A Quick and focused overview of R data types and ggplot2 syntax

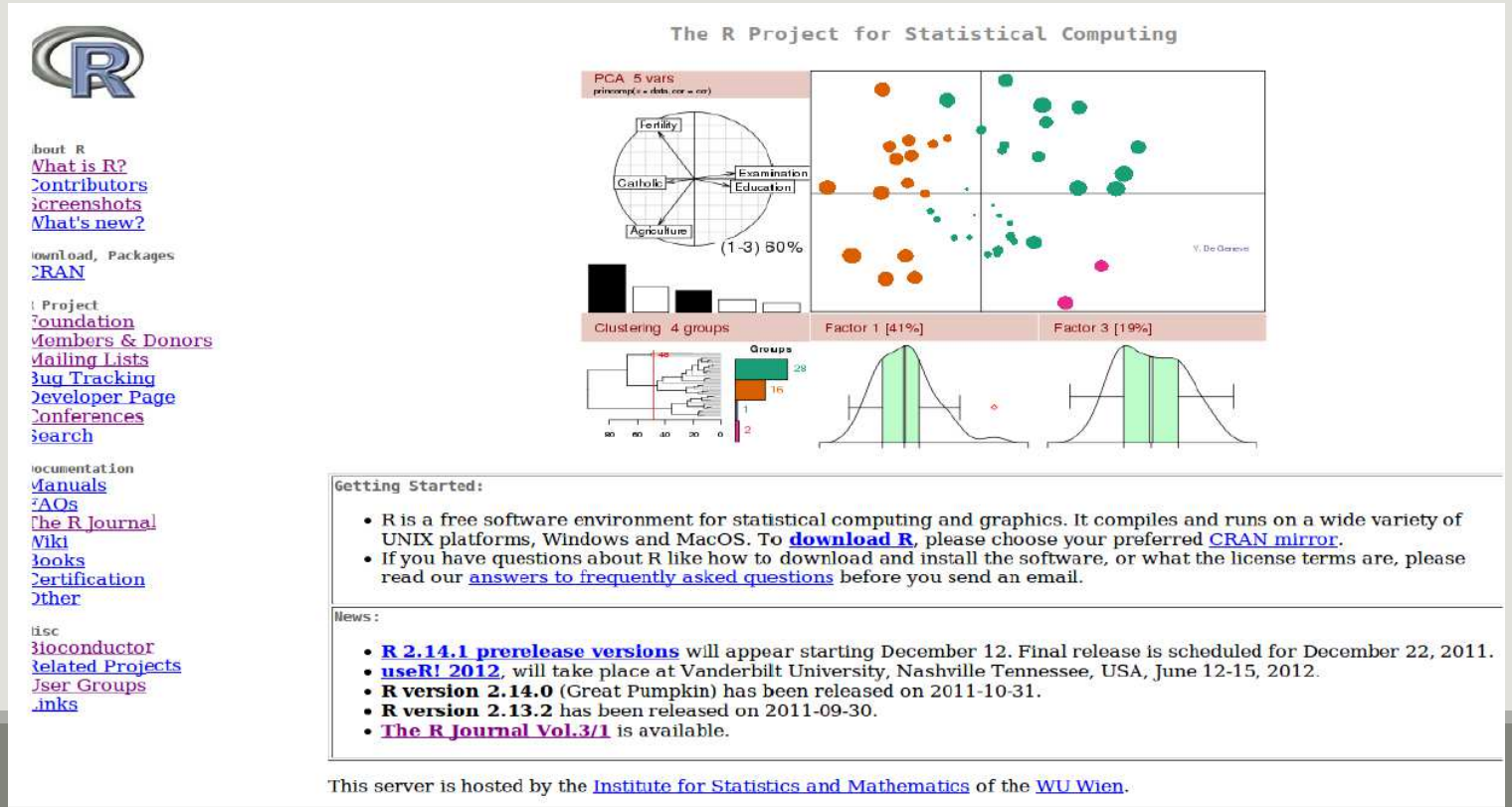
MAHENDRA MARIADASSOU, MARIA BERNARD, GERALDINE PASCAL, LAURENT CAUQUIL

R and RStudio

OVERVIEW

R and RStudio

R is a free and open environment for computational statistics and graphics (Open source, Open development, under GNU General Public Licence): http://www.r-project.org/

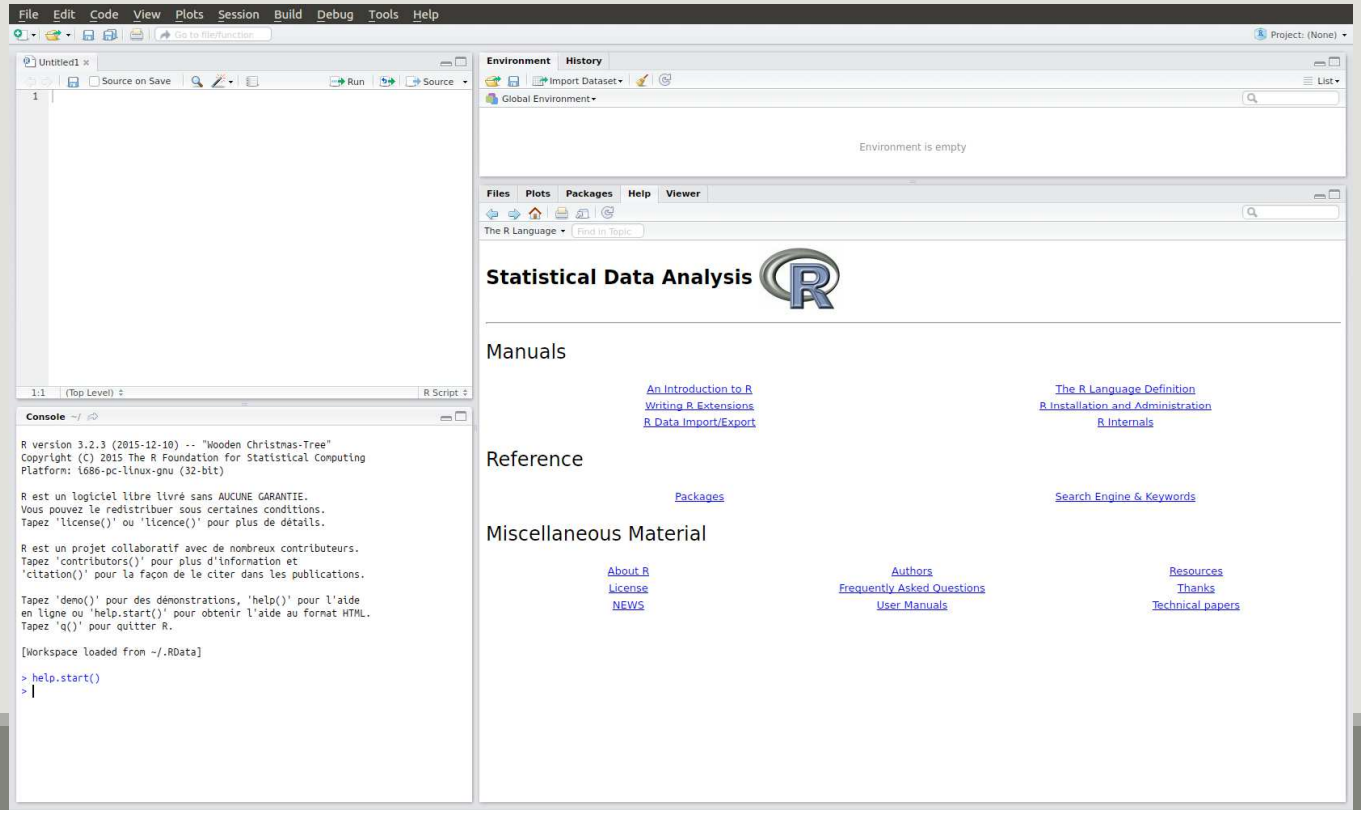


R and RStudio

- R is an interpreted language
- There is no compilation
- One can work in the console (this tutorial) or in an script file
- Good for interactive use of the language
- Bad for speed (when performing heavy computations)

R and RStudio

RStudio provides a nice front-end to R with 4 panels (script, console, workspace, graphics) : https://www.rstudio.com/



R and RStudio

Installing packages

- From CRAN :
  - The main strength of R comes from the thousands of packages that provide nice functions and utilities to the language. Most are available from the CRAN (Comprehensive R Archive Network) and easy to install:
  - install.packages("package_name")
- From Bioconductor :
  - Bioconductor , is an other repository. It stores packages dedicated to biology analysis
  - source("http://bioconductor.org/biocLite.R")
  - biocLite ("package_name")

R and RStudio

Loading packages is equally easy:

```
library(ggplot2)   # comming from CRAN
library(phyloseq)  # comming from Bioconductor
```

- Most packages must be loaded at each new session (see the ”Packages” tab in RStudio)

R and RStudio

Getting help

Packages include help files for the functions they provide.

