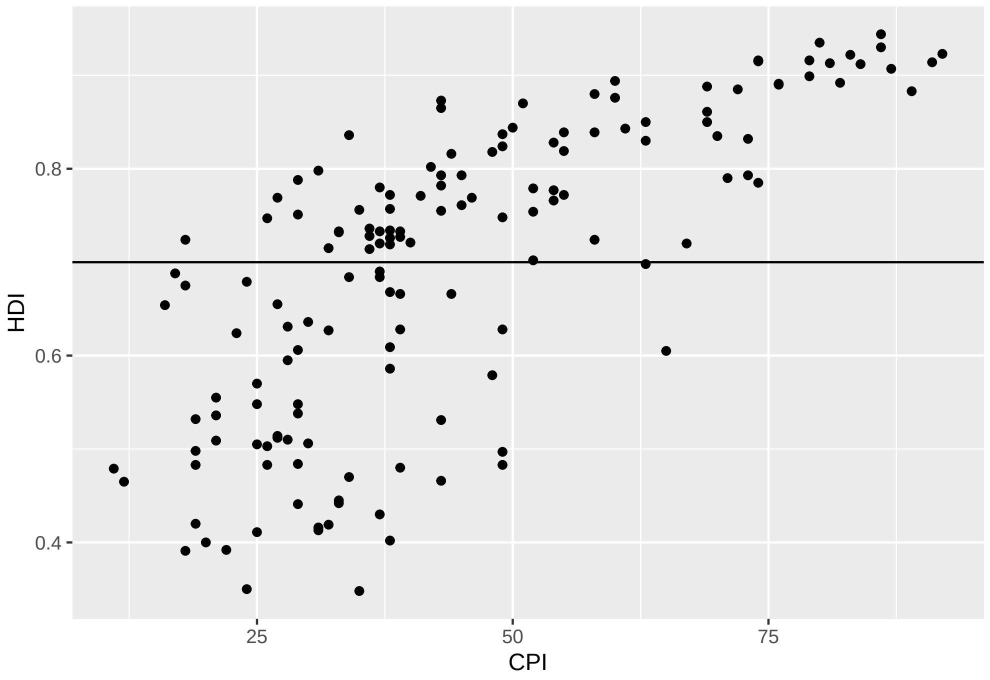
```r
p <- ggplot(ind, aes(CPI, HDI)) + geom_point()
p
```



```r
names(p)
```

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

```r
saveRDS(p, "../../extdata/lec06_p.rds")
p <- readRDS("../../extdata/lec06_p.rds")
p + geom_hline(yintercept = 0.7)
```

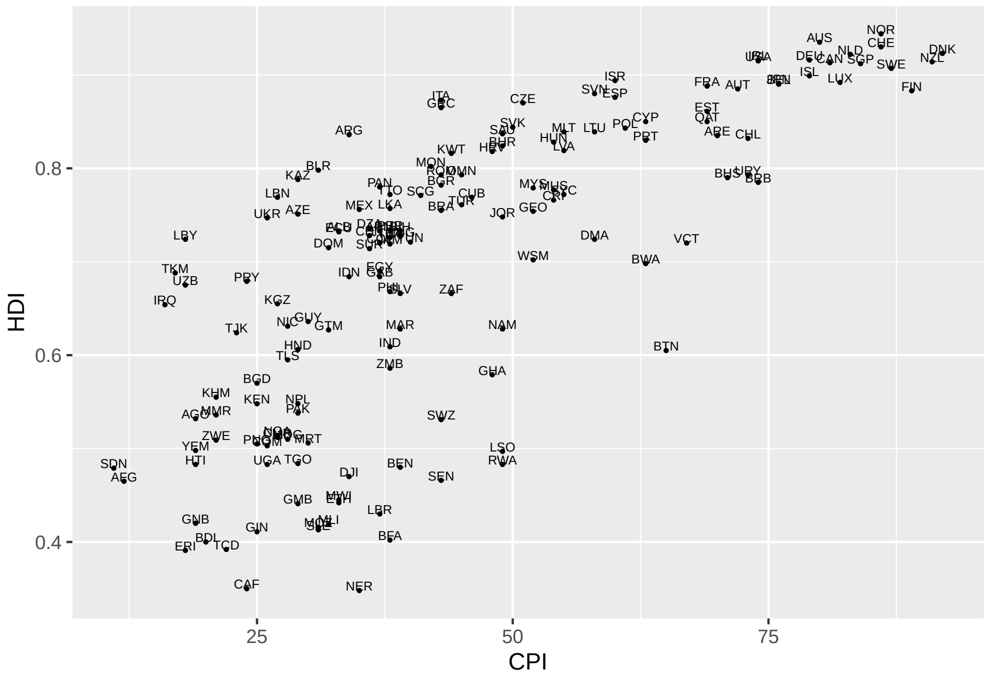


3.4 Mapping of aesthetics, colors, sizes and shapes

Mapping of aesthetics can be done globally at `ggplot()` or at individual layers

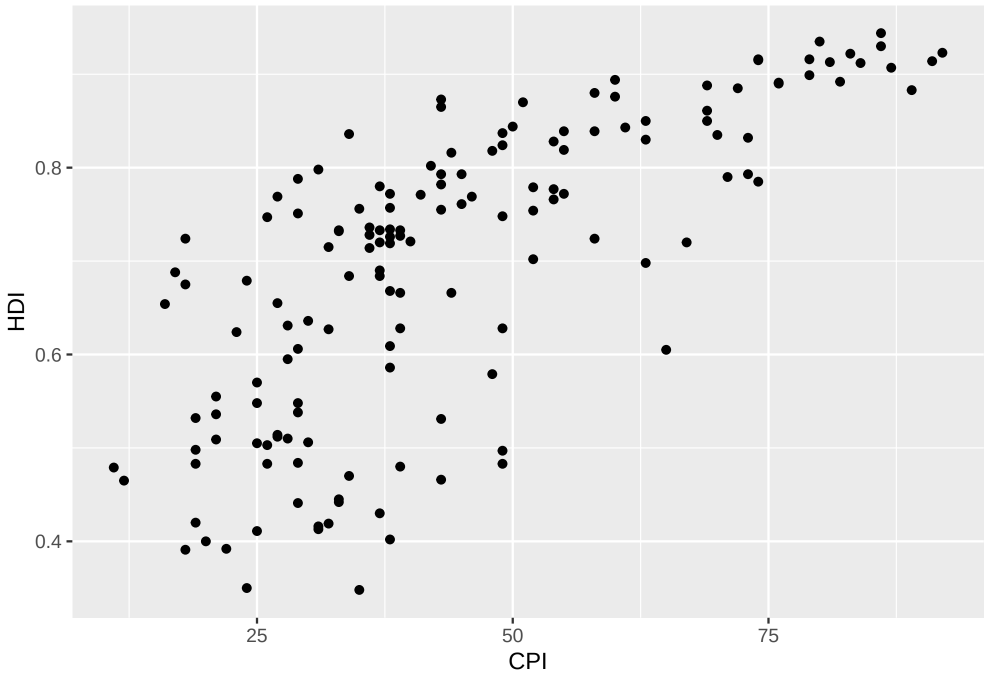
You can code like this globally.

```r
ggplot(ind, aes(CPI, HDI)) + geom_point(size = 0.5) + geom_text(aes(label = wbcode),
    size = 2, vjust = 0)
```



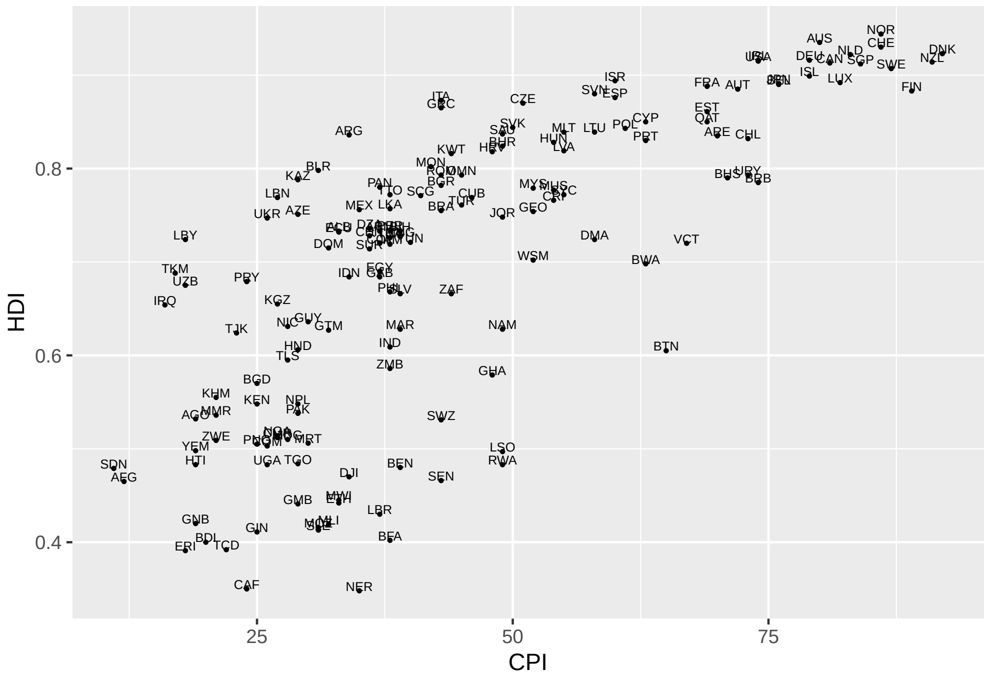
Or you can code like this for each layer.

```r
ggplot(ind) + geom_point(aes(CPI, HDI))
```



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

```r
ggplot(ind) + geom_point(aes(CPI, HDI), size = 0.5) + geom_text(aes(CPI, HDI, label = wbcode),
    size = 2, vjust = 0)
```

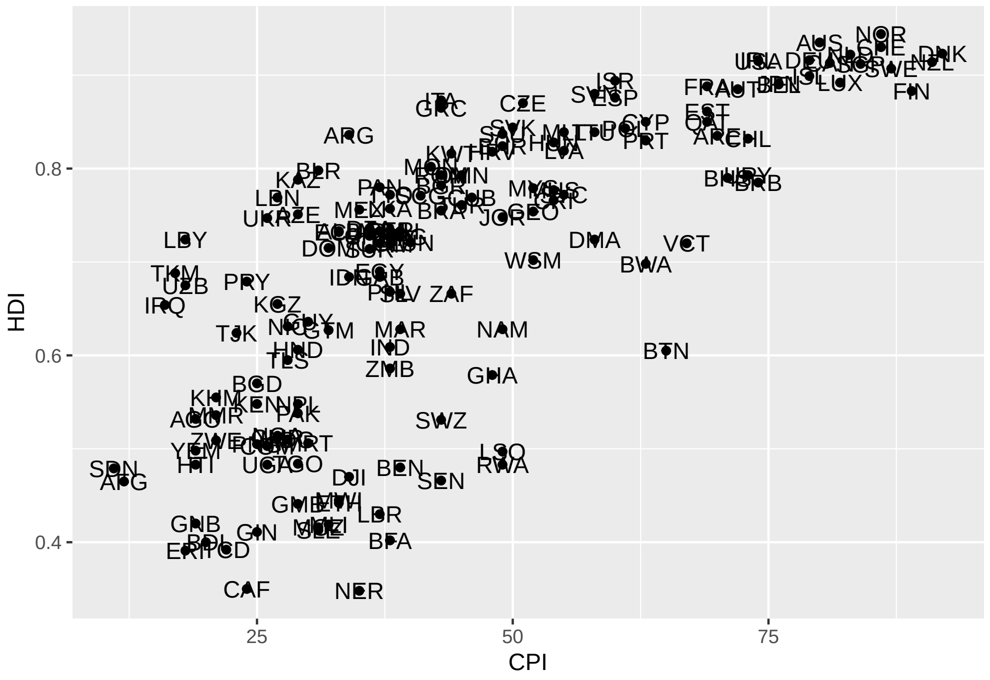


Individual layer mapping cannot be recognized by other layers

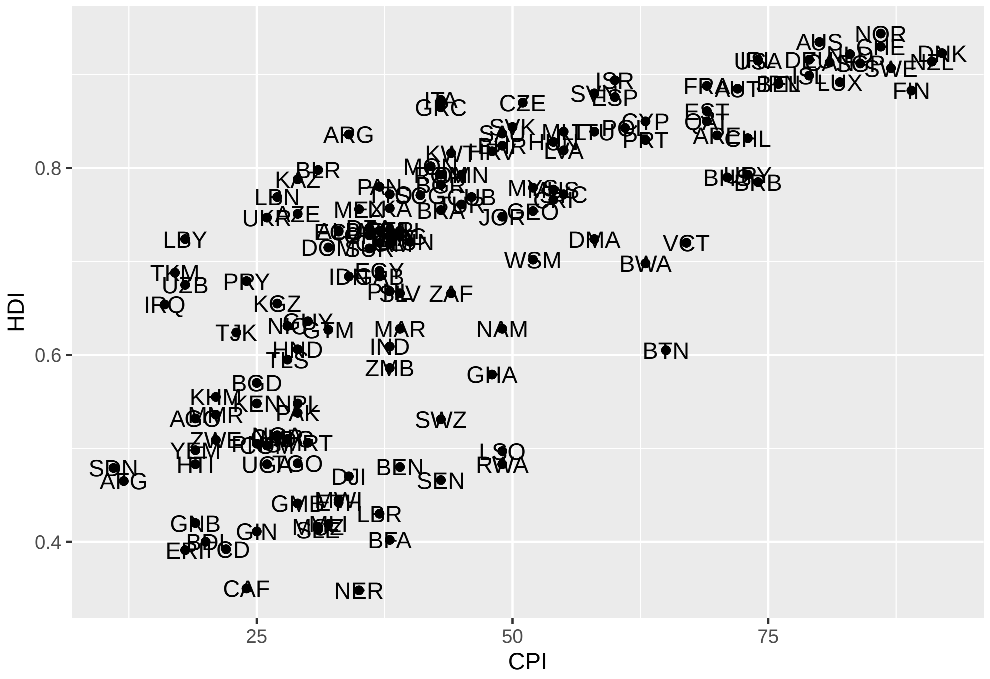
```
# this doesn't work as geom_text didn't know aes(x = CPI, y = HDI)
ggplot(ind) + geom_point(aes(x = CPI, y = HDI)) + geom_text(aes(label = wbcode))
```

```
## Error: geom_text requires the following missing aesthetics: x and y
```

```
# this would work but too rebundant
ggplot(ind) + geom_point(aes(x = CPI, y = HDI)) + geom_text(aes(x = CPI, y = HDI,
    label = wbcode))
```



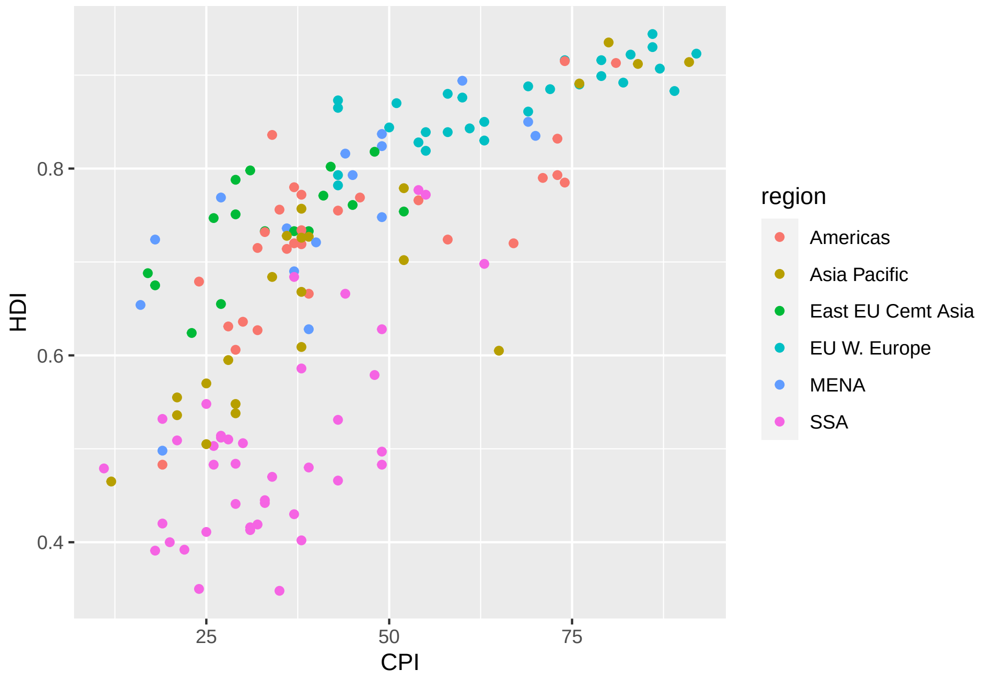
```r
# the common aes(x = CPI, y = HDI) shared by all the layers can be put in the
# ggplot()
ggplot(ind, aes(x = CPI, y = HDI)) + geom_point() + geom_text(aes(label = wbcode))
```



You can easily map variables to different colors, sizes or shapes!

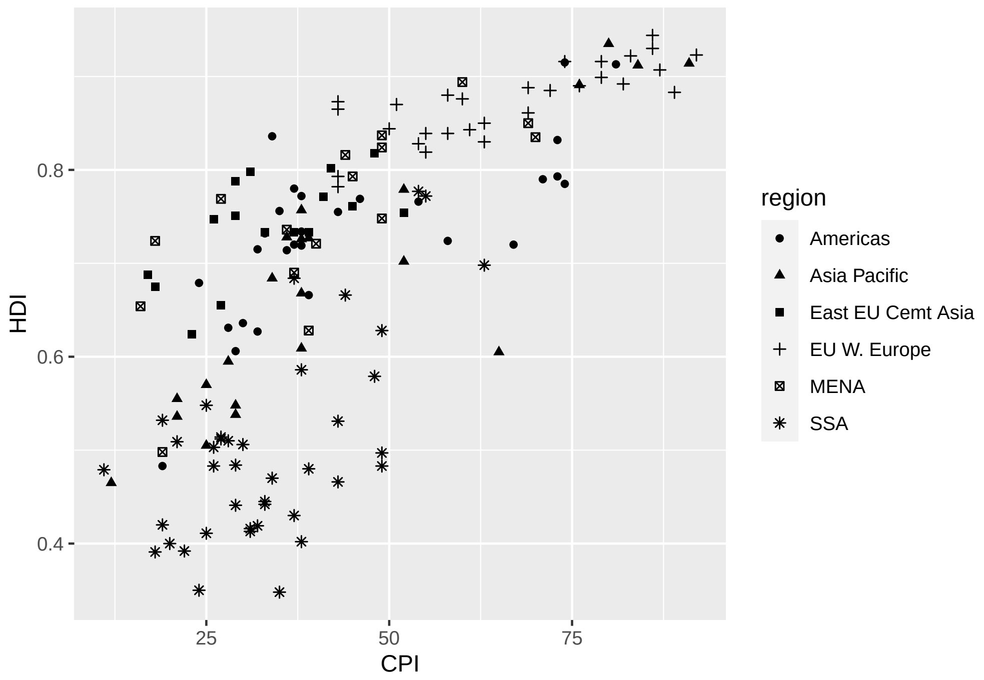
ggplot2 automatically scales for you.

```r
ggplot(data = ind) + geom_point(aes(CPI, HDI, color = region))
```



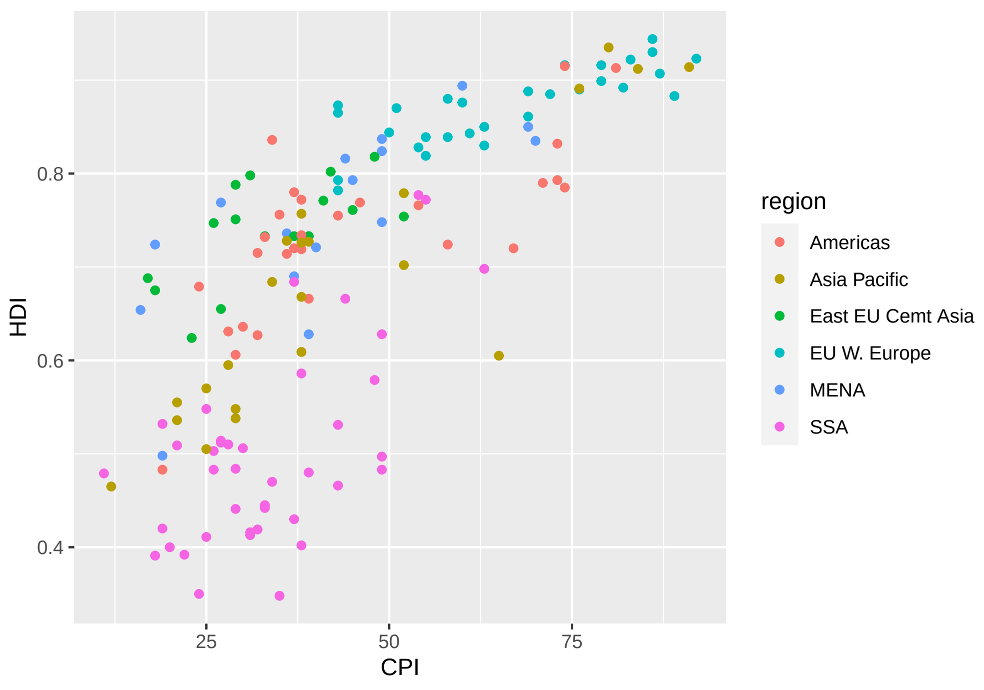
American `color` or British `colour` both acceptable.

```
ggplot(data = ind) + geom_point(aes(CPI, HDI, shape = region))
```



Aesthetic mappings can also be supplied in individual layers

```
ggplot(ind, aes(CPI, HDI)) + geom_point(aes(color = region))
```



3.5 Plotting libraries

- http://www.r-graph-gallery.com/portfolio/ggplot2-package/
- http://ggplot2.tidyverse.org/reference/
- https://plot.ly/r/
- https://plot.ly/ggplot2/

Chapter 4

Data.table

Similar to a data.frame, but because it modifies columns by *reference*, a data.table is a memory efficient and faster implementation of data.frame offering:

- sub-setting
- ordering
- merging

It accepts all data.frame functions, it has a shorter and more flexible syntax (not so straightforward in the beginning but pays off) and saves time on two fronts:

- programming (easier to code, read, debug and maintain)
- computing (fast and memory efficient)

The general form of data.table syntax is:

```
DT[ i,  j,  by ] # + extra arguments
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do?
     ---> on which rows?
```

The way to read this out loud is: "Take DT, subset rows by i, then compute j grouped by by. Here are some basic usage examples expanding on this definition.

Like data.frame, data.table is a list of vectors. It doesn't have rownames; instead, it's a collection of attributes.

Each column can be a different type, including a list

It has enhanced functionality in `[`. We can make operations inside `[]`.

For a detailed documentation go check the help page for data.table!

To create a data.table we just name its columns and then we populate them. As before with the data.frame, all the columns have to have the same length.

```r
# install.packages('data.table')
library(data.table)
DT <- data.table(x = rep(c("a", "b", "c"), each = 3), y = c(1, 3, 6), v = 1:9)
DT  # note how column y was recycled
```

```
##    x y v
## 1: a 1 1
## 2: a 3 2
## 3: a 6 3
## 4: b 1 4
## 5: b 3 5
## 6: b 6 6
## 7: c 1 7
## 8: c 3 8
## 9: c 6 9
```

```
# For comparison, let's make a data.frame from it:
DF <- as.data.frame(DT)
```

If we want to converting a data frame to data.table all we have to do it to call the `as.data.table()` function.

```
# Using the data frame from the previous slide:
DT <- as.data.table(DF)
class(DT)
```

```
## [1] "data.table" "data.frame"
```

Here you can see that the class function informs us that DT is both a data.table and a data.frame.

The structure of the data.table is that essentially is a list of list. To get the dimensions of our data.table we can either use the `dim()` function to get the dimensions of our data.table (M rows x N columns) or to use the `ncol()` `nrow()` functions.

```
class(DT)
```

```
## [1] "data.table" "data.frame"
```

```
is.list(DT)
```

```
## [1] TRUE
```

```
summary(DT)
```

```
##       x                   y               v    
##  Length:9           Min.   :1.000   Min.   :1  
##  Class :character   1st Qu.:1.000   1st Qu.:3  
##  Mode  :character   Median :3.000   Median :5  
##                     Mean   :3.333   Mean   :5  
##                     3rd Qu.:6.000   3rd Qu.:7  
##                     Max.   :6.000   Max.   :9  
```

```
nrow(DT)  # ncol(DT)
```

```
## [1] 9
```

```
dim(DT)
```

```
## [1] 9 3
```

4.1 Accessing a data.table by rows (*i*)

- The general form of data.table syntax is `DT[i, j, by]`
- "Take **DT**, subset rows using **i**, then select or calculate **j**, grouped by **by**".

```
DT[2, ]  # Access the 2nd row (also DT[2] or DT[i = 2])
```

```
##    x y v
## 1: a 3 2
```

```
DT[1:2]  # Access multiple consecutive rows.
```

```
##    x y v
## 1: a 1 1
## 2: a 3 2
```

```
DT[c(3, 5)]  # Access multiple rows.
```

```
##    x y v
## 1: a 6 3
## 2: b 3 5
```

4.2 Sub-setting rows according to some condition

We can subset a data table according to different conditions using the operators:

- `==`
- `<`
- `>`
- `!=`
- `%in%`

To exemplify this, load the predefined `iris` dataset as a data.table

```
iris_dt <- as.data.table(iris)
summary(iris_dt)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
##        Species  
##  setosa    :50  
##  versicolor:50  
##  virginica :50  
##                 
##                 
##                 
```

```
# View the species setosa. Data frame syntax: iris[iris$Species == 'setosa', ]
# Note how we use double [] to subset
iris_dt[Species == "setosa"][4:6]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1:          4.6         3.1          1.5         0.2  setosa
## 2:          5.0         3.6          1.4         0.2  setosa
## 3:          5.4         3.9          1.7         0.4  setosa
```

```
# View all the species except setosa Same as iris_dt[Species %in% c('versicolor',
# 'virginica')]
iris_dt[Species != "setosa"][1:3]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1:          7.0         3.2          4.7         1.4 versicolor
## 2:          6.4         3.2          4.5         1.5 versicolor
## 3:          6.9         3.1          4.9         1.5 versicolor
```

```
# View all flowers with a petal length greater than 6
iris_dt[Petal.Length > 6][1:3]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1:          7.6         3.0          6.6         2.1 virginica
## 2:          7.3         2.9          6.3         1.8 virginica
## 3:          7.2         3.6          6.1         2.5 virginica
```

4.3 Accessing a data.table by columns (DT[i, j, by])

Once you're inside the `[]`, you're in the data.table environment. Inside this scope, there's no need to put the column names in quotation marks, as columns are seen as variables already.

It is not advisable to access a column by its number. **Use the column name instead**.

The table format can change (new columns can be added), so using column names prevents bugs e.g. if you have a dataset with 50 columns, how do you know which one is column 18?

This way the code is more readable: `DT[, age]` instead of `DT[, 5]`.

```
DT[, x]  # Access column x (also DT$x or DT[j=x]).
```

```
## [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
```

```
DT[4, x]  # Access a specific cell.
```

```
## [1] "b"
```

When accessing many columns, we probably want to return a data.table instead of a vector. For that, we need to provide R with a list, so we use `list(colA, colB)`, or its simplified version `.(colA, colB)`.

```
# Note that 1 and 3 were coerced into strings because a vector can have only 1
# type
DT[1:2, c(x, y)]
```

```
## [1] "a" "a" "1" "3"
```

```
# Access a specific subset. Data.frame: DF[1:2, c('x','y')]
DT[1:2, list(x, y)]
```

```
##    x y
## 1: a 1
## 2: a 3
```

```
# Same as before.
DT[1:2, .(x, y)]
```

```
##    x y
## 1: a 1
## 2: a 3
```

4.4 Data.table environment

The following examples are to show how the `[]` bring us inside the data.table environment.

Using DT and DF, compute the product of columns *y* and *v*.

Use the `with(data, expr, ...)` function, which evaluates an R expression in an environment constructed from `data`.

```
# We enter the environment of DT, and simply compute the product
with(DT, y * v)
```

```
# Easier way, with [] we're inside the environment. Data.table runs a
# with(dt,...) on the j argument
DT[, y * v]

# In data.frame, the [] doesn't imply we're inside the environment
DF[, y * v]
```

4.5 Basic operations

We saw already that inside the `[]`, columns are seen as variables, so we can apply functions to them.

```
# Similar to mean(iris_dt[, Petal.Length])
iris_dt[, mean(Petal.Length)]
```

```
## [1] 3.758
```

```
iris_dt[Species == "virginica", mean(Petal.Length)]
```

```
## [1] 5.552
```

To compute operations in multiple columns, we must provide a list (unless we want the result to be a vector).

```
# Same as iris_dt[, .(mean(PL), median(PL))]
iris_dt[, list(mean(Petal.Length), median(Petal.Length))]
```

```
##       V1   V2
## 1: 3.758 4.35
```

```
# Give meaningful names
iris_dt[, .(mean_PL = mean(Petal.Length), median_PL = median(Petal.Length))]
```

```
##    mean_PL median_PL
## 1:   3.758      4.35
```

We can also apply basic operations on multiple columns

`sapply(DT, FUN)`, applies function FUN column-wise to DT. Remember that `sapply` returns a vector, while `lapply` returns a list.

```
sapply(iris_dt, class)  # Try the same with lapply
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##    "numeric"    "numeric"    "numeric"    "numeric"     "factor"
```

```
sapply(iris_dt, sum)
```

```
## Error in Summary.factor(structure(c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, :
'sum' not meaningful for factors
```

```
# Note that we can access columns stored as variables by setting with=F.  In this
# case, `colnames(iris_dt)!='Species'` returns a logical vector and ` iris_dt` is
# subseted by the logical vector

# Same as sapply(iris_dt[, 1:4], sum)
sapply(iris_dt[, colnames(iris_dt) != "Species", with = F], sum)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        876.5        458.6        563.7        179.9
```

4.6 The ‘by’ option (DT[i, j, by])

A very useful option used to apply some function to every group of a data table is the by argument. Although usually `i` and `j` are omitted from the syntax, we do write the `by`.

```
# Compute the mean petal length of every species
iris_dt[, .(mean_PL = mean(Petal.Length)), by = Species]
```

```
##       Species mean_PL
## 1:     setosa   1.462
## 2: versicolor   4.260
## 3:  virginica   5.552
```

```
# Compute the mean and sd of petal length of every species
iris_dt[, .(mean_PL = mean(Petal.Length), sd_PL = sd(Petal.Length)), by = Species]
```

```
##       Species mean_PL     sd_PL
## 1:     setosa   1.462 0.1736640
## 2: versicolor   4.260 0.4699110
## 3:  virginica   5.552 0.5518947
```

4.7 The .N command

The `.N` is a special in-built variable that counts the number observations of a particular group.

```
# Get the number of observations for each species
iris_dt[, .N, by = Species]
```

```
##       Species  N
## 1:     setosa 50
## 2: versicolor 50
## 3:  virginica 50
```

Remember the data.table definition: “Take **DT**, subset rows using **i**, then select or calculate **j**, grouped by **by**”

```
# For each species, get the number of observations with Sepal.Width greater than
# 3
iris_dt[Sepal.Width > 3, .(SW_greater_3 = .N), by = Species]
```

```
##       Species SW_greater_3
## 1:     setosa           42
## 2: versicolor            8
## 3:  virginica           17
```

4.8 The := command

The `:=` operator updates the data.table you are working on, so writing DT <- DT[,. . . := . . .] is redundant. This operator, plus all `set` functions, change their input by *reference*. No copy of the object is made, that is why it is faster and uses less memory.

```
# Add a new column called yv whose value is the product of y and v
DT[, `:=`(yv, y * v)][1:3]
```

```
##    x y v yv
## 1: a 1 1  1
## 2: a 3 2  6
## 3: a 6 3 18
```

```r
# Add columns with sepal and petal area. Note the syntax of multiple assignment.
iris_dt[, `:=`(Sepal.Area = Sepal.Length * Sepal.Width, Petal.Area = Petal.Length *
    Petal.Width)][1:3]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Area
## 1:          5.1         3.5          1.4         0.2  setosa      17.85
## 2:          4.9         3.0          1.4         0.2  setosa      14.70
## 3:          4.7         3.2          1.3         0.2  setosa      15.04
##    Petal.Area
## 1:       0.28
## 2:       0.28
## 3:       0.26
```

You can also delete columns by using the := command.

```r
# Let's assume setosa flowers are orange, versicolor purple and virginica pink.
# Add a column with these colors.
iris_dt[Species == "setosa", `:=`(color, "orange")]
iris_dt[Species == "versicolor", `:=`(color, "purple")]
iris_dt[Species == "virginica", `:=`(color, "pink")]
unique(iris_dt[, .(Species, color)])
```

```
##       Species  color
## 1:     setosa orange
## 2: versicolor purple
## 3:  virginica   pink
```

```r
# We can delete this new column by setting it to NULL
iris_dt[, `:=`(color, NULL)]
colnames(iris_dt)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
## [6] "Sepal.Area"   "Petal.Area"
```

4.9 By reference

What do we mean when we say that data.table modifies columns *by reference*? That no new copy of the object is made in the memory, unless we actually create one using `copy()`.

```r
or_dt <- data.table(a = 1:10, b = 11:20)
# No new object is created, both new_dt and or_dt point to the same memory chunk.
new_dt <- or_dt
new_dt[, `:=`(ab, a * b)]
colnames(or_dt)  # or_dt was also affected by changes in new_dt
```

```
## [1] "a"  "b"  "ab"
```

```r
or_dt <- data.table(a = 1:10, b = 11:20)
copy_dt <- copy(or_dt)  # By creating a copy, we have 2 objects in memory
copy_dt[, `:=`(ab, a * b)]
colnames(or_dt)  # Changes in the copy don't affect the original
```

```
## [1] "a" "b"
```

4.10 Merging

Merge, join or combine 2 data tables into one by common column(s).