- For a particular function
  - help("function name") leads to the help page of function name

Try it !

```
help("mean") ## or ?mean
```

Widely used packages include detailed files called "vignette" for the functions they provide

- For a particular packages
  - vignette("vignette name")

Try it !

```
vignette("extending-ggplot2")
vignette("phyloseq-basics")
```

R and RStudio

The console is a gloried calculator,

- you submit some R code and press Enter
- R evaluates the expression and returns the answers

```
2+2
```

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

When using RStudio, you can use "CTRL + Enter" to execute some code from the script (as opposed to "Enter" to execute it from the console).

R and RStudio

Variable assignment

- You can save the value of some R code using the "arrow operator": <-
- The syntax is simple: variable_name <- value

```
a <- 2*4
```

- And you can access and manipulate the value of that variable

```
a
```

```
## [1] 8
```

```
a/2
```

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

R and RStudio

Variable assignment

The arrow is also used to change the value of an object:

```
a <- 4
a
## [1] 4
```

Modifications made to a copy do no impact the original object:

```
b <- a; b <- 8 # ";" simply separates two commands
a; b
## [1] 4
## [1] 8
```

R and RStudio

DATA/VARIABLE

Data/Variable

In R every basic object has four characteristics:

- a name
- a mode
- a length
- a content

The three main modes are numeric, logical, character

Data/Variable

The class function return the mode of a variable

Numeric	Character	Logical

`x <- 1`
`class(x)`
`## [1] "numeric"`
	`x <- "hello"`
	`class(x)`
	`## [1] "character"`
		`x <- TRUE`
		`class(x)`
		`## [1] "logical"`

- a logical can only take value TRUE or FALSE
- a character can be defined using simple (') or double (") quotes

Data/Variable : conversion

When possible, the functions as.something change a variable from one type to another:

```
as.numeric("5")
## [1] 5
as.logical(0.0)
## [1] FALSE
```

```
as.numeric(TRUE)
## [1] 1
as.character(TRUE)
## [1] "TRUE"
```

```
as.numeric("5.56")
## [1] 5.56
as.logical(2)
## [1] TRUE
```

But sometimes fail (producing NA, Not Available) when the conversion is not properly defined:

```
as.numeric("INRA")
## Warning: NAs introduced by coercion
## [1] NA
```

Character is more general than numeric, itself more general than logical.

Data/Variable : conversion

Guess the results of the following commands and check your guesses in the console:

```
## Numeric
as.numeric(2/3)
as.numeric(5.67)
as.numeric(FALSE)
as.numeric(TRUE)
as.numeric("5.67")
as.numeric("MaIAGE")
```

```
## Character
as.character(2/3)
as.character(5.67)
as.character(FALSE)
as.character(TRUE)
as.character(5)
as.character(5+7)
```

```
## Logical
as.logical(2/3)
as.logical(0)
as.logical("45")
as.logical("MaIAGE")
```

Using the conversion rules from logical to numeric, guess the value of:

```
TRUE + TRUE + FALSE * TRUE + TRUE * TRUE
```

Data/Variable : special value

There are special values in R, in particular

- NA which stands for Not Available and is a code for missing data

```
a <- NA; length(a); is.na(a)
```

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

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

Data/Variable : structure

R offers many data structures to organize data. The main ones are:

- vector (1D array)
- factor
- matrix (2D array)
- data.frame

Data/Variable : vector

- Multiples elements of the **same mode** (numeric, character, logical) can be collected in a vector (1D array) using the c command:

```
x <- c(2, 4, 8, 9, 0)
x
## [1] 2 4 8 9 0
```

- Elements of x can be accessed with the indexing operations:

```
x[1] ## first element
## [1] 2
```

```
x[c(3, 5)] ## third and fifth elements
## [1] 8 0
```

- Elements of different types are coerced to the most general mode before collection:

```
c(3.4, 2, TRUE)
## [1] 3.4 2.0 1.0
```

```
c(3.4, "MaIAGE", TRUE)
## [1] "3.4" "MaIAGE" "TRUE"
```

Data/Variable : vector

If x is a named vector, elements can be accessed by name rather than by position:

```
x <- c("A" = 1, "B" = 4, "C" = 9)
```

```
x
```

```
## A B C
## 1 4 9
```

Guess :

```
x[1]
```

```
## A
## 1
```

```
x["C"]
```

```
## C
## 9
```

Data/Variable : vector

Names can be set or changed after creating a vector using the function names

```
x <- c(1, 4, 9)
x
## [1] 1 4 9
names(x) <- c("first", "second", "third")
x
## first second third
## 1 4 9
```

Exercice : Guess the result of the following code, check your guess in the console:

```
x <- c("O", "G", "F", "S", "R")
x[c(3, 5, 1, 2, 4)]
## "F" "R" "O" "G" "S"
```

Data/Variable : vector

Logical indexing

A vector x can be indexed by a logical vector index specifying which elements should be kept. In that case, index and x should have the same length ...

```
x <- 1:6
index <- c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE)
x[index] ## = x[c(1, 3, 4)]
## [1] 1 3 4
```

...otherwise strange things can happen

```
index <- c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, TRUE)
x[index] ## = x[c(1, 3, 4, 7)] but x[7] does not exist
## [1] 1 3 4 NA
```

Data/Variable : vector

Exercice:

Try to reorder this rank's vector thanks to position index!

```
rank <-c("Order", "Kingdom", "Genus", "Class", "Family",
"Species", "Phylum")

reordered_rank <- rank[c(2, 7, 4, 1, 5, 3, 6)]
```

Data/Variable : matrix

Matrices are essentially 2-D vectors: all elements must have the **same mode**.

Indexing works the same way as for vectors but with two indices: the first for rows, the second for columns.

```
x <- matrix(1:18, nrow = 3, ncol = 6)
x
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    4    7   10   13   16
## [2,]    2    5    8   11   14   17
## [3,]    3    6    9   12   15   18

x[2, 4] ## element in 2nd row, 4th column
## [1] 11
x[, 2] ## 2nd column
## [1] 4 5 6
```

```
x[2, ] ## 2nd row
## [1] 2 5 8 11 14 17
```

Data/Variable : matrix

```
x <- as.matrix(read.csv("data/introR/matrix.tsv", sep= "\t", row.names=1))
x
      sample1     sample2      sample3
otu_1 45          60           0
otu_2 10          5            21
otu_3 0           54           32
```

Try to guess what the following commands do, check in the console

```
x[ , 3]                    x["otu_2",]
x[c(1, 2), ]               x[c(1, 3), c(2, 3)]
```

How to access to the count of sample2 for otu_3 ?

```
x["otu_3", "sample2" ]              x[3, 2 ]
x["otu_3", 2 ]                      x[3, "sample2" ]
```

Data/Variable : factor

Factors are used for categorical variables that only take a finite number of values (also called levels)

```
x <- factor(c("male", "male", "female", "male", "female"))
class(x)
## [1] "factor"
Levels can be accessed with levels
levels(x)
## [1] "female" "male"
```

Internally, R treats x as an integer vector and associates each level to a value:

here 1 = "female", 2 = "male" (alphabetical order by default) so that x = c(2, 2, 1, 2, 1).

Data/Variable : factor

Sometimes it's convenient to impose a different ordering with the argument levels of the factor function.

```
x <- as.factor(c("strong", "strong", "weak", "middle", "weak"))
levels(x)
## [1] "middle" "strong"" "weak"
y <- factor(x, c("weak", "middle", "strong"))
levels(y)
## [1] "weak" "middle" "strong"
```

Data/Variable : data.frame

A data.frame is a table-like structure (created with the function data.frame) used to store contextual data of different modes. Technically a data.frame is a list of equal-length vectors and/or factors.

```
x <- data.frame(number = c(1:4),
       group = factor(c("A", "A", "B", "B")),
       desc = c("riri", "fifi", "lulu", "picsou"),
       stringsAsFactors = FALSE)
```

```
x
##   number group  desc
## 1      1     A  riri
## 2      2     A  fifi
## 3      3     B  lulu
## 4      4     B picsou
class(x)
## [1] "data.frame"
```

```
class(x[ , 1])
## [1] "integer"
class(x[ , 2])
## [1] "factor"
x[2, "desc"] ## or x[2, 3]
## [1] "fifi"
```

Data/Variable : data.frame

A data.frame has two dimensions: rows and columns (just like a matrix)

```
dim(x);nrow(x);ncol(x)
```

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

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

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

Its columns can be named and accessed with the special operator $

```
x$group
```

```
## [1] A A B B
```

```
## Levels: A B
```

Data/Variable : data.frame

Guess what the following code does and check in the console.

```
x
##    ID  group        value
## 1   1      A   1.29891241
## 2   2      A  -0.06922655
## 3   3      A  -0.21717540
## 4   4      A  -0.23028309
## 5   5      A  -0.17481615
## 6   6      B  -1.30304922
## 7   7      B  -1.27979172
## 8   8      B  -1.54874545
## 9   9      B  -0.64328443
## 10 10      B   0.20690014
```

```
ii <- 1:5

df <- x[ii, c("ID", "value")]

df

df[ , 2]

class(df[ , 2])

df[2, ]

class(df[2, ])
```

Data/Variable: summary

- vector (and matrix): 1-D (and 2-D) array of basic data, all of the same type (integer, numeric, logical, character)
- factor: used for categorical data, collection of elementary variables that can only take a finite number of values (e.g. small, medium, large)
- data.frame: used for experimental results, a table-like structure (technically, a list of equal-length vectors). All elements in a column have the same type but different columns may have different types

Data/Variable: summary

- position: index elements by position in a `vector/factor` (`x[i]`) or 2 positions (row, column) in a `matrix/data.frame` (`x[i, j]`)
- name: index elements by name in a `vector/factor` (`x["first"]`) or 2 names (row, column) in a `matrix/data.frame` (`x["row", "column"]`)
- logical index: use a logical mask index of the same size as x that specifies which elements to keep (`x[index]`)
- names with $ (for list): use a component's name to extract it from a list. Works for `data.frame` which are a special kind of list(`x$name`)

More than one element (or row, column) can be indexed at the same time with a vector of position/name/logical : `x[c(i1, i2, ..., in)]`

Data/Variable : filtering

R provides a built-in way to build logical indexes using logical operations (e.g. to filter data)

```
x <- 11:15 ; x
## [1] 11 12 13 14 15
z <- (x < 13); z ## the first command returns a logical vector
## [1] TRUE TRUE FALSE FALSE FALSE
z <- (x < 14) & (x > 11); z ## logical AND
## [1] FALSE TRUE TRUE FALSE FALSE
z <- (x < 12) | (x > 14); z ## logical OR
## [1] TRUE FALSE FALSE FALSE TRUE
!z ## logical NOT
## [1] FALSE TRUE TRUE TRUE FALSE
```

Data/Variable : filtering

The logical indexes can be transformed to integer indexes using which

```
which(z)
```

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

and used to extract part of the data

```
z <- which(x < 14)
```

```
x[z]
```

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

```
## or equivalently
```

```
x[x < 14]
```

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

Data/Variable : import

The simplest way to import a tabulated text file* is read.table()

read.table() outputs a data.frame and is very flexible. Its main arguments are:

Argument	Description
file	File name, or complete path to file (can be an URL)
header	First line = variable names? (**FALSE** by default)
sep	Field separator character (**white character** by default), write "\t" for tabulation.
dec	Character used for decimal points ("." by default)
na.string	Character vector of strings to be interpreded as NA (**NA** by default)
row.names	Column number (or name) where the rownames are stored.

* : think excel worksheet, but in text format

Data/Variable : export

Matrix-like objects (matrices, data.frame) can be exported as tabulated text files (human-readable) with write.table().

The typical use is:

```
## for tsv
write.table(matrix_object, file = "my_file.tsv", sep = "\t")
```

To save several objects as R objects in one file (more compact), use save() (and load() to load them back).

```
save(object1, object2, file = "data.RData")
load("data.RData")
```

Finally, save.image() is a shortcut to save the complete workspace.

* : think excel worksheet, but in text format

R and RStudio : website

- http://www.r-project.org/
- http://www.bioconductor.org/help/publications/
- https://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf

ggplot2

OVERVIEW

ggplot2 : overview

- ggplot2 is a powerful package by Hadley Wickham to produce elegant statistical graphics
- it has relatively simple syntax
- gg stands for grammar of graphics (Leland Wilkinson, 2005)
- the plot is built one component at a time with smart defaults settings

```
library(ggplot2)
```

ggplot2 : overview

These slides are not a complete introduction to ggplot2. They only intend to introduce elements used in the phyloseq training session and therefore to :

- present the *syntax* of a ggplot
- present simple *examples* of ggplot graphs
- illustrate the data to visual characteristics mapping
- show how to *modify* a graph by:
  - adding a custom color scale
  - changing the color scale
  - subdividing the data to draw small multiple plots

ggplot2

BUILD A PLOT

ggplot2: overview

A ggplot is composed of:

- At least
  - data stored as a data.frame
  - aesthetics: visual characters that represent the data (which variables for coordinates x, y, or color, and aesthetics caracteristics: fill, position, size, etc.)
- and additionnal layers
  - scales: for each aesthetic, the conversion from data to display value (color scale, size scale, transparency scales, log-transformation of continuous values, etc)
  - geoms: type of geometric objects used to represent the data (points, line, bar, etc.)
  - facets: a way to split the data into subsets (e.g. male only/female only) and to represent data as small multiple plots

The general syntax is

```
p <- ggplot(data, aes(x, y)) + layer1 + layer2 + ...
```

ggplot2 : diamonds dataset

We'll work with the built-in diamonds dataset (10 attributes of almost 54000 diamonds, see ?diamonds for details)

```
data(diamonds) ## import datasets
```

```
class(diamonds) ## data.frame
```