`merge(x, y, by, by.x, by.y, all, all.x, all.y, ...)`

In data.table, the default is to merge by shared *key* columns while in data.frame, the default is to merge by columns with the same name. We can also specify the columns to merge `by`.

There are different types of merging:

- **Inner (default)**: consider only rows with matching values in the key columns.
- **Outer or full**: 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.

4.10.1 Merging - inner and outer examples

```
(dt1 <- data.table(table = "table1", id = c(1, 2, 3), a = rnorm(3)))
```

```
##     table id          a
## 1: table1  1  0.7555092
## 2: table1  2 -0.7032523
## 3: table1  3 -0.2868516
```

```
(dt2 <- data.table(table = "table2", id = c(1, 2, 4), b = rnorm(3)))
```

```
##     table id          b
## 1: table2  1  1.8411069
## 2: table2  2 -0.1567643
## 3: table2  4 -1.3898026
```

```
# Inner merge: default one, all = FALSE
merge(dt1, dt2, by = "id", all = F)
```

```
##    id table.x          a table.y          b
## 1:  1  table1  0.7555092  table2  1.8411069
## 2:  2  table1 -0.7032523  table2 -0.1567643
```

```
# Outer (full) merge: all = TRUE
merge(dt1, dt2, by = "id", all = T)
```

```
##    id table.x          a table.y          b
## 1:  1  table1  0.7555092  table2  1.8411069
## 2:  2  table1 -0.7032523  table2 -0.1567643
## 3:  3  table1 -0.2868516    <NA>         NA
## 4:  4    <NA>         NA  table2 -1.3898026
```

Note that the column order got changed after the merging.

```
# Left merge: all.x = TRUE
merge(dt1, dt2, by = "id", all.x = T)[]
```

```
##    id table.x          a table.y          b
## 1:  1  table1  0.7555092  table2  1.8411069
## 2:  2  table1 -0.7032523  table2 -0.1567643
## 3:  3  table1 -0.2868516    <NA>         NA
```

```
# Right merge: all.y = TRUE
merge(dt1, dt2, by = "id", all.y = T)[]
```

```
##    id table.x          a table.y          b
## 1:  1  table1  0.7555092  table2  1.8411069
## 2:  2  table1 -0.7032523  table2 -0.1567643
## 3:  4    <NA>         NA  table2 -1.3898026
```

4.11 Summary

By now, you should be able to answer the following questions:

- Why do we say that data.table is an enhanced data.frame?
- How to subset by rows or columns? Remember: DT[i, j, by].
- How to add columns?
- How to make operations with different columns?
- Which are the different types of merging?

Even if you were able to answer them, practice:

- Check all vignettes and reference manual from https://cran.r-project.org/web/packages/data.table/
- A really concise cheat sheet:
- https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

4.12 Data.table resources

https://cran.r-project.org/web/packages/data.table/

https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

http://r4ds.had.co.nz/relational-data.html

http://adv-r.had.co.nz/Environments.html

Chapter 5

Tidy data

5.1 Tidy data prerequisites

```
library(data.table)  # melt, dcast, ...
library(magrittr)  # pipe operator %>%
suppressMessages(library(tidyr))  # table1, table2, ...

# Data used throughout the lecture
table1 <- tidyr::table1 %>% as.data.table  # use data.table instead of tibble
table2 <- tidyr::table2 %>% as.data.table
table3 <- tidyr::table3 %>% as.data.table
table4a <- tidyr::table4a %>% as.data.table
table4b <- tidyr::table4b %>% as.data.table
table5 <- tidyr::table5 %>% as.data.table
```

For clarity, we will often use the syntax: - `tidyr::unite()` - instead of just `unite()`

We will also use the pipe operator %>%

- conceptually similar to the unix pipe: `$ cat file.txt | sort -u | head`
- makes the code more readable
- typical use cases:
  - exploratory analysis at the command line: `dt %>% unique %>% head`
  - dplyr data table operations
- http://r4ds.had.co.nz/pipes.html

```
foo <- little_bunny()
```

```
foo <- hop(foo, through = forest)
foo <- scoop(foo, up = field_mice)
foo <- bop(foo, on = head)
```

```
foo %>% hop(through = forest) %>% scoop(up = field_mouse) %>% bop(on = head)
```

```
## Argument not occuring first: use .
"hop_scoop_bop" %>% gsub("_", " ", .)
```

```
## [1] "hop scoop bop"
```

5.2 Tidy data definition

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

```
table1
```

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

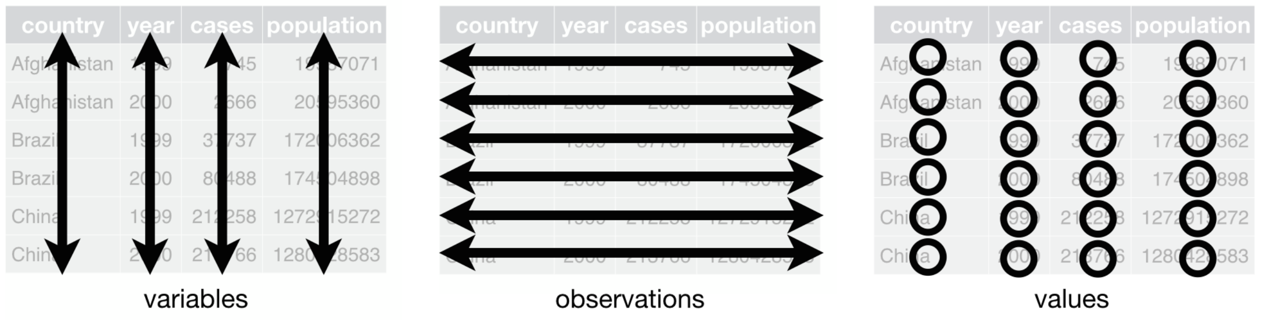


Figure 5.1: Data table layout

5.3 Common signs of untidy datasets

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

Column headers are values, not variable names

Untidy: 1999 and 2000 are values of the variable *year*.

```
table4a
```

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

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

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

5.3.1 Multiple variables are stored in one column

Untidy

```
table3
```

```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

5.3.2 Variables are stored in both rows and columns

- The variable *date* is stored across rows and columns
- The `element` column is not a variable; it stores the names of variables

Untidy: days of the month (d1, d2, . . .) are values, not variables

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	—	—	—	—	—	—	—	—

Tidy

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

5.3.3 Multiple types of observational units are stored in the same table

Untidy

year	artist	time	track	date	week	rank
2000	2 Pac	4:22	Baby Don't Cry	2000-02-26	1	87
2000	2 Pac	4:22	Baby Don't Cry	2000-03-04	2	82
2000	2 Pac	4:22	Baby Don't Cry	2000-03-11	3	72
2000	2 Pac	4:22	Baby Don't Cry	2000-03-18	4	77
2000	2 Pac	4:22	Baby Don't Cry	2000-03-25	5	87
2000	2 Pac	4:22	Baby Don't Cry	2000-04-01	6	94
2000	2 Pac	4:22	Baby Don't Cry	2000-04-08	7	99
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-02	1	91
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-09	2	87
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-16	3	92
2000	3 Doors Down	3:53	Kryptonite	2000-04-08	1	81
2000	3 Doors Down	3:53	Kryptonite	2000-04-15	2	70
2000	3 Doors Down	3:53	Kryptonite	2000-04-22	3	68
2000	3 Doors Down	3:53	Kryptonite	2000-04-29	4	67
2000	3 Doors Down	3:53	Kryptonite	2000-05-06	5	66

Tidy

id	artist	track	time
1	2 Pac	Baby Don't Cry	4:22
2	2Ge+her	The Hardest Part Of ...	3:15
3	3 Doors Down	Kryptonite	3:53
4	3 Doors Down	Loser	4:24
5	504 Boyz	Wobble Wobble	3:35
6	98^0	Give Me Just One Nig...	3:24
7	A*Teens	Dancing Queen	3:44
8	Aaliyah	I Don't Wanna	4:15
9	Aaliyah	Try Again	4:03
10	Adams, Yolanda	Open My Heart	5:30
11	Adkins, Trace	More	3:05
12	Aguilera, Christina	Come On Over Baby	3:38
13	Aguilera, Christina	I Turn To You	4:00
14	Aguilera, Christina	What A Girl Wants	3:18
15	Alice Deejay	Better Off Alone	6:50

id	date	rank
1	2000-02-26	87
1	2000-03-04	82
1	2000-03-11	72
1	2000-03-18	77
1	2000-03-25	87
1	2000-04-01	94
1	2000-04-08	99
2	2000-09-02	91
2	2000-09-09	87
2	2000-09-16	92
3	2000-04-08	81
3	2000-04-15	70
3	2000-04-22	68
3	2000-04-29	67
3	2000-05-06	66

5.3.4 A single observational unit is stored in multiple tables

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
```

5.3.5 Recap - same dataset, different representations

```
table1
```

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

```
head(table2)
```

```
##        country year       type      count
## 1: Afghanistan 1999      cases        745
## 2: Afghanistan 1999 population   19987071
## 3: Afghanistan 2000      cases       2666
## 4: Afghanistan 2000 population   20595360
## 5:      Brazil 1999      cases      37737
## 6:      Brazil 1999 population  172006362
```

```
table3
```

```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
table4a
```

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

```
table4b
```

```
##        country       1999       2000
## 1: Afghanistan   19987071   20595360
## 2:      Brazil  172006362  174504898
## 3:       China 1272915272 1280428583
```

5.4 Importance of data.tables

- Easier manipulation using data.table commands
  - sub-setting by rows
  - sub-setting by columns
  - `by` operations
- Many other tools work better with tidy data - consistent way of storing data
  - example: ggplot2
- Vectorized operations become easier to use

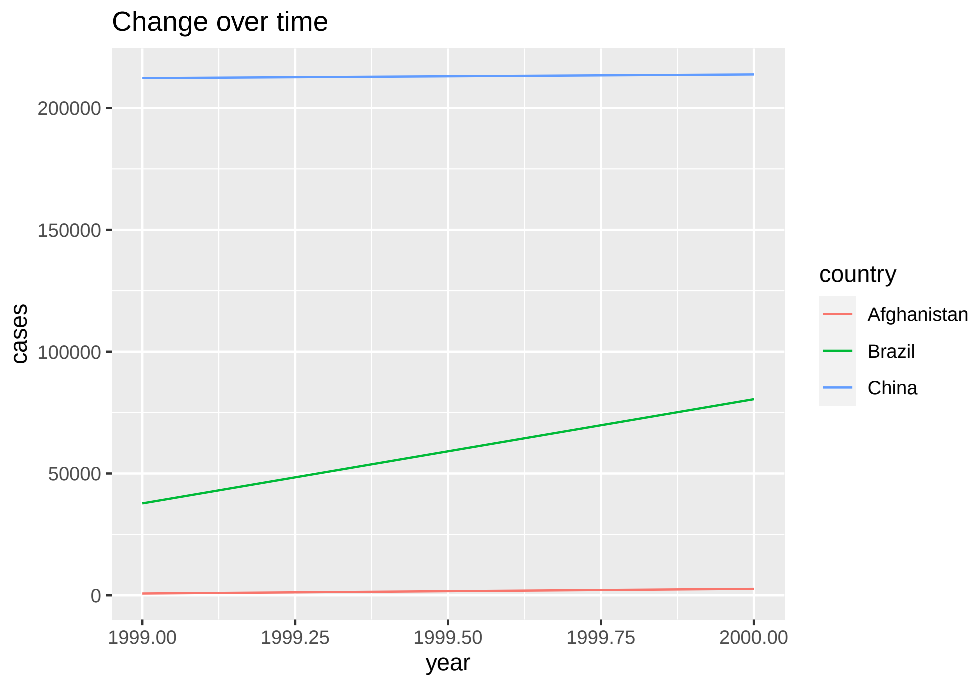
Tidy data can be easily manipulated

```
dt <- table1

# Compute rate per 10,000
dt[, `:=`(rate, cases/population * 10000)]  # vectorized operations; dt is modified

# Compute cases per year
dt[, .(cases = sum(cases)), by = year]  # note that this does not modify dt
```

```
##    year  cases
## 1: 1999 250740
## 2: 2000 296920
```

Tidy data works better with many packages

```
ggplot(dt, aes(year, cases, color = country)) + ggtitle("Change over time") + geom_line()
```



To sum up:

- In a tidy dataset, each variable must have its own column
- Each row corresponds to one unique observation
- Each cell contains a single value
- Tidy datasets are easier to work with
- Data.table library has functions to transform untidy datasets to tidy

Words to live by

Happy families are all alike; every unhappy family is unhappy in its own way.

- Leo Tolstoy

Tidy datasets are all alike, but every messy dataset is messy in its own way.

- Hadley Wickham

5.5 Lists to data.table

Very frequent scenarios in R

- read in a bunch of similar `.csv` files (`date1.csv`, `date2.csv`, `...`)
- nested data types (json/xml from API)
- `lapply` output (say doing a cross-validation, using different parameters, ...)

If possible, have one data.table instead of a (nested) list of data.tables

JSON -> R list

```
# for demonstrational purposes I've put 'simplifyDataFran
sw_list <- jsonlite::fromJSON(sw_json, simplifyDataFrame
str(sw_list)
```

```
## List of 2
##  $ :List of 5
##   ..$ Name     : chr "Anakin"
##   ..$ Gender   : chr "male"
##   ..$ Homeworld: chr "Tatooine"
##   ..$ Born     : chr "41.9BBY"
##   ..$ Jedi     : chr "yes"
##  $ :List of 5
##   ..$ Name     : chr "R2-D2"
##   ..$ Gender   : chr "unknown"
##   ..$ Homeworld: chr "Naboo"
##   ..$ Born     : chr "33BBY"
##   ..$ Jedi     : chr "no"
```

JSON example

```
sw_json <- "[
  {
    \"Name\": \"Anakin\",
    \"Gender\": \"male\",
    \"Homeworld\": \"Tatooine\",
    \"Born\": \"41.9BBY\",
    \"Jedi\": \"yes\"
  },
  {
    \"Name\": \"R2-D2\",
    \"Gender\": \"unknown\",
    \"Homeworld\": \"Naboo\",
    \"Born\": \"33BBY\",
    \"Jedi\": \"no\"
  }
]"
```

To convert a list to a data.table, we use `as.data.table` to convert a list to a data.table with one row and `rbindlist` to stack data.tables

If something is not working as expected try with: * `rbindlist(fill = TRUE, use.names = TRUE)` * prefer data.table's `rbindlist` over `rbind`

```
sw_dt <- lapply(sw_list, as.data.table) %>% rbindlist
sw_dt
```

```
##      Name  Gender Homeworld    Born Jedi
## 1: Anakin    male  Tatooine 41.9BBY  yes
## 2:  R2-D2 unknown     Naboo   33BBY   no
```

A very common situation: **list name is a variable**

```
dt_list %>% lapply(head, n = 3)
```

```
## $young
##    x         y
## 1: 1 0.9304811
## 2: 2 2.2392414
## 3: 3 3.2504199
##
## $old
##    x        y
## 1: 1 1.780951
## 2: 2 2.596584
## 3: 3 6.820628
```

How would you reshape it?

```
dt <- rbindlist(dt_list, idcol = "age")  # call the new variable 'age'

dt %>% print(3)
```

```
##       age  x          y
##  1: young  1  0.9304811
##  2: young  2  2.2392414
##  3: young  3  3.2504199
## ---
## 18:   old  8 14.4904427
## 19:   old  9 16.8440569
## 20:   old 10 19.6451449
```

5.6 Melting and Casting (wide data <-> long data)

Typical problem:

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

Untidy datasets:

```r
## Wide data
print(table4a)
```

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

```r
## Long data
print(table2)
```

```
##         country year       type      count
##  1: Afghanistan 1999      cases        745
##  2: Afghanistan 1999 population   19987071
##  3: Afghanistan 2000      cases       2666
##  4: Afghanistan 2000 population   20595360
##  5:      Brazil 1999      cases      37737
##  6:      Brazil 1999 population  172006362
##  7:      Brazil 2000      cases      80488
##  8:      Brazil 2000 population  174504898
##  9:       China 1999      cases     212258
## 10:       China 1999 population 1272915272
## 11:       China 2000      cases     213766
## 12:       China 2000 population 1280428583
```

Solution in R:

Transform wide data -> long data

- `data.table::melt()`
- `tidyr::gather()`

Transform long data -> wide data

- `data.table::dcast()`
- `tidyr::spread()`

To make wide data into long data we use `data.table::melt`

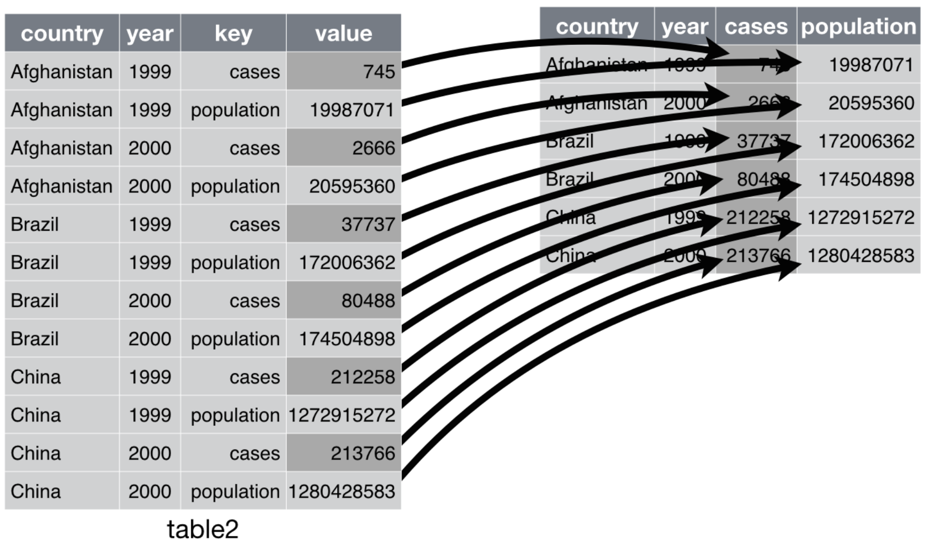
```r
print(table4a)
```

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

```r
table4a %>% data.table::melt(id.vars = "country",
                             measure.vars = c("1999", "2000"),
                             # would work also without specifying *either* measure.vars OR id.vars
                             variable.name = "year",
                             value.name = "cases")
```

```
##        country year  cases
## 1: Afghanistan 1999    745
## 2:      Brazil 1999  37737
## 3:       China 1999 212258
## 4: Afghanistan 2000   2666
## 5:      Brazil 2000  80488
## 6:       China 2000 213766
```

```
## Alternatively: table4a %>% tidyr::gather(`1999`, `2000`, key = "year", value = "cases")
```



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

table2

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Figure 5.2: Melted country dataset

To make long data into wide data we use `data.table::dcast`

```
## Help
dcast(data, formula, fun.aggregate = NULL, sep = "_", ..., margins = NULL, subset = NULL,
    fill = NULL, drop = TRUE, value.var = guess(data), verbose = getOption("datatable.verbose"))
```

```
data.table::dcast(table2, ... ~ type, value.v print(table2)
```

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

```
##         country year       type      count
##  1: Afghanistan 1999      cases        745
##  2: Afghanistan 1999 population   19987071
##  3: Afghanistan 2000      cases       2666
##  4: Afghanistan 2000 population   20595360
##  5:      Brazil 1999      cases      37737
##  6:      Brazil 1999 population  172006362
##  7:      Brazil 2000      cases      80488
##  8:      Brazil 2000 population  174504898
##  9:       China 1999      cases     212258
## 10:       China 1999 population 1272915272
## 11:       China 2000      cases     213766
## 12:       China 2000 population 1280428583
```

Compare

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

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

```
print(table1)
```

```
##        country year  cases population     rate
## 1: Afghanistan 1999    745   19987071 0.372741
## 2: Afghanistan 2000   2666   20595360 1.294466
## 3:      Brazil 1999  37737  172006362 2.193930
## 4:      Brazil 2000  80488  174504898 4.612363
## 5:       China 1999 212258 1272915272 1.667495
## 6:       China 2000 213766 1280428583 1.669488
```

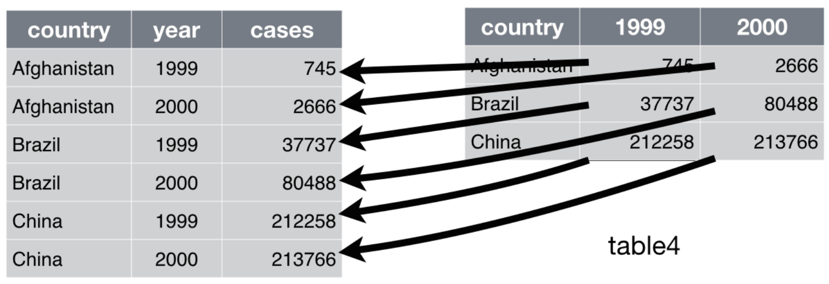


Figure 5.3: Dcasted country dataset

5.7 Separating and Uniting (1 <-> more variables)

Typical problem:

1. One column contains multiple variables
2. Multiple columns contain one variable

Untidy datasets

```
## One column contains multiple variables
print(table3)
```

```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
## Multiple columns contain one variable
print(table5)
```

```
##        country century year              rate
## 1: Afghanistan      19   99      745/19987071
## 2: Afghanistan      20   00     2666/20595360
## 3:      Brazil      19   99   37737/172006362
## 4:      Brazil      20   00   80488/174504898
## 5:       China      19   99 212258/1272915272
## 6:       China      20   00 213766/1280428583
```

Solution in R:

1) variable -> multiple variables
   - `tidyr::separate()`
2) multiple variables -> 1 variables
   - `tidyr::unite()`

other useful functions:

- `data.table::tstrsplit`, `strsplit`, `paste`, `substr`

To separate 1 variable to multiple variables we use `tidyr::separate()`.

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE,
    extra = "warn", fill = "warn", ...)
```

```
table3
```

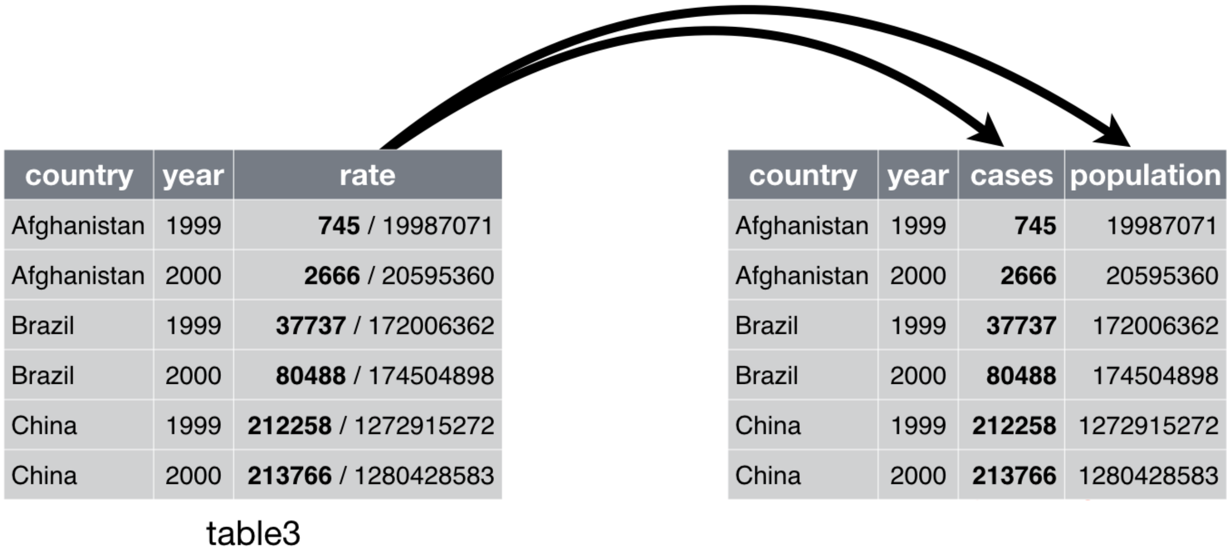
```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
separate(table3, col = rate, into = c("cases", "popul
```

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

```
separate(table3, col = rate, into = c("cases", "popul
```

```
## [1] "data.table" "data.frame"
```



country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 5.4: Seperated country dataset

To unite multiple variables to 1 variable we use `tidyr::unite()`.

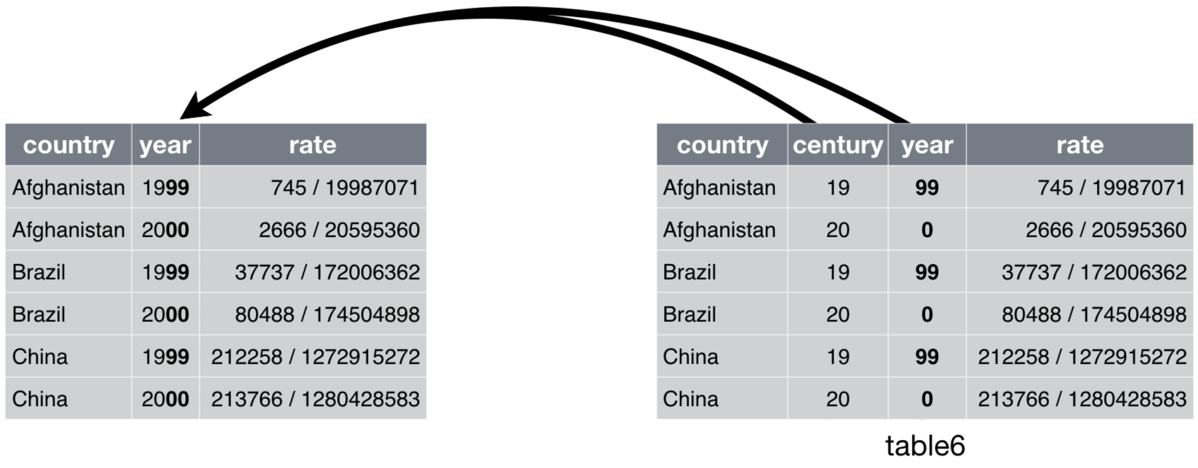
```
unite(data, col, ..., sep = "_", remove = TRUE)
```

```
table5
```

```
##        country century year              rate
## 1: Afghanistan      19   99      745/19987071
## 2: Afghanistan      20   00     2666/20595360
## 3:      Brazil      19   99   37737/172006362
## 4:      Brazil      20   00   80488/174504898
## 5:       China      19   99 212258/1272915272
## 6:       China      20   00 213766/1280428583
```

```
unite(table5, col = new, century, year, sep = "")
```

```
##        country  new              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```



country	year	rate
Afghanistan	19**99**	745 / 19987071
Afghanistan	20**00**	2666 / 20595360
Brazil	19**99**	37737 / 172006362
Brazil	20**00**	80488 / 174504898
China	19**99**	212258 / 1272915272
China	20**00**	213766 / 1280428583

country	century	year	rate
Afghanistan	19	**99**	745 / 19987071
Afghanistan	20	**0**	2666 / 20595360
Brazil	19	**99**	37737 / 172006362
Brazil	20	**0**	80488 / 174504898
China	19	**99**	212258 / 1272915272
China	20	**0**	213766 / 1280428583

Figure 5.5: United country dataset

5.8 Non-tidy data

- Performance advantage using certain functions
  - `colSums()` or `heatmap()` on matrices
- Field convention
- Memory efficiency
  - don't worry, you should be fine with tidy-data in `data.table`

Interesting blog post:

- http://simplystatistics.org/2016/02/17/non-tidy-data/

Chapter 6

Advanced plotting

Define a theme

```
mysize <- 15
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()
```

In the following examples we will use the `ind` dataset

```
ind
```

```
##       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
##   4:   4      Angola    AGO  19 0.532               SSA
##   5:   5   Argentina    ARG  34 0.836          Americas
##  ---
## 147: 147     Uruguay    URY  73 0.793          Americas
## 148: 148  Uzbekistan    UZB  18 0.675 East EU Cemt Asia
## 149: 149       Yemen    YEM  19 0.498              MENA
## 150: 150      Zambia    ZMB  38 0.586               SSA
## 151: 151    Zimbabwe    ZWE  21 0.509               SSA
```

HDI: Human Development Index (http://hdr.undp.org/)

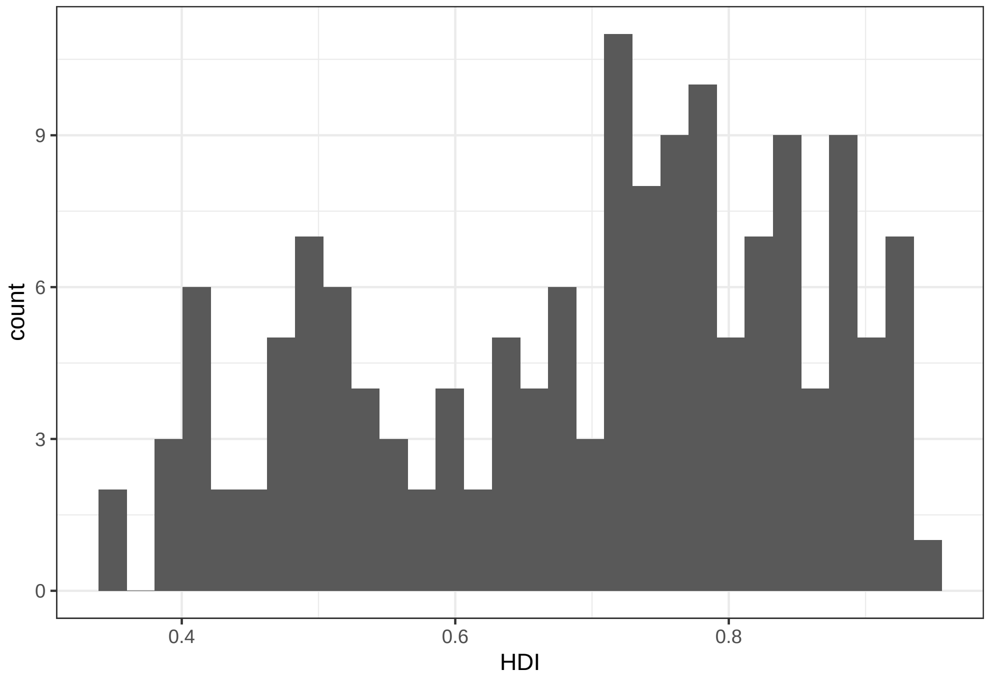
Year: 2014

6.1 Histogram

A histogram represents the frequencies of values of a variable bucketed into ranges. It takes as input numeric variables only. Histogram is similar to bar chart but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range. Each bar of the histograms is called bin.

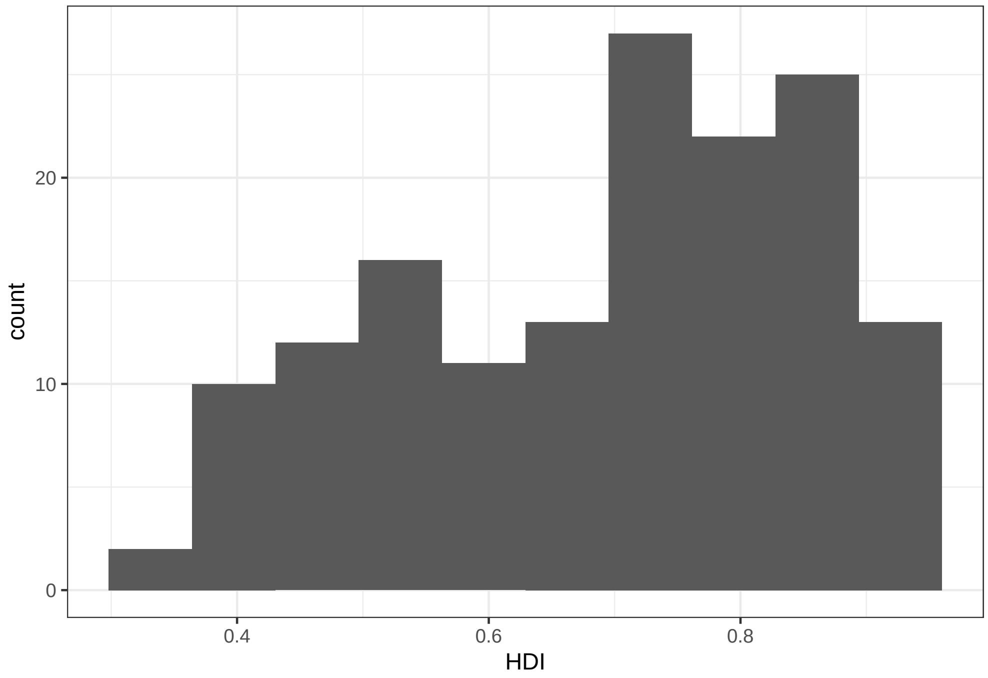
Histogram of Human Development Index (HDI) in the ind dataset:

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



Setting the number of bins

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



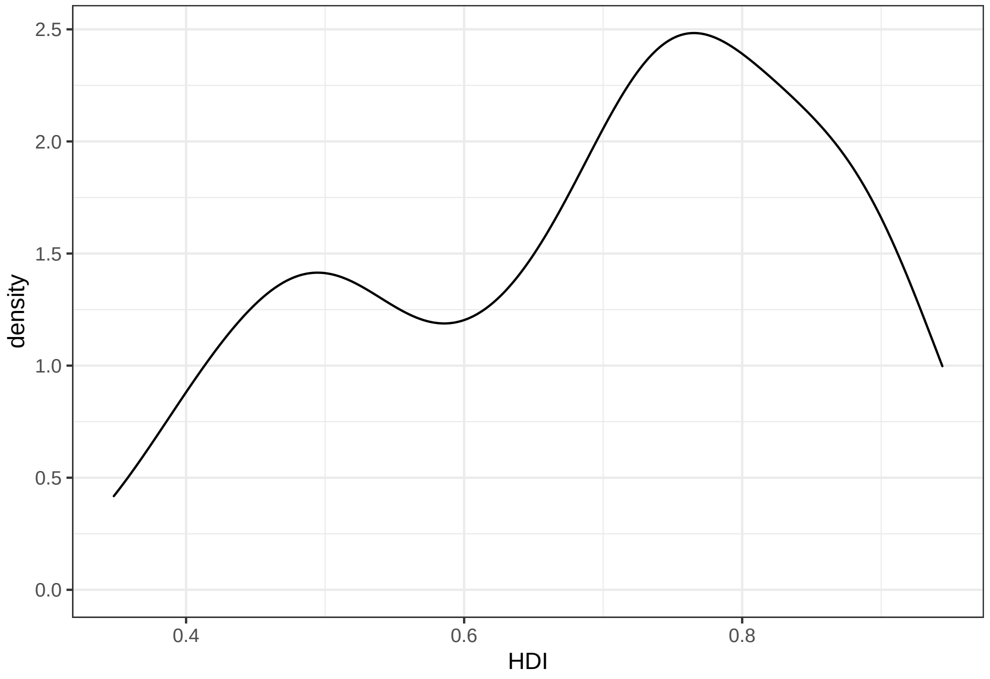
6.2 Density plots

Sometimes it is better to represent the distribution of a numeric variable using density plots and not histograms.