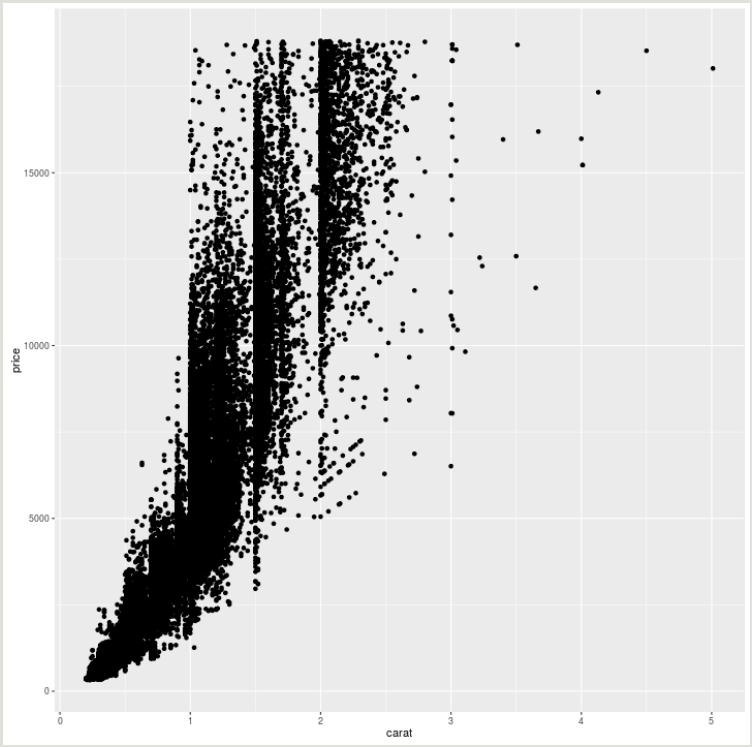
```
head(diamonds) ## first lines of the data.frame
```

```
  carat       cut color clarity depth table price    x    y    z
1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
```

```
help(diamonds) ## description of dataset
```

ggplot2 : build a plot

```
## set base plot, x coordinate is carat, y is price
p <- ggplot(diamonds, aes(x = carat, y = price))

## Add a layer to represent data as point
p1 <- p + geom_point()

plot(p1)
```



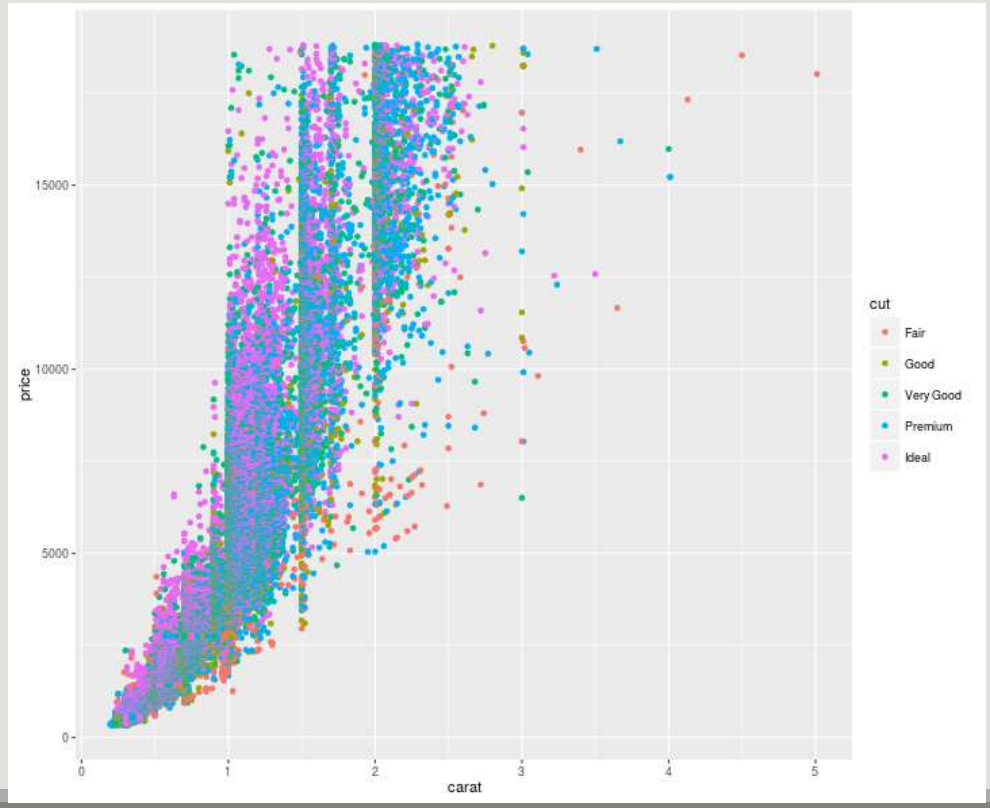
ggplot2: build a plot, aesthetics

- The first command line tells ggplot that
  - data is stored in the diamonds data.frame
  - global aesthetics (set with aes) are as follows : *carat* is mapped to x coordinate, *price* to y coordinate
- The second one adds a layer in which data are represented by points (geom_point)

ggplot2 : build a plot, aesthetics

ggplot allow to add easily color scale in function of an other variable

```
## set base plot, x coordinate is carat, y is price and colored by cut
p <- ggplot(diamonds, aes(x = carat, y = price, color = cut ))
p2 <- p + geom_point()
plot(p2)

## Or precise color aesthetics in geom_point function
p <- ggplot(diamonds, aes(x = carat, y = price))
p2 <- p + geom_point(aes(color=cut))
plot(p2)
```



NB : For color scale you must choose variables with finite number of values.

ggplot2: build a plot, aesthetics

- Local aesthetics (aes(color = cut))can be added for the point layer (geom_point()).
  - *cut* value is mapped to the color of the points and both a legend and a color scale are automatically constructed

If local aesthetics is:

- identical for all points: the argument must be given outside of aes.

```
geom_point(color = "black"))
```

- mapped to a variable value (here cut): the argument must be given inside of aes.

```
geom_point(aes(color = cut) )
```

ggplot2: build a plot, aesthetics

We played with color but with geom_point we can also play with

- shape
- size
- alpha (transparency)
- fill

ggplot2: build a plot, aesthetics

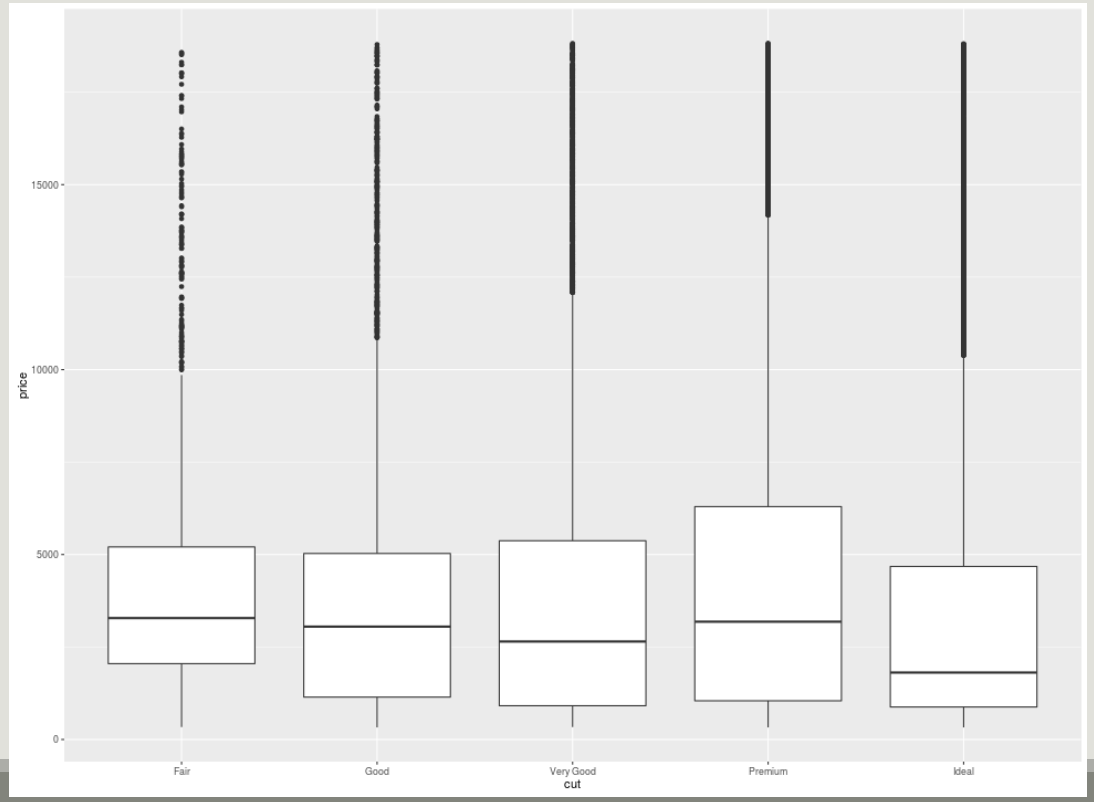
About geom:

- Here we used geom_point to represent data as points. We could have used other geometric representations of the data:
  - geom_line
  - geom_bar
  - geom_density
  - geom_boxplot
  - geom_histogram
- Each geometry expects and accepts different aesthetics (e.g linetype is useful for lines but useless for points)

ggplot2: build a plot, facetting

Try to represent the distribution of **price** in function of **cut** thanks to a **boxplot**.

```
p <- ggplot(diamonds, aes(x =
 cut, y = price)
p3 <- p + geom_boxplot()
plot(p3)
```

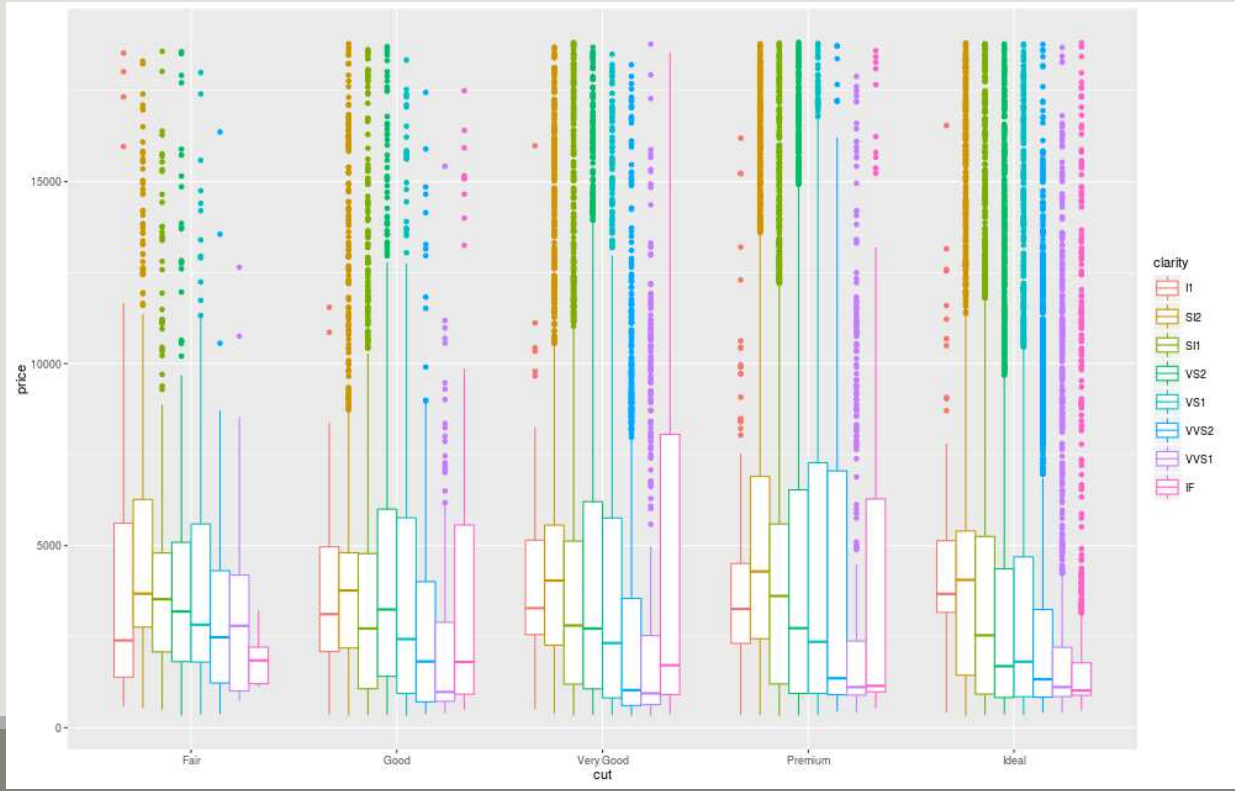


NB : For boxplot you must choose variables with finite number of values for x.

ggplot2: build a plot

Add some color in function of **clarity**

```
p4 <- ggplot(diamonds, aes(x =
 cut, y = price, color =
 clarity)) + geom_boxplot()
plot(p4)
```



ggplot2: build a plot, facetting

Go back to geom_point plot of **price** in function of **carat** colored by **cut**

```
p2 <- ggplot(diamonds, aes(x = carat, y = price, color = cut )) +
geom_point()
```

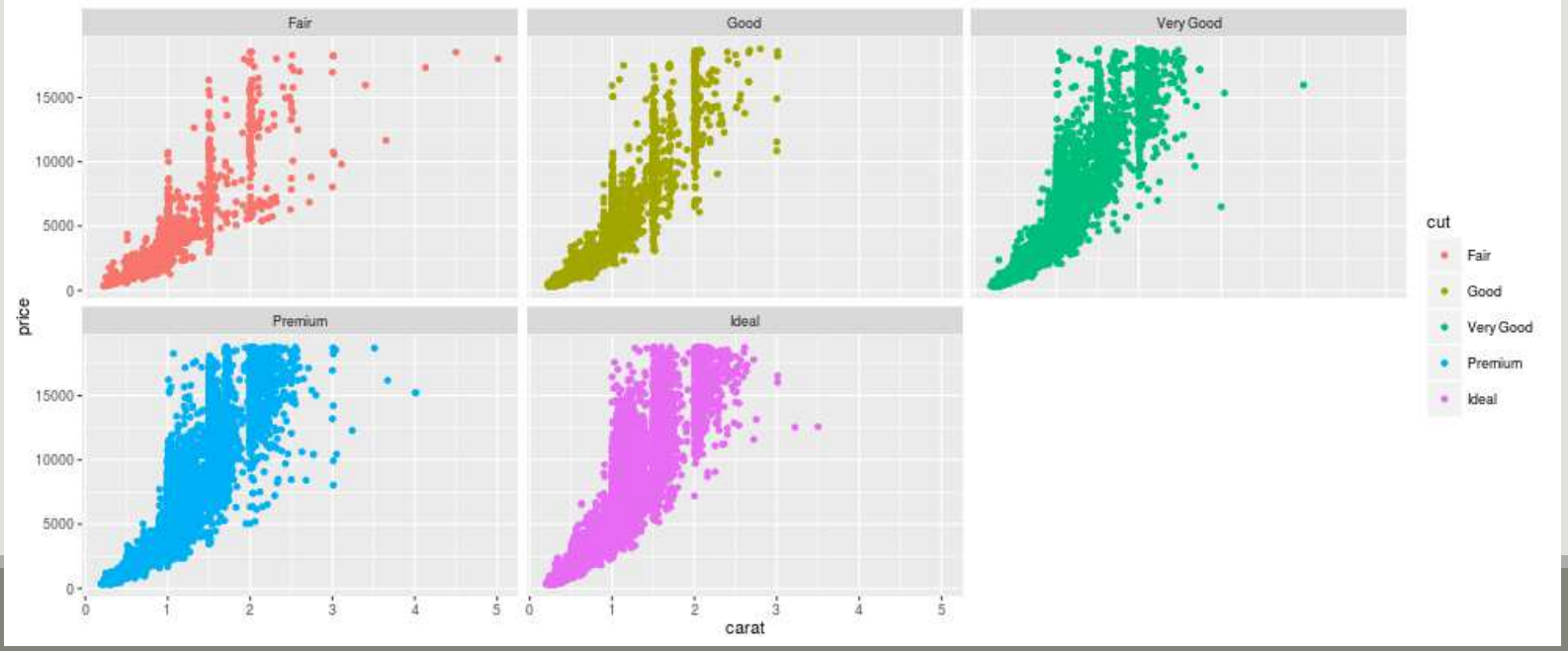
We can split the data in subsets to draw small multiple plots using facetting. There are two variants of facetting:

- facet_wrap if only one variable is used for facetting
- facet_grid, usually used for two or more variables (but can be used for one)

ggplot2: build a plot, facetting

Compare facet_wrap and facet_grid when using only one variable for facetting

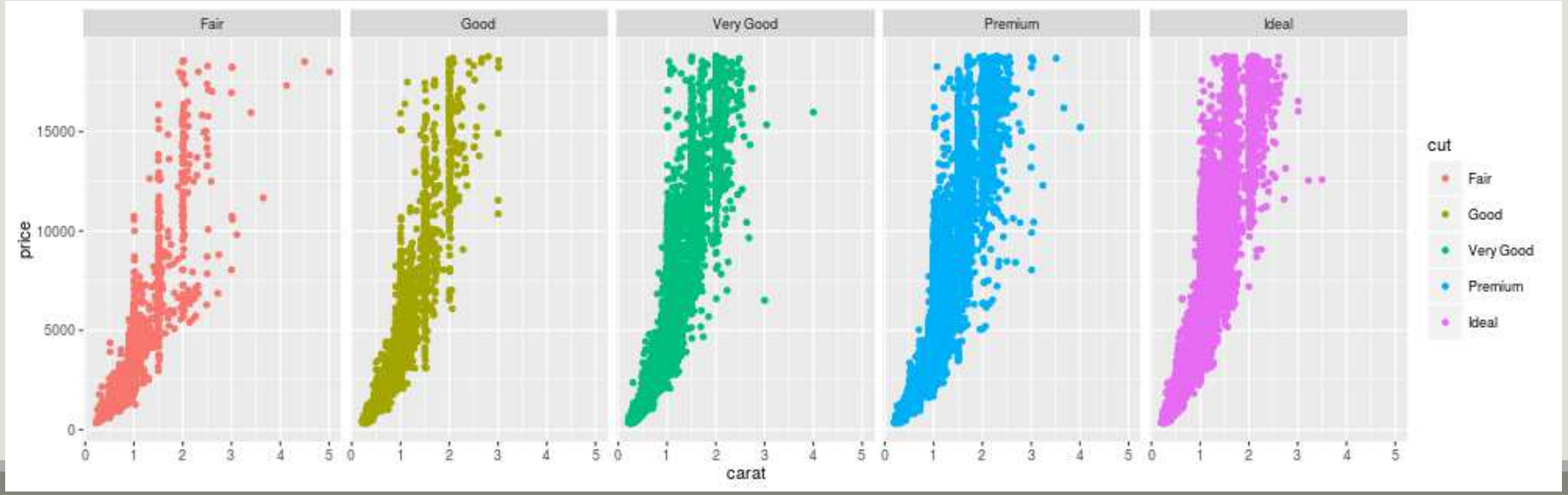
```
## facet along cut
p5 <- p2 + facet_wrap(~ cut)
plot(p5)
```



ggplot2: build a plot, facetting

Compare facet_wrap and facet_grid when using only one variable for facetting

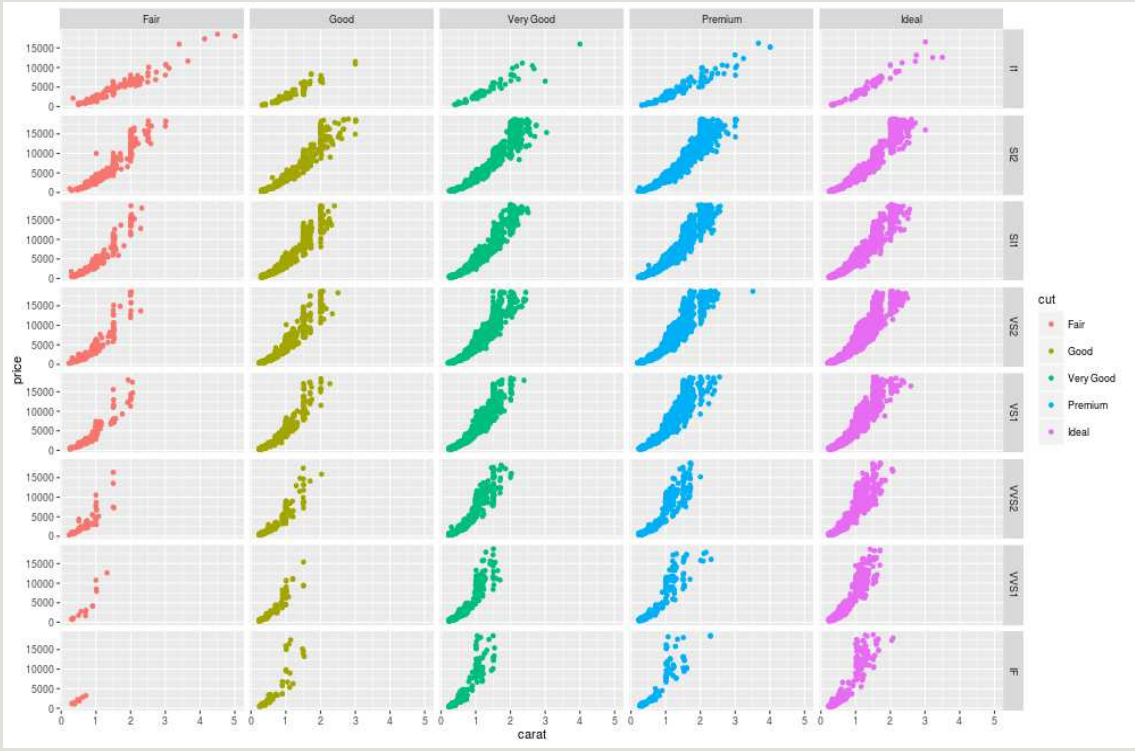
```
## facet along cut, only points from a given cut appear in a facet
p6 <- p2 + facet_grid(~ cut)
plot(p6)
```



ggplot2: build a plot, facetting

facet_grid is most useful when splitting the data along two factors

```
## facet along clarity(rows) * cut(column)
p7 <- p2 + facet_grid(clarity ~ cut)
plot(p7)
```



ggplot2: build a plot, facetting

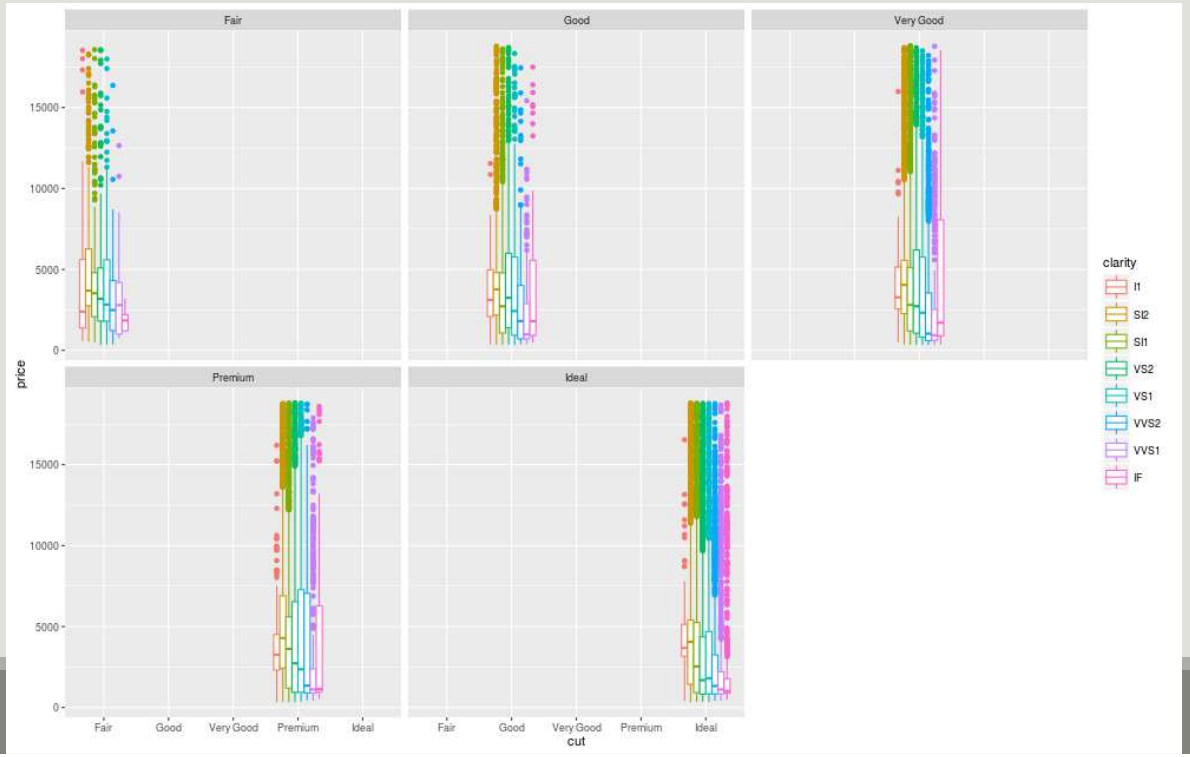
Sometimes, facetting wastes spaces when using same variable for coordinates and facetting.