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

Density plot of Human Development Index (HDI) in the ind dataset:

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

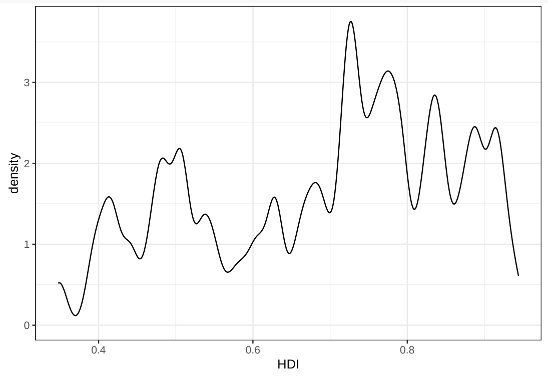


6.3 Kernel density plots - smoothing bandwidth

Using the bandwidth argument of the `geom_density` function, one can tweak how close the density should the distribution be. It can be set manually. Default option is a bandwidth rule (which is usually a good choice).

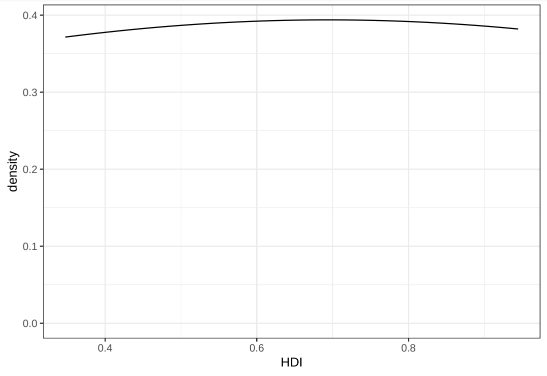
Small bandwidth:

```r
ggplot(ind, aes(HDI)) + geom_density(bw = 0.01) +
```



Large bandwidth:

```r
ggplot(ind, aes(HDI)) + geom_density(bw = 1) + mytheme
```



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

6.4 Multidimensional data

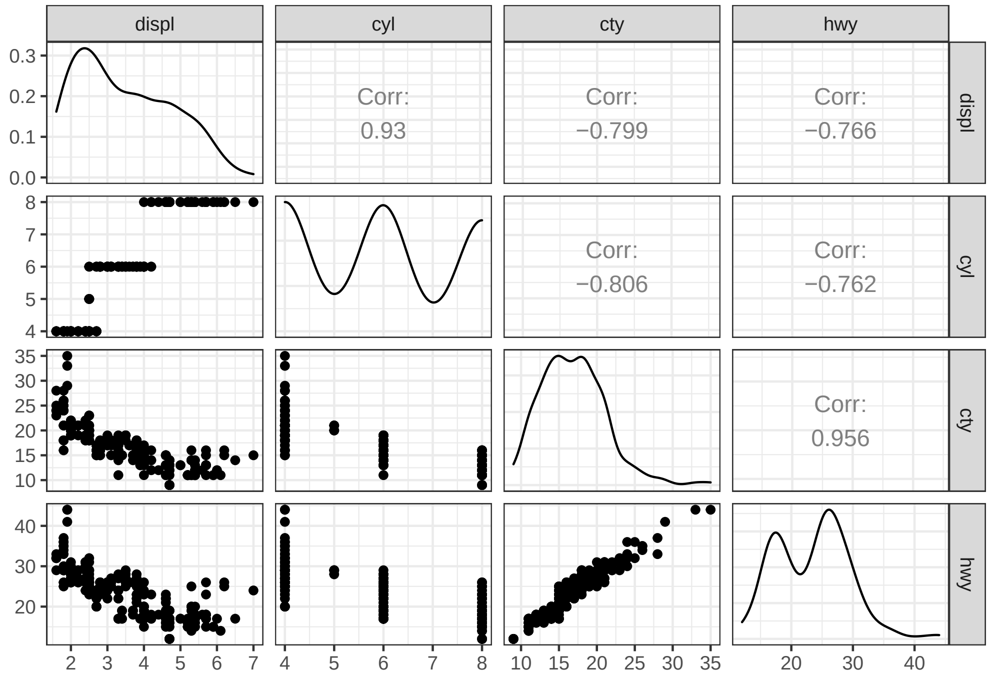
- For this part, we assume all variables to be quantitative.
- That means, all variable values will be plotted on continuous (xy-)axes.

- More complicated, mixed binary/multinomial/ordinal/continuous data might require specialized plotting and data analysis strategies.

6.5 Small dimensionality, ~3-10 variables

Plot matrix

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

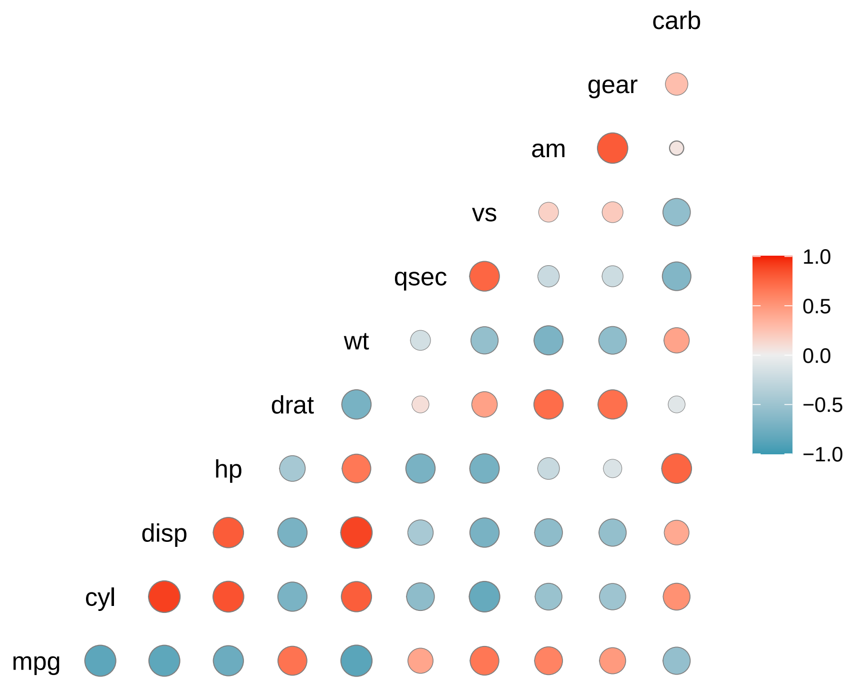


6.6 Medium dimensionality, <100 variables

Correlation plot is a graph of correlation matrix. Useful to highlight the most correlated variables in a data table. In this plot, correlation coefficients are colored according to the value. Correlation matrix can be also reordered according to the degree of association between variables.

Correlation plots are also called "corrgrams" or "correlograms"

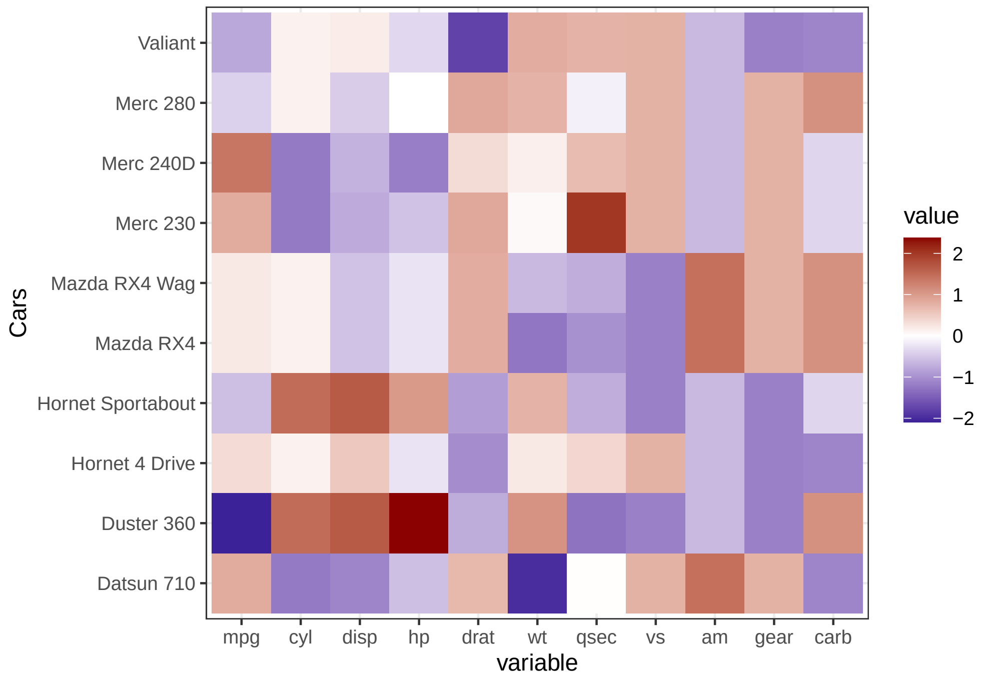
```r
ggcorr(mtcars, geom = "circle")
```



6.7 High dimensionality, >100, >>100 variables

For the representation of high dimensionality datasets with more that 100 variables we use heatmaps. A heatmap is a graphical representation of data that uses a system of color-coding to represent different values.

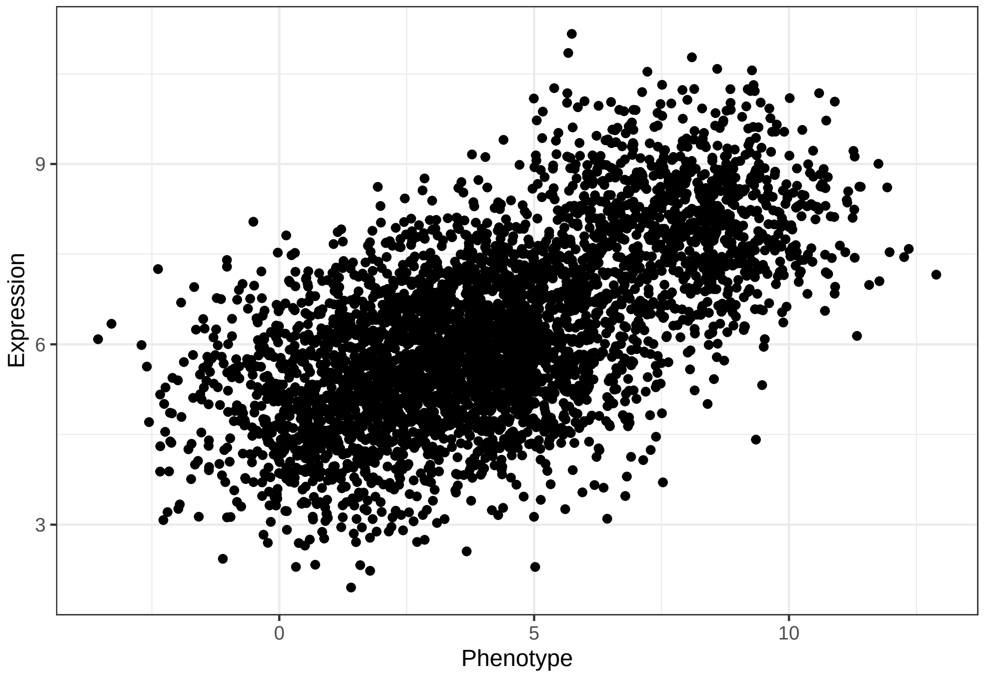
```
mtcars_melted <- as.data.table(scale(mtcars[1:10, ]), keep.rownames = "Cars") %>%
    melt(id.var = "Cars")
mtcars_melted <- mtcars_melted
ggplot(mtcars_melted, aes(variable, Cars)) + geom_tile(aes(fill = value)) + scale_fill_gradient2(low = "dar
    mid = "white", high = "darkred") + mytheme
```



6.8 Conditional correlation

In this data, the two variables Phenotype and Expression are nicely correlated:

```
ggplot(dt, aes(Phenotype, Expression)) + geom_point() + mytheme
```



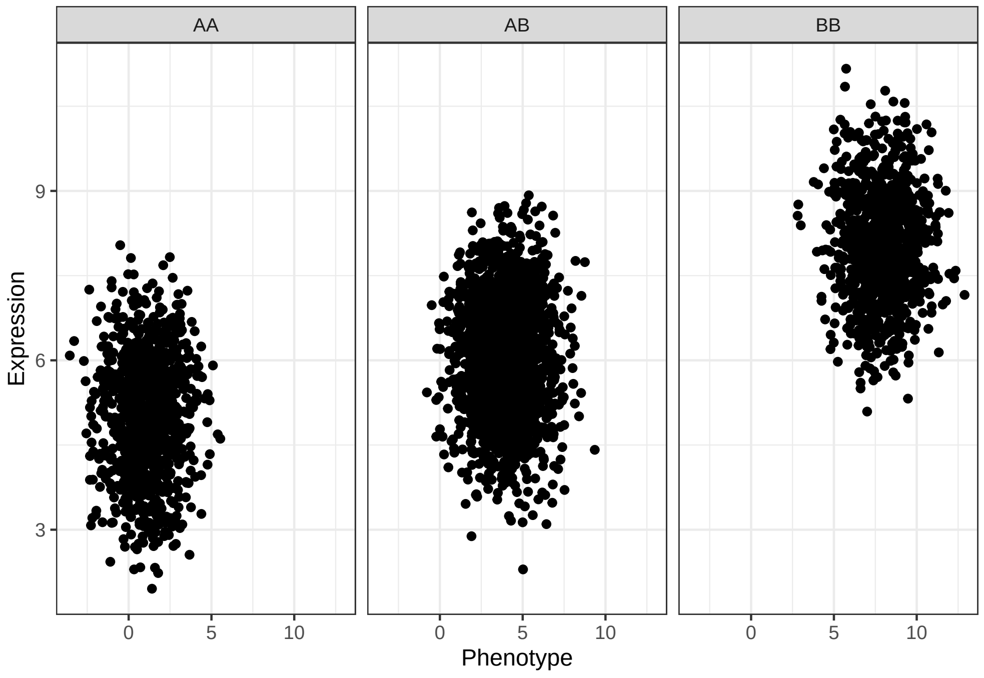
```
dt[, cor(Phenotype, Expression)]
```

```
## [1] 0.6303033
```

. . . but are they directly correlated or is that due to a further variable?

6.9 Facets are good to check conditional correlation

```
ggplot(dt, aes(Phenotype, Expression)) + geom_point() + facet_wrap(~Genotype) + mytheme
```



Statistically speaking: The two variables *are* correlated, but their conditional correlation (given the genotype) is zero.

6.10 The `diamonds` data

```
## # A tibble: 1 x 10
##   carat cut   color clarity depth table price     x     y     z
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal E     SI2      61.5    55   326  3.95  3.98  2.43
```

Description of R datasets are available through the help.

```
`?`(diamonds)
```

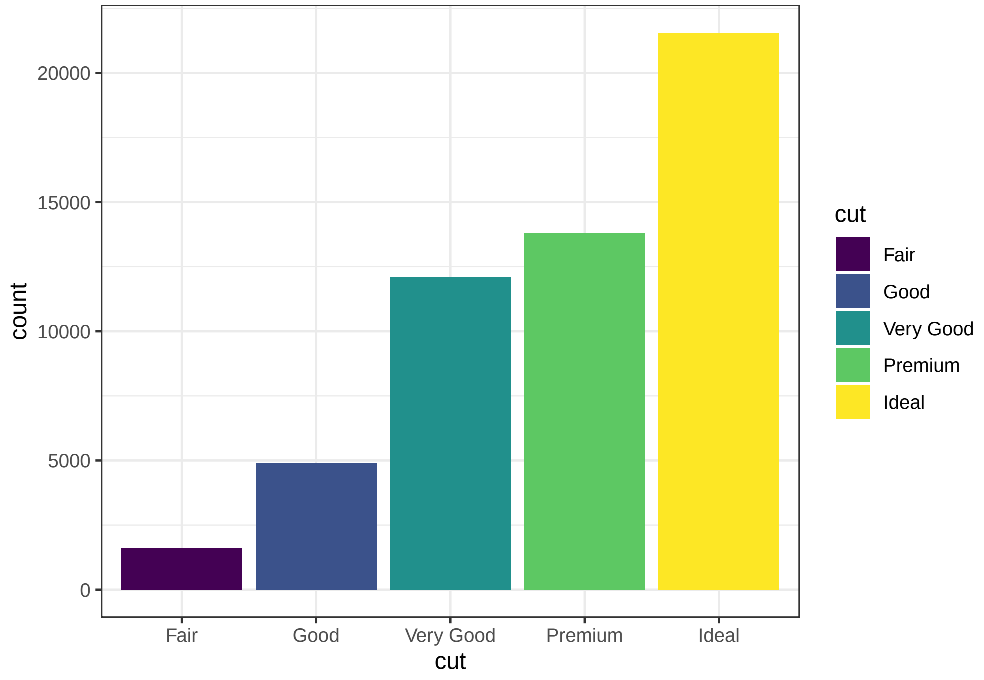
Variable	Description
price	price in US dollars ($326, $18,823)
carat	weight of the diamond (0.2, 5.01)
cut	quality of the cut (Fair, Good, Very Good, Premium, Ideal)
color	diamond colour, from J (worst) to D (best)
clarity	a measurement of how clear the diamond is (I1 (worst), SI1, SI2, VS1, VS2, VVS1, VVS2, IF (best))
x	length in mm (0.00, 10.74)
y	width in mm (0.00, 58.9)
z	depth in mm (0.00, 1.8)
depth	total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43, 79)
table	width of top of diamond relative to widest point (43, 95)

6.11 Conditioning with categorical variables: multi-way barplots

In the following examples we are going to use barplots. A barplot (or barchart) is one of the most common types of graphic. It shows the relationship between a numeric and a categoric variable. Each entity of the categoric variable is represented as a bar. The size of the bar represents its numeric value.

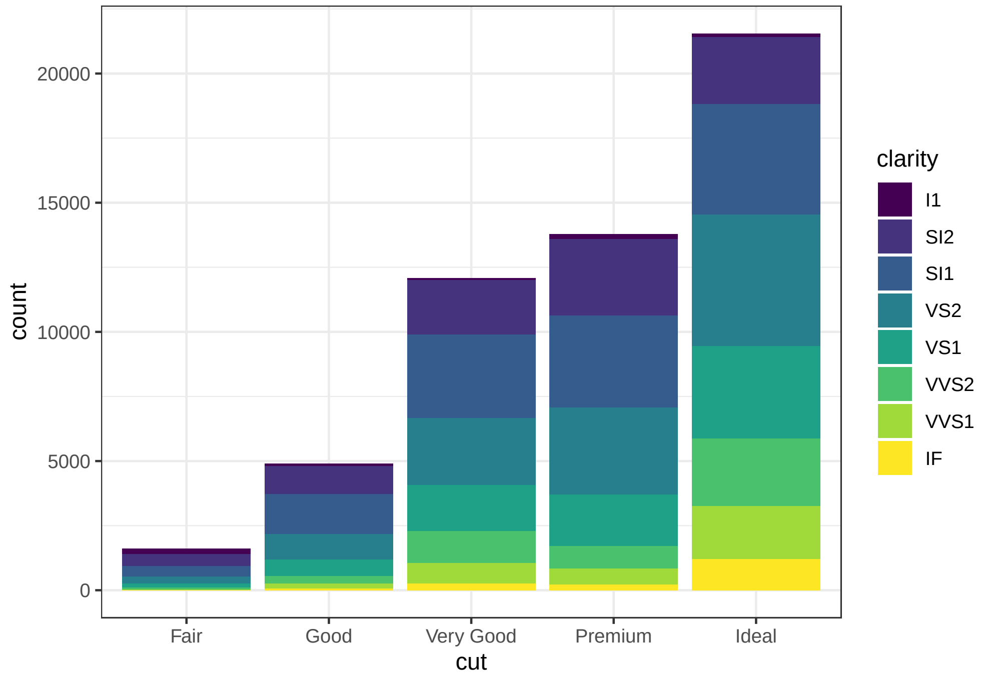
Additionally, we introduced that, one can map variables to colours easily with ggplot.

```r
ggplot(data = diamonds, aes(cut, fill = cut)) + geom_bar() + mytheme
```



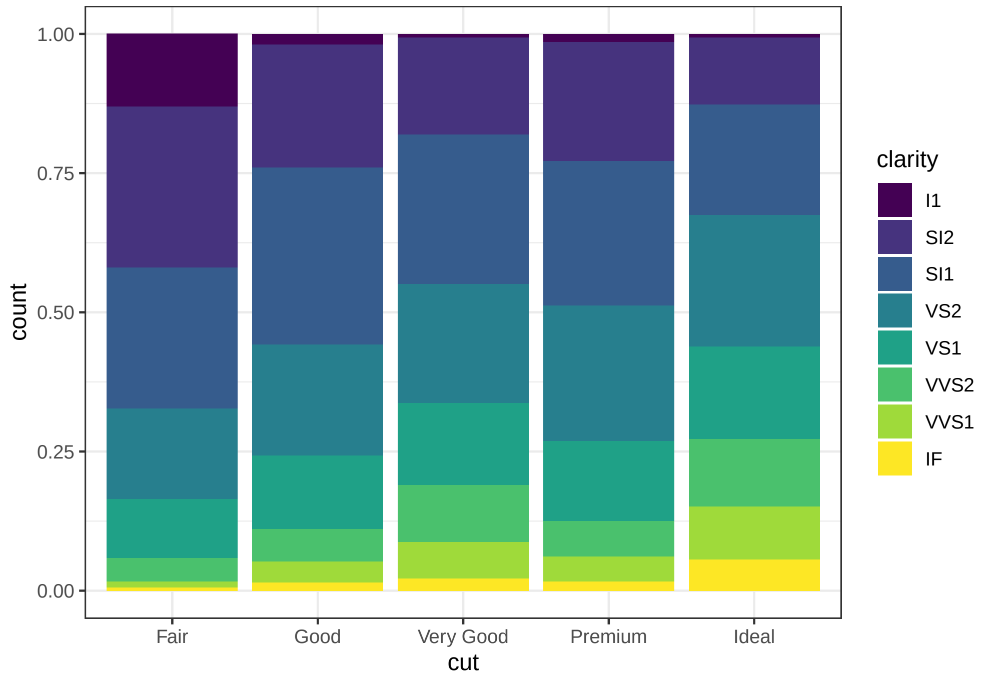
What happens if you map fill colour to other variable like `clarity`?

```r
ggplot(diamonds, aes(x = cut, fill = clarity)) + geom_bar() + mytheme
```



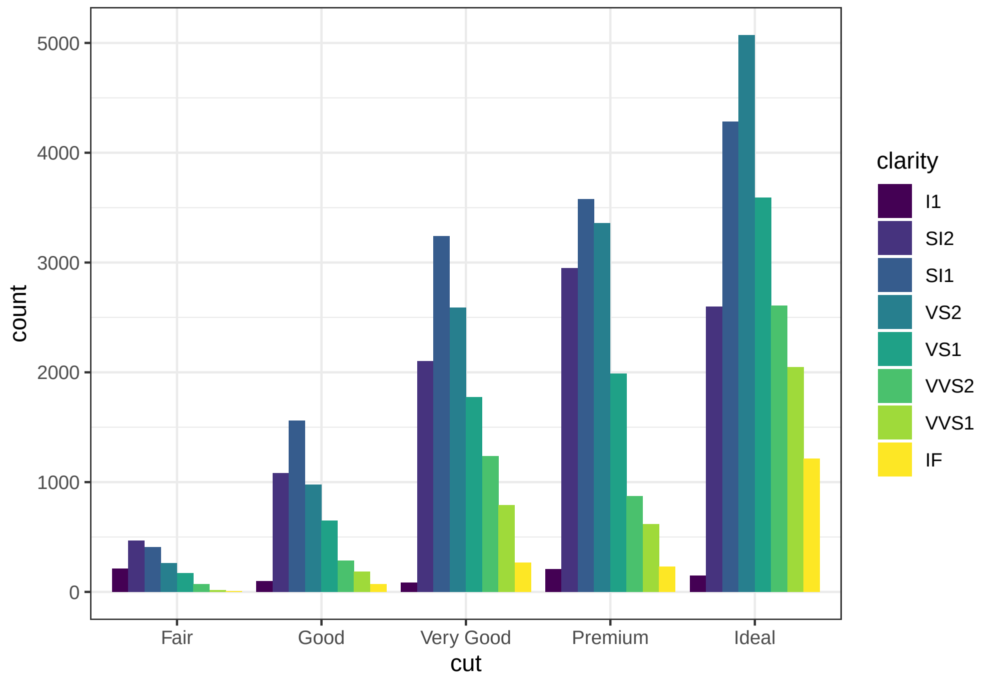
The stacking is performed automatically by the position adjustment specified by the position argument. The default position is “stack”.

```
ggplot(diamonds, aes(x = cut, fill = clarity)) + geom_bar(position = "fill") + mytheme
```



`position = "fill"` works like stacking, but makes each set of stacked bars the have same height. This makes it easier to compare proportions across groups.

```
ggplot(diamonds, aes(x = cut, fill = clarity)) + geom_bar(position = "dodge") + mytheme
```



`position = "dodge"` places overlapping objects directly beside one another. This makes it easier to compare individual values.

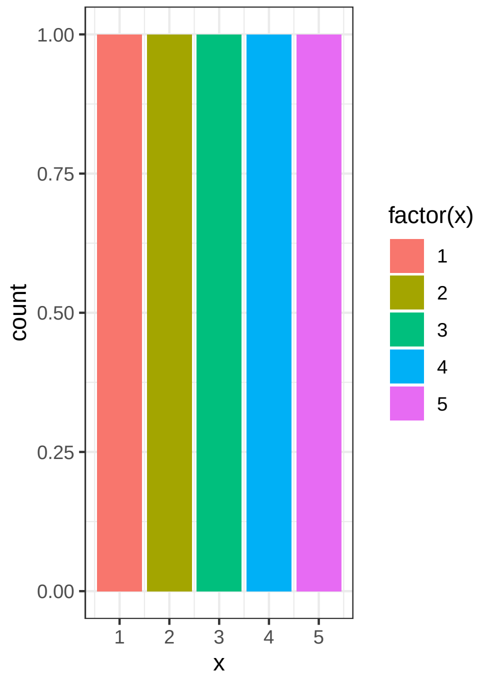
Other position types include `"jitter"` and `"identity"`.

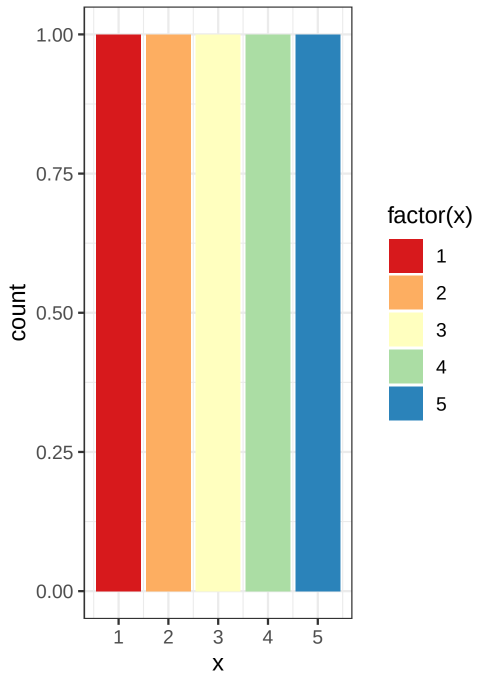
6.12 Color coding in R

Four ways to code colors in R (ggplot2):

1. Default aesthetic mapping (`aes(color = variable)`)

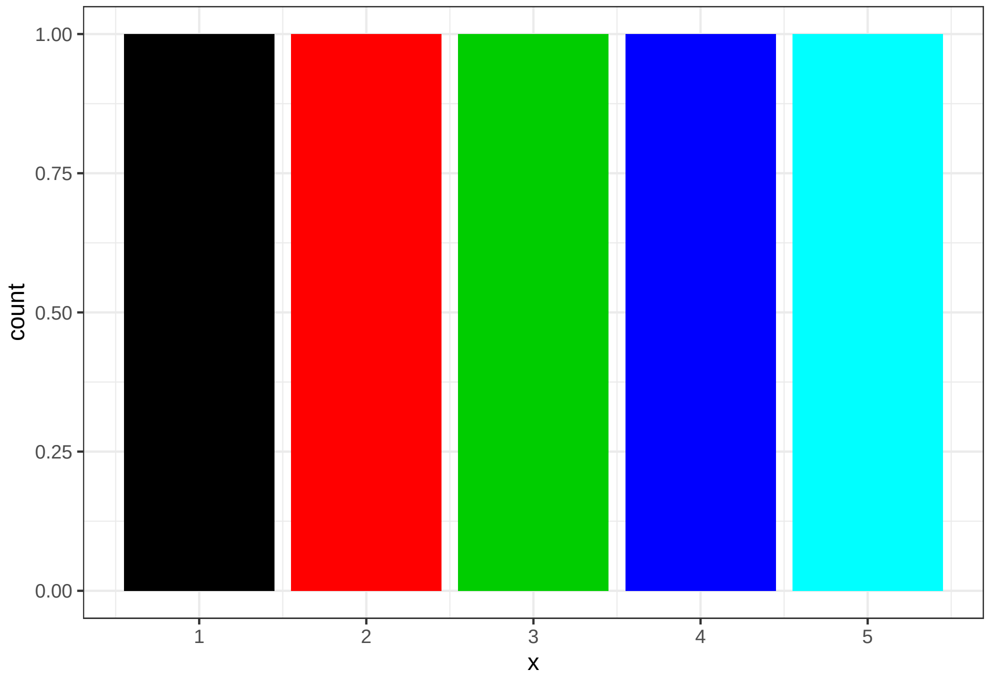
```
x <- c(1:5)
dt <- data.table(x)
p <- ggplot(dt, aes(x))  # save the mapping to p
p + geom_bar(aes(fill = factor(x)))  # factor convert x into factors, then map to ggplot default color pal
p + geom_bar(aes(fill = factor(x))) + scale_fill_brewer(palette = "Spectral")  # change default color pale
```





2. Predefined color numbers

```
x <- c(1:5)
dt <- data.table(x)
p <- ggplot(dt, aes(x))  # save the mapping to p
p + geom_bar(fill = x) + mytheme
```

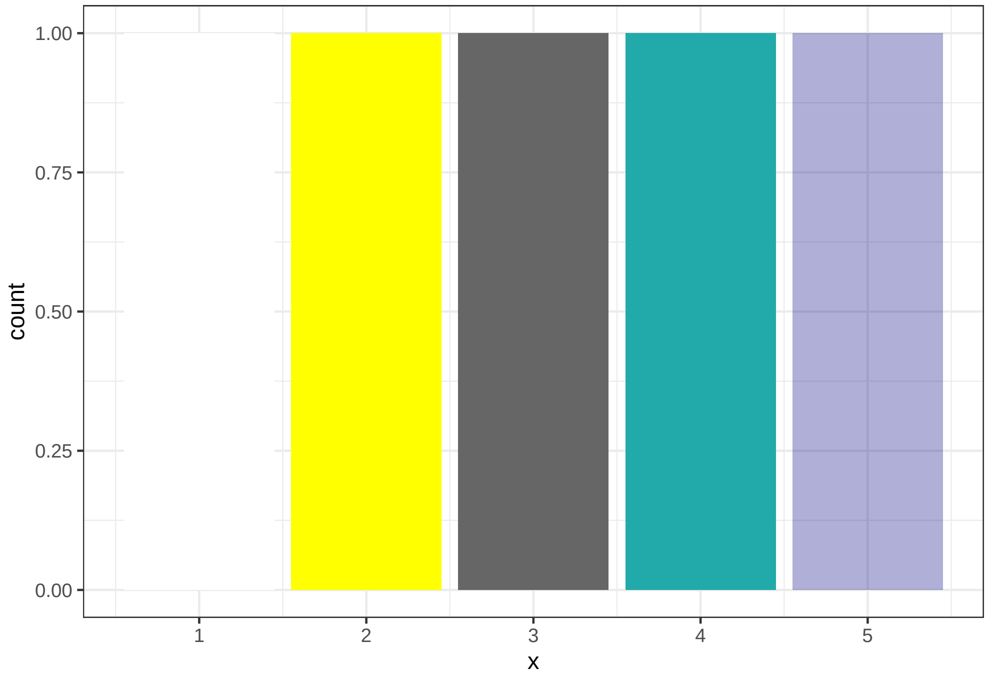


3. Predefined color names

- *Additive color model*
- Hexadecimal coding of three number pairs for red, green, blue; and alpha (00%-99%) for transparency
- `00-FF` corresponds to 0-255