On boxplot of **price** in function of **cut,** try to facet by cut.

```
p4 <- ggplot(diamonds, aes(x = cut, y = price, color = clarity))
+ geom_boxplot()
```

```
p8 <- p4 + facet_wrap(~cut)
plot(p8)
```

Each cut is represented in only one facet and the common x-scale wastes a lot of space



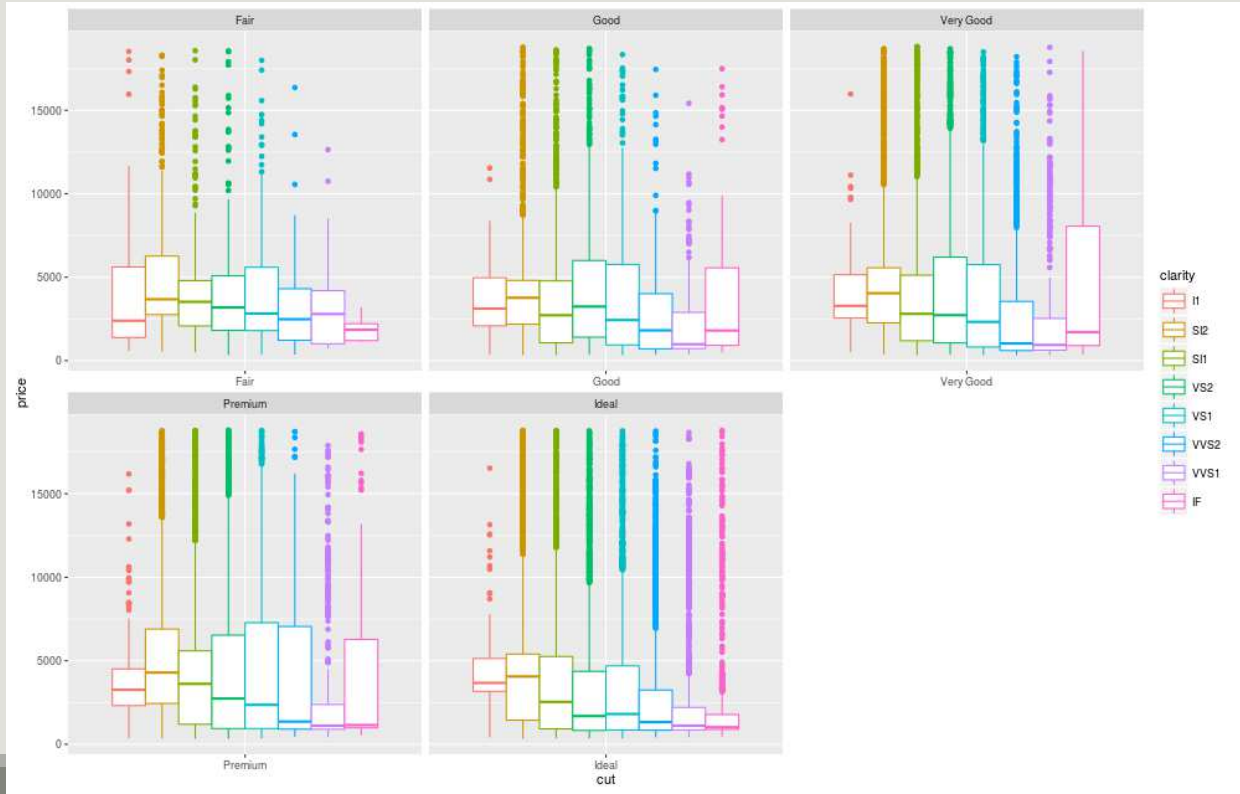
ggplot2: build a plot, facetting

We facet by cut but do not impose a common x-scale which leads to a much better use of space.

```
p9 <- p4 + facet_wrap(~cut,
scales = "free_x")

plot(p9)
```

scales = "free_y" would lead to one y-scale per facet

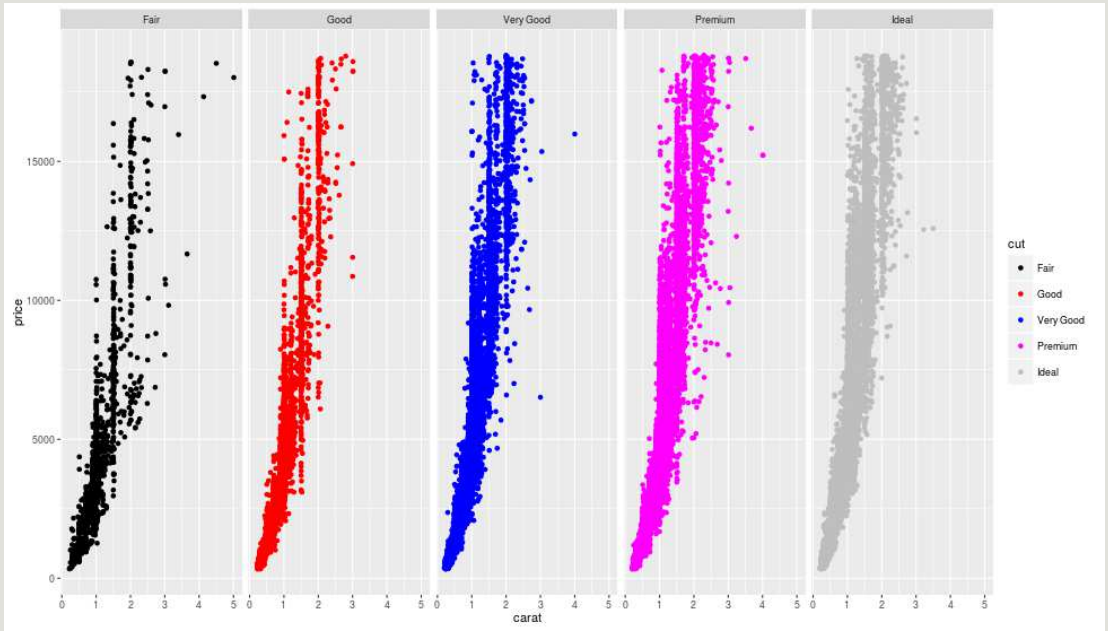
scales = "free" to one y-scale and one x-scale per facet



ggplot2: build a plot, color scales

cut is a factor, with a discrete number of values. We can change the color scale manually with the family of functions scale_color_something

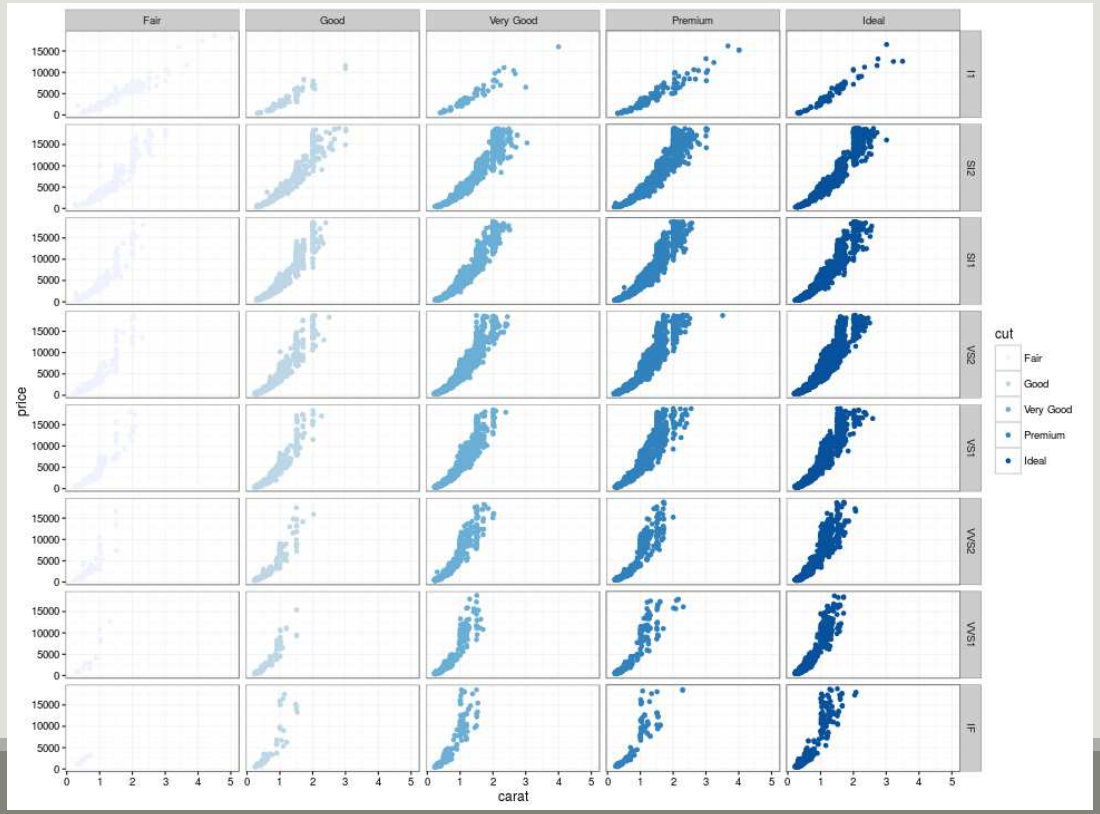
```
palette <- c("black", "red",
"blue", "magenta", "gray")

names(palette) <- c("Fair", "Good",
"Very Good", "Premium", "Ideal")

palette

## Manual color scale
p6.1 <- p6 +
scale_color_manual(values =
palette)
plot(p6.1)
```



ggplot2: build a plot, color scales

cut is a factor, with a discrete number of values. We can change the color scale manually with the family of functions scale_color_something

```
## Use built-in color palette
p7.1 <- p7 +
scale_color_brewer()
plot(p7.1)
```



ggplot2: build a plot, aesthetics

About scales:

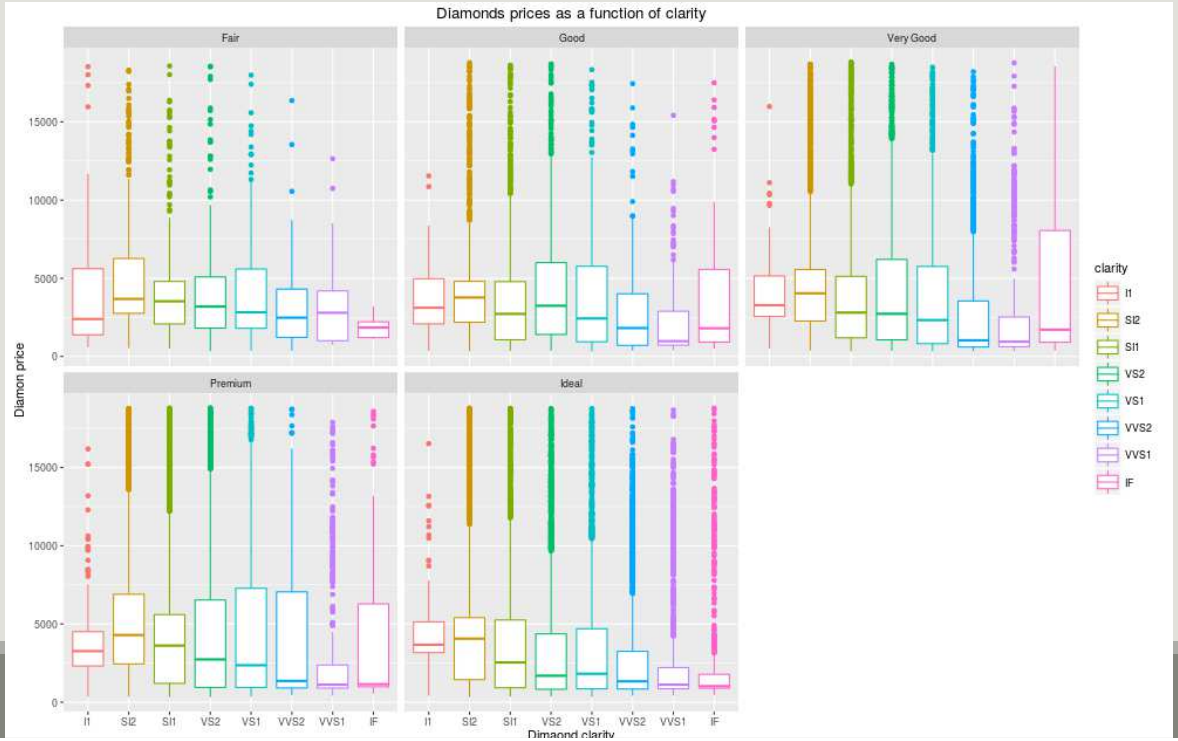
- Each aesthetic is associated with a scale
- Whenever possible, ggplot2 will try to merge the scales (like color and fill)
- For aesthetics mapped to a variable, the scale will vary depending on the nature of the variable: numeric (continuous), factor or logical (discrete)
- Every scale is built in the following way
  - they all begin with scale_ and
  - continue with the aesthetic name (linetype, fill, color)
  - and end with the name of the scale (manual, discrete, brewer)

ggplot2: build a plot, title and labels

You can add (or change) title and axis labels with the commands ggtitle, xlab and ylab.

```
p10 <- p9 + ggtitle("Diamond prices as a function of clarity") +
xlab("Diamond clarity") + ylab("Diamond price")
```

```
plot(p10)
```



ggplot2

EXPORT AND LEARN

ggplot2: export

- You can save graphics using ggsave,
- It guesses the file type from the filename extension
- By default, it saves the last plot with its current dimensions
- But you can override the dimensions at will

```
## the last three arguments are optional
ggsave("myplot.png", plot = p, width = 10, height = 4)
```

ggplot2: references

- docs.ggplot2.org/current
- http://groups.google.com/group/ggplot2
- http://cran.r-project.org/web/packages/ggplot2/index.html
- Wickman, "ggplot2. Elegant Graphics for Data Analysis" Springer, 212p.

Annexe

R : length function

The length() function returns the length of an object:

```
a <- 2
a
## [1] 2
length(a)
## [1] 1
```

In the previous example, a is a vector of length 1, with a single element

Hence the mysterious [1] in the output of a

R : rm function

Variable deletion

The rm() function is used to remove an object from the workspace:

```
a
## [1] 4
rm(a)
a ## a does not exist anymore
## Error in eval(expr, envir, enclos): objet 'a' introuvable
```