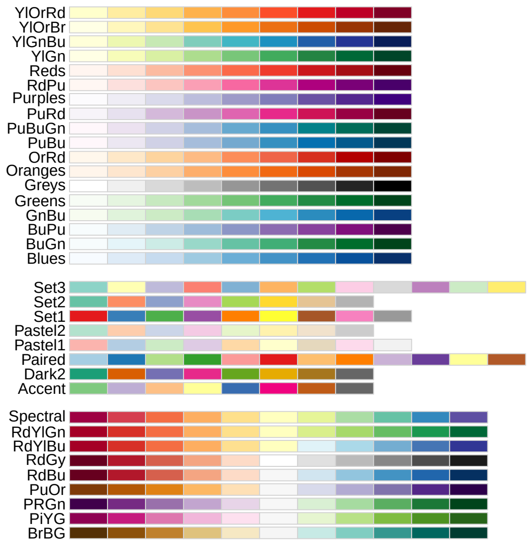
3. Predefined color names or RGB - HTML color codes (last bar with 50% transparency)

```
x <- c(1:5)
dt <- data.table(x)
p <- ggplot(dt, aes(x))  # save the mapping to p
p + geom_bar(fill = c("#FFFFFF", "#FFFF00", "#666666", "#22AAAA", "#00008050")) +
    mytheme
```



The RColorBrewer package provides a set of nice color palettes

```r
library(RColorBrewer)
display.brewer.all()
```



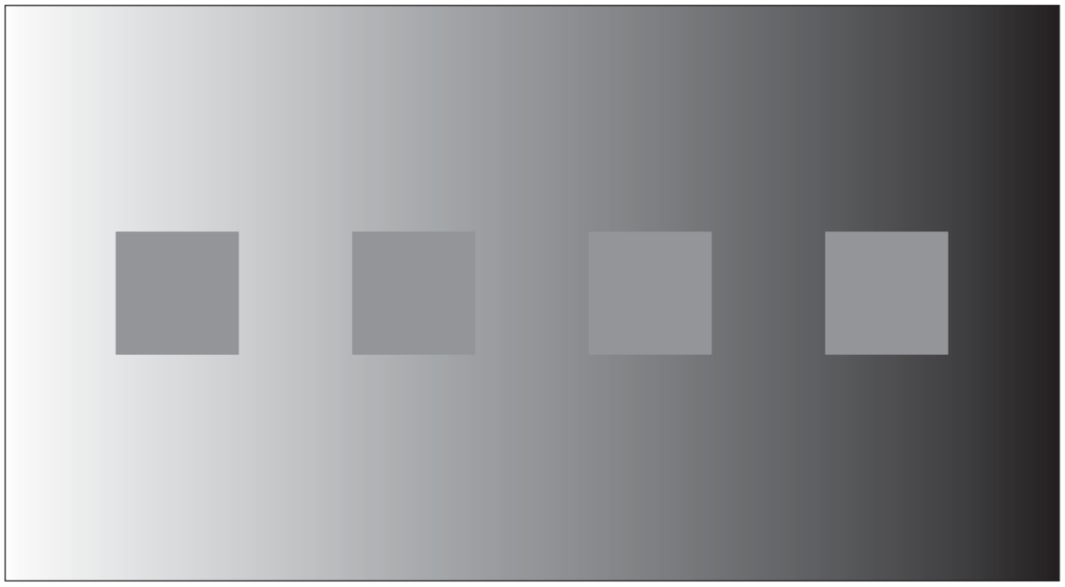
Like all aspects of visual perception, we do not perceive color in an absolute manner.

- Perception of an object is influenced by the context
- Visual perception is relative, not absolute.

You should use colors meaningfully and with restraint

Rule #1

If you want different objects of the `same color` in a table or graph to look the same, make sure that the `background is consistent`.



Rule #2

If you want objects in a table or graph to be `easily seen`, use a background color that `contrasts sufficiently` with the object.

This is an example of double-encoding (position and colors).

Rule #3

Use color only when needed to serve a particular communication goal.

Rule #4

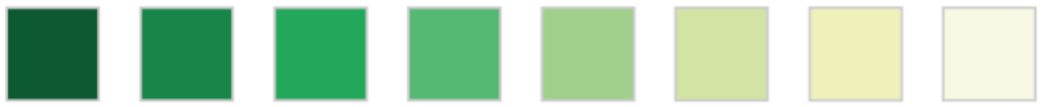
Use different colors only when they correspond to differences of meaning in the data.

Rule #5

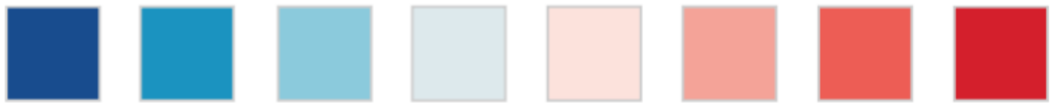
Use `soft`, natural colors to display `most` information and `bright` and/or dark colors to `highlight` information that requires greater attention.

6.13 Define standard palettes of colors for particular purposes

Different types of color palettes * Categorical: separate items into distinct groups * Sequential: quantitative differences. Typically for quantitative positive values.



- Diverging: quantitative differences, breakpoints. For quantitative signed values (e.g. correlations)



Xiong et al., Science 2014

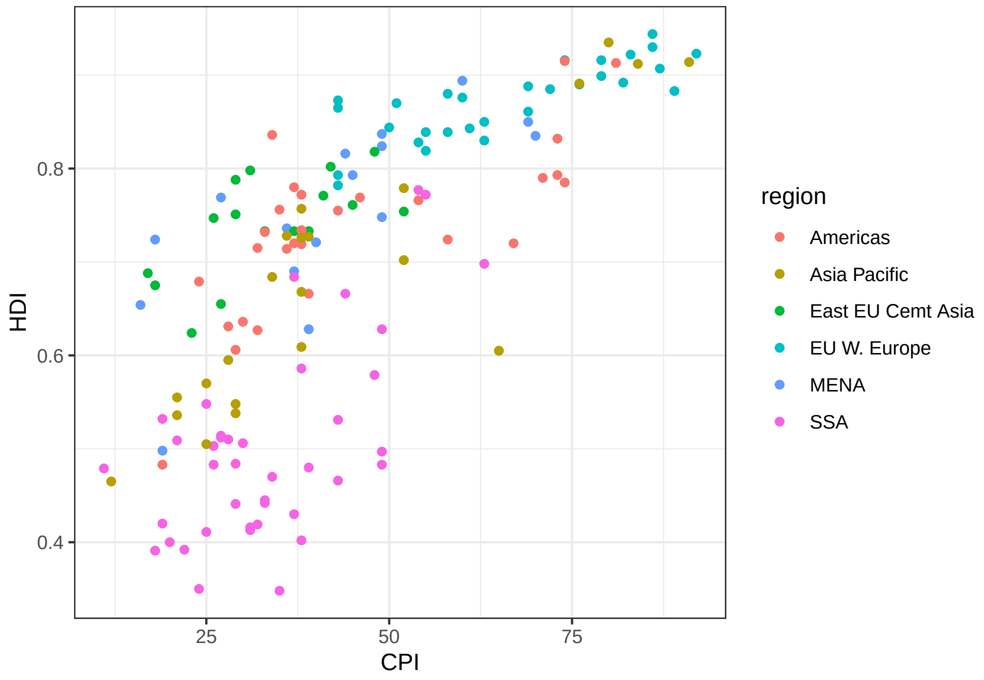
6.14 Themes control non-data parts of your plots:

- Complete theme
- Axis title
- Axis text
- Plot title
- Legends

They control the appearance of your plots: size, color, position...

But have nothing to do with how the data is represented.

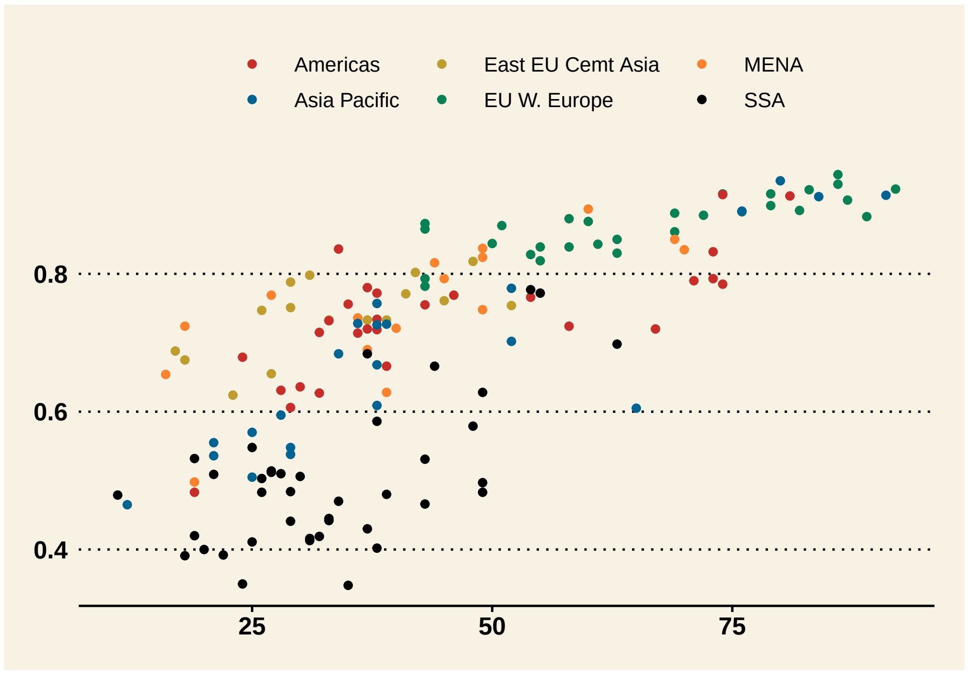
```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_point() + theme_bw()   # black and white
```



Other complete themes: `theme_classic`, `theme_minimal`, `theme_light`, `theme_dark`.

You can add more themes with `ggthemes` package (from github)

```r
library(ggthemes)
ggplot(ind, aes(CPI, HDI, color = region)) + geom_point() + theme_wsj() + scale_colour_wsj("colors6",
    ""))
```



wsj: Wall Street Journal

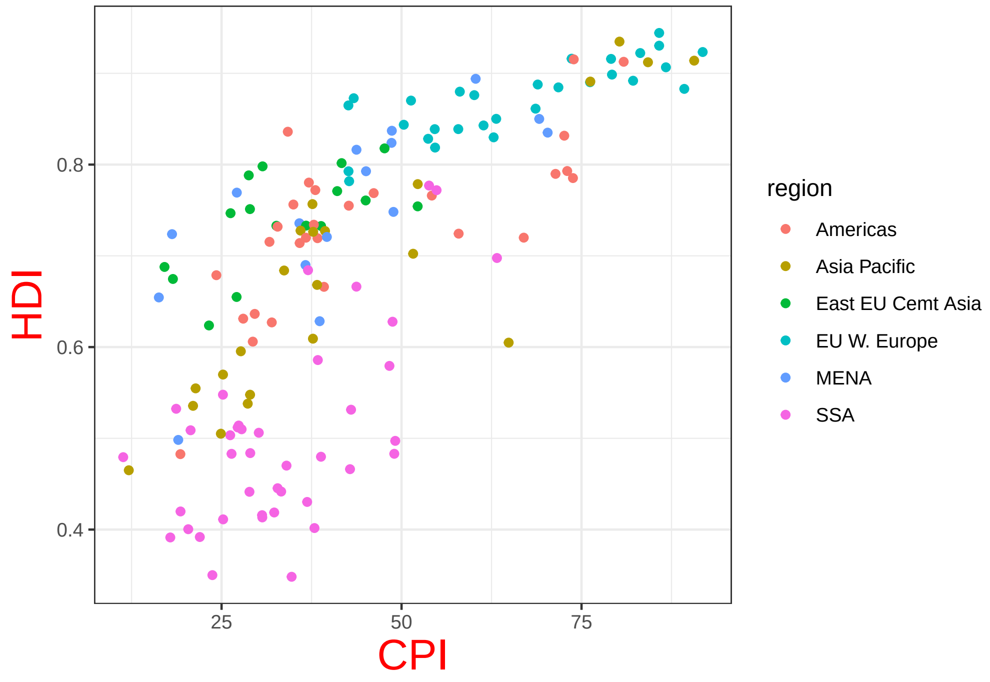
6.15 Axis

To manipulate the axis title, you need to tweak the following arguments of the `theme()`:

- `axis.title`
- `axis.title.x`
- `axis.title.y`

In case of many themes, the latter overrides the former

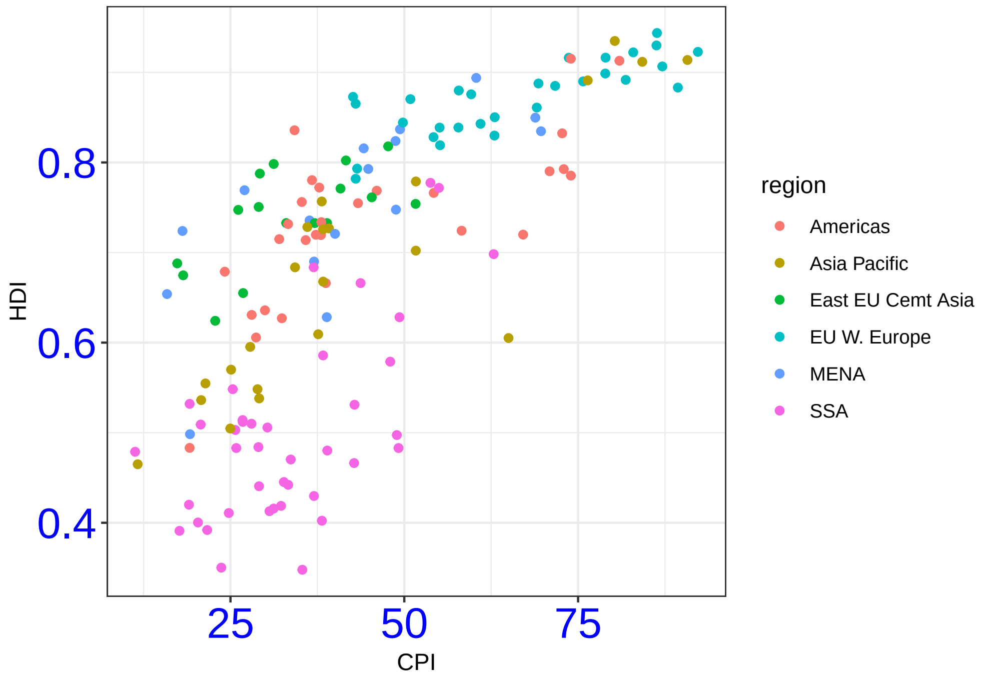
```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + theme_bw() + theme(axis.title = element_text(s
    color = "red"))
```



To manipulate the axis text, you need to tweak the following arguments of the `theme()`:

- `axis.text`
- `axis.text.x`
- `axis.text.y`

```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + theme_bw() + theme(axis.text = element_text(s
    color = "blue"))
```



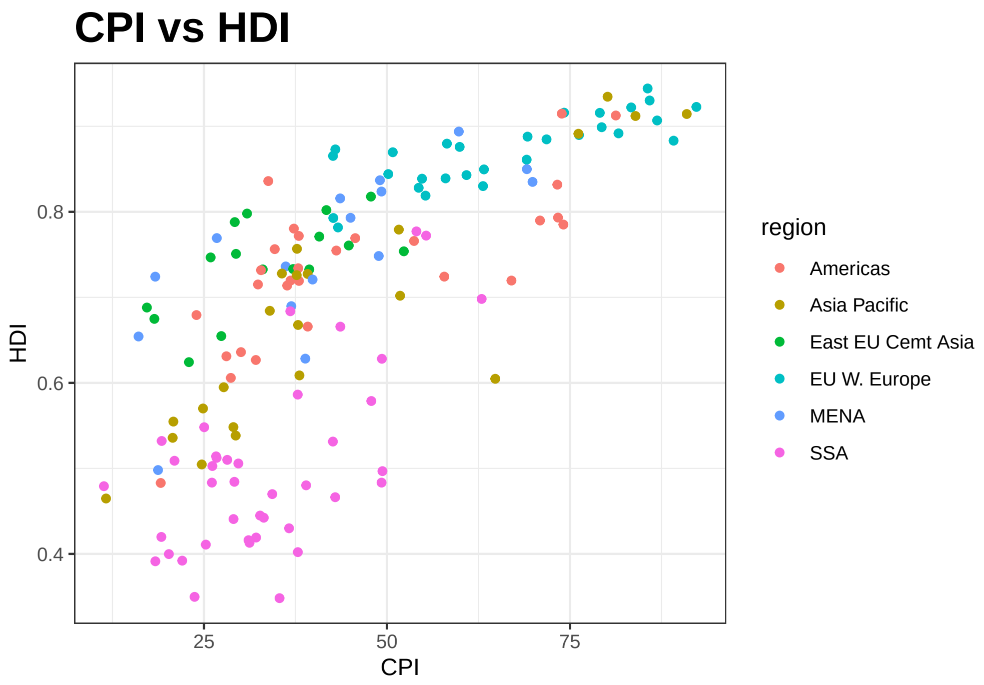
Summary axis elements

The axis elements control the appearance of the axes:

Element	Setter	Description
axis.line	`element_line()`	line parallel to axis (hidden in default themes)
axis.text	`element_text()`	tick labels
axis.text.x	`element_text()`	x-axis tick labels
axis.text.y	`element_text()`	y-axis tick labels
axis.title	`element_text()`	axis titles
axis.title.x	`element_text()`	x-axis title
axis.title.y	`element_text()`	y-axis title
axis.ticks	`element_line()`	axis tick marks
axis.ticks.length	`unit()`	length of tick marks

6.16 Plot title themes

```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + ggtitle("CPI vs HDI") +
    theme_bw() + theme(plot.title = element_text(size = 20, face = "bold"))
```

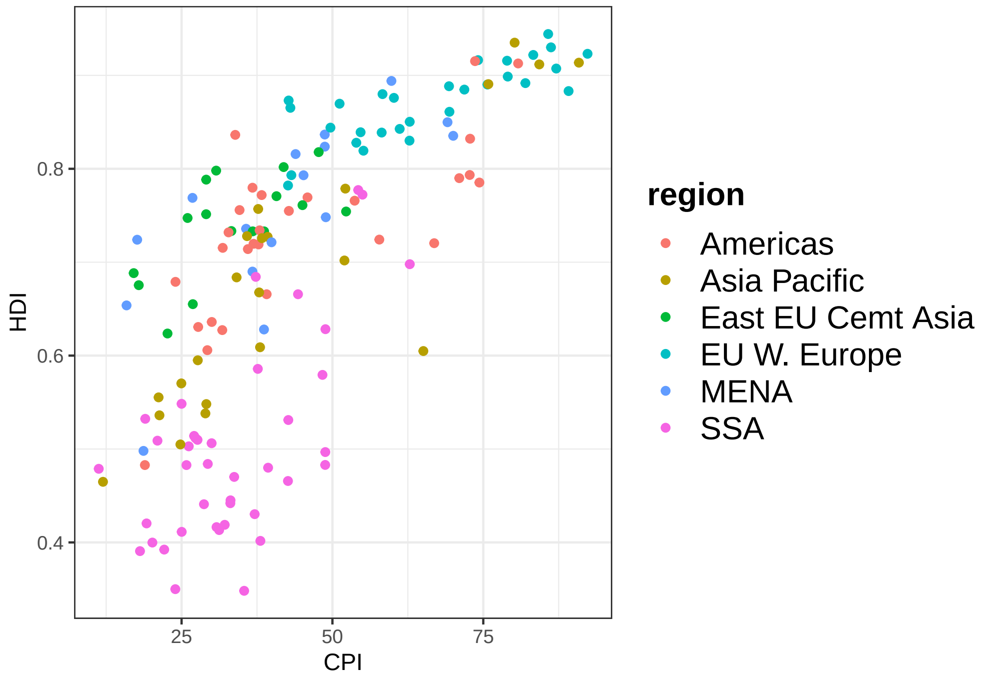


Some elements affect the plot as a whole:

Element	Setter	Description
plot.background	`element_rect()`	plot background
plot.title	`element_text()`	plot title
plot.margin	`margin()`	margins around plot

6.17 Legend themes

```
base <- ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + theme_bw()
base + theme(legend.text = element_text(size = 15), legend.title = element_text(size = 15,
    face = "bold"))
```



The legend elements control the appearance of all legends. You can also modify the appearance of individual legends by modifying the same elements in `guide_legend()` or `guide_colourbar()`.

Element	Setter	Description
legend.background	`element_rect()`	legend background
legend.key	`element_rect()`	background of legend keys
legend.key.size	`unit()`	legend key size
legend.key.height	`unit()`	legend key height
legend.key.width	`unit()`	legend key width
legend.margin	`unit()`	legend margin
legend.text	`element_text()`	legend labels
legend.text.align	0–1	legend label alignment (0 = right, 1 = left)
legend.title	`element_text()`	legend name
legend.title.align	0–1	legend name alignment (0 = right, 1 = left)

6.18 Interactive plots with plotly

```
library(plotly)
p <- ggplot(mtcars, aes(factor(cyl), mpg)) + geom_boxplot()
ggplotly(p)
```

6.19 Chart junk

Source: Darkhorse Analytics

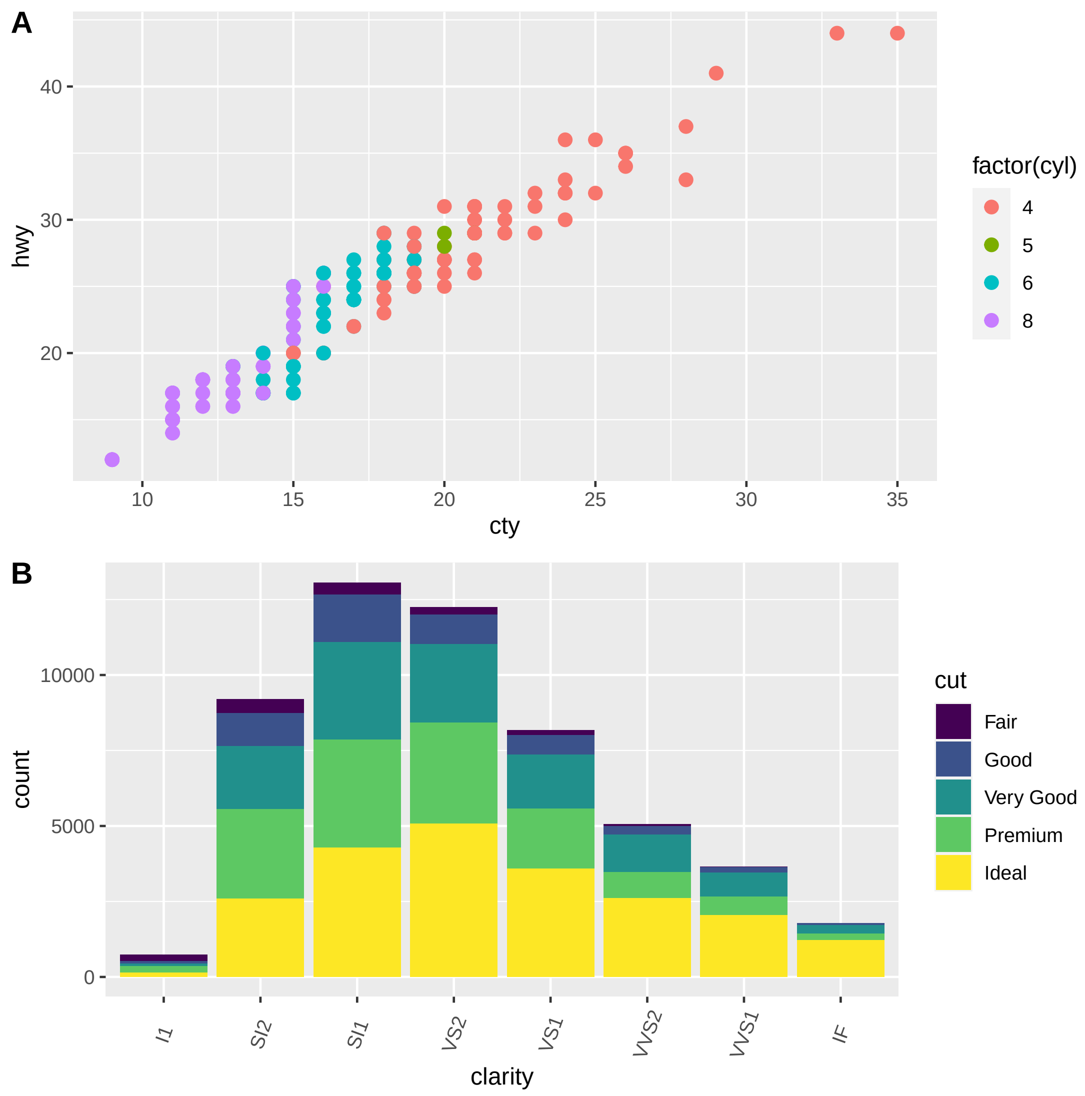
- poor **data-ink ratio**
- double encoding (color and axis encode the same)
- heavy or dark grid lines
- unnecessary text
- ornamented chart axes

- pictures within graphs
- shading, color gradients or 3D perspective

6.20 Assembling paper figures

`cowplot` package is a simple add-on to ggplot2. It provides publication-ready theme for ggplot2.

```
library(cowplot)
plot_mpg <- ggplot(mpg, aes(x = cty, y = hwy, colour = factor(cyl))) + geom_point(size = 2.5)
plot_diamonds <- ggplot(diamonds, aes(clarity, fill = cut)) + geom_bar() + theme(axis.text.x = element_tex
    vjust = 0.5))
plot_grid(plot_mpg, plot_diamonds, labels = c("A", "B"), nrow = 2)  # can modify ncol or nrow
```



Chapter 7

Reproducible data analysis with Snakemake

Basic idea:

- Decompose workflow into “rules” (steps).
- Rules define how to obtain output files from input files.
- Snakemake infers dependencies and execution order with a Directed Acyclic Graph.
- Any change in the input, only affected downstream rules will be executed.
- Disjoint paths in the DAG of jobs can be executed in parallel.
- Re-run all workflows with single `snakemake` command.

Example

```
rule sort:
    input:
        "path/to/dataset.txt"
    output:
        "dataset.sorted.txt"
    shell:
        "sort {input} > {output}"
```

```
rule sort:
    input:
        a="path/to/{dataset}.txt"
    output:
        b="{dataset}.sorted.txt"
    run:
        with open(output.b, "w") as out:
            for l in sorted(open(input.a)):
                print(l, file=out)
```

```
DATASETS = ["D1", "D2", "D3"] # use native python code

rule all:
    input:
        expand("{dataset}.sorted.txt", dataset=DATASETS)

rule sort:
    input:
        "path/to/{dataset}.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "sort {input} > {output}"
```

Chapter 8

Quizes

8.1 data.table

1. From DT, display a named vector with the means of *y* and *v*. The names of the elements are mean_y and mean_v. Hint: remember which command returns a vector and which one a data.table.

   A. DT[, c(mean_y = mean(y), mean_v = mean(v))]

   B. DT[, list(mean_y = mean(y), mean_v = mean(v))]

   C. DT[, .(mean_y = mean(y), mean_v = mean(v))]

2. As part of a new project, you got the results of a set of experiments from a lab in a data.table (x) with 3 columns: experiment id, sample id and value. On each experiment, more than one sample was measured. On another data.table (y) you got the dates of all the experiments the lab has made after a certain date. Some experiments are not part of your project, but the lab did not subset the table. Before a certain date, the lab could not find the experiments date, but you don't want to discard those results. You want to merge both data tables in order to have only one with 4 columns: experiment id, sample id, value and experiment date. Which merge would you use? Hint: Sketch both data tables if necessary.

   A. Inner, all = FALSE

   B. Full, all = TRUE

   C. Left, all.x = TRUE

   D. Right, all.y = TRUE

8.2 tidy data

1. What transformation are required to tidy 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
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

A. Cast

B. Melt

C. Cast and Melt

D. Data are tidy already

2. What transformation are required to tidy 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	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

A. Gather

B. Unite

C. Melt and cast

D. Melt, unite and cast

8.3 ggplot2

1. What's the result of the following command? `ggplot(data = mpg)`. Hint: ggplot builds plot layer by layer.

   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

D. All data in `mpg` will be visualized

2. What's the result of the following command: `ggplot(data = mpg, aes(x = hwy, y = cty))`. Hint: ggplot builds plot layer by layer

   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

   D. A scatter plot will be produced

3. What's the result of the following command: `ggplot(data = mpg, aes(x = hwy, y = cty)) + geom_point()`. Hint: ggplot builds plot layer by layer.

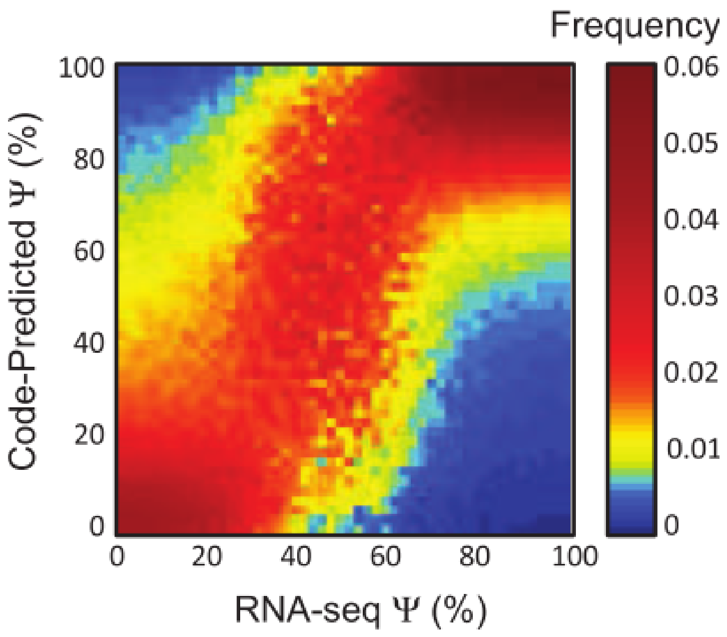
   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

   D. A scatter plot will be produced

4. What's wrong with the following plot?



Xiong et al., Science 2014

5. Which item is *not* chart junk?

   A. a bright red plot border

   B. light grey major grid lines

   C. bold labels and yellow grid lines

   D. data labels in Batik Gangster font

Chapter 9

Quiz solutions

9.1 data.table

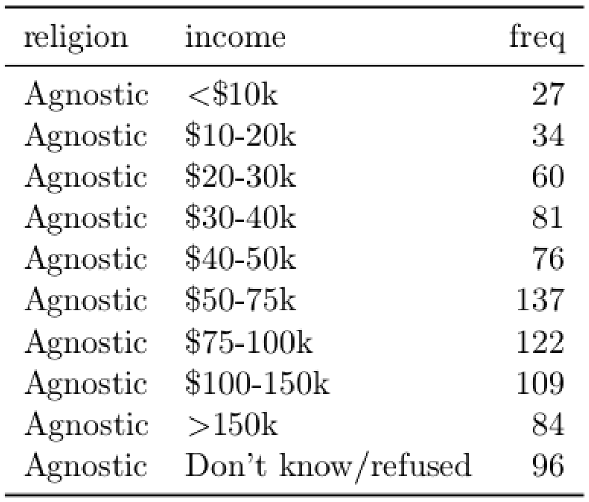
1. A, including the column names inside c() will return a vector.

2. C, inner join will omit the experiments whose dates are not provided. Full join will add rows of the experiments we are not interested in. Left join will leave the rows we are interested in and simply add the dates when available and NAs when not. Right join will leave us with experiments we are not interested and omit ones that we are interested but don't have the date.

9.2 tidy data

1. B, melt

   Tidy form:

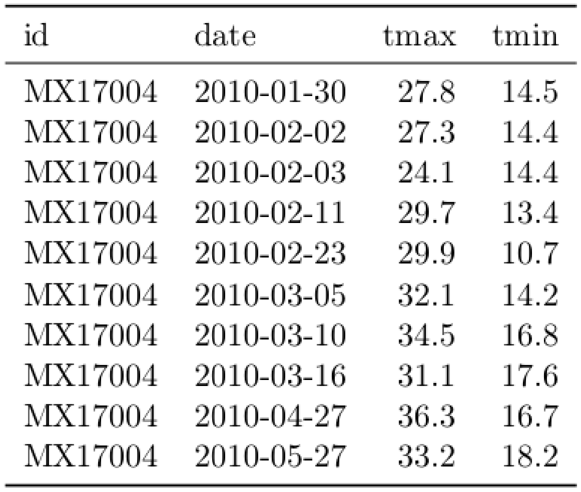


religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Figure 9.1: Tidy religion dataset

2. D 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

Figure 9.2: Tidy weather dataset

9.3 ggplot2

1. B, because neither variables were mapped nor geometry specified.
2. C, because while axis x and y are mapped, no geometry is specified.
3. D Axis x and y are mapped. But no geometry specified.
4.
    1. Rainbow color where there are no breakpoints. Color boundaries are created based on author’s interest.
    2. Manually twisted color scale.
5. B, light grey does not draw attention and grid lines are less important than data.