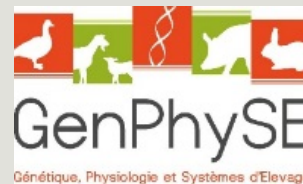


# A Quick and focused overview of R data types and ggplot2 syntax

---

MAHENDRA MARIADASSOU, MARIA BERNARD, GERALDINE PASCAL,  
LAURENT CAUQUIL



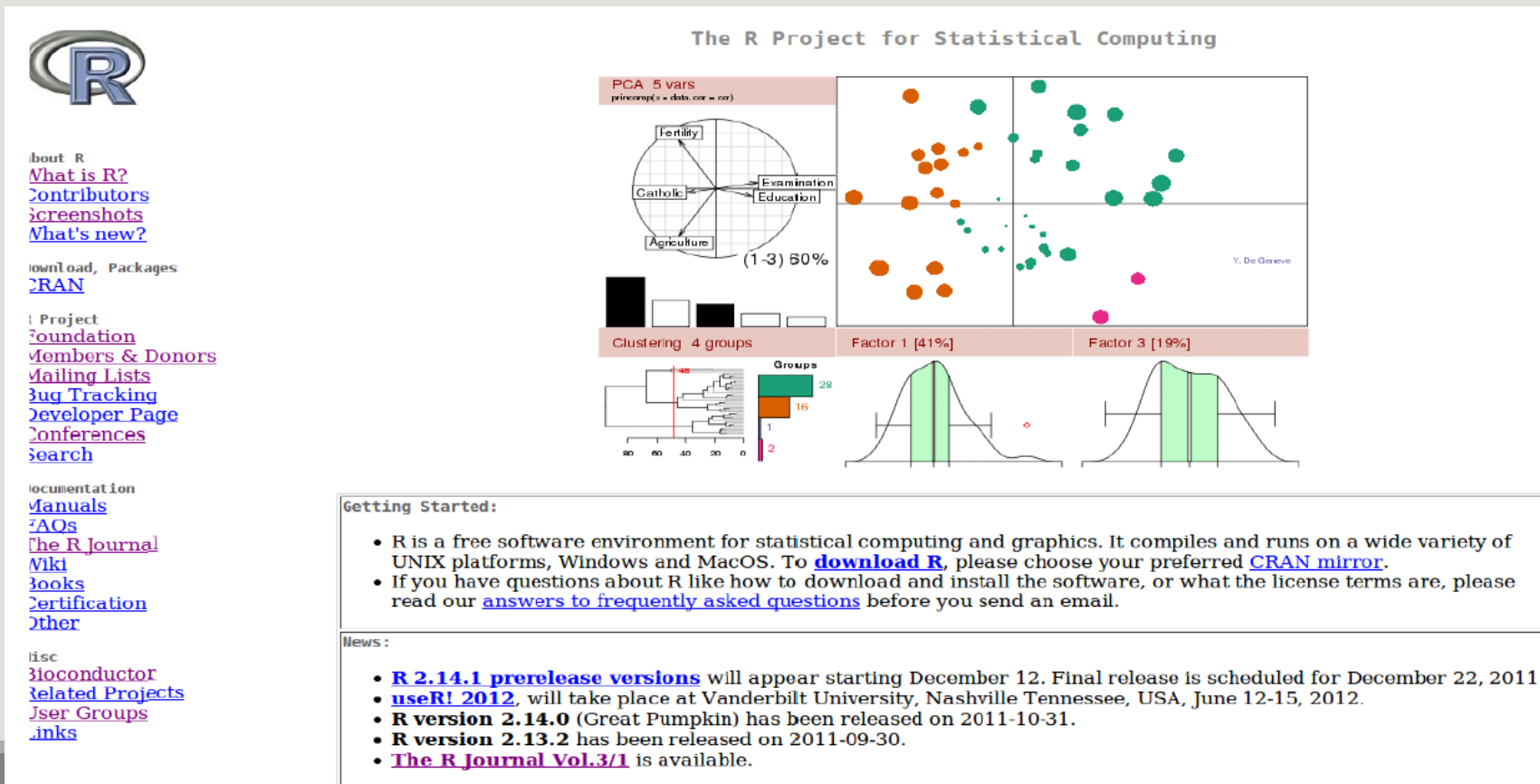
# R and RStudio

---

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/>



The R Project for Statistical Computing

PCA 5 vars  
prcomp(x = data, cor = cor)

Fertility  
Catholic  
Agriculture  
Examination  
Education  
(1-3) 60%

Clustering 4 groups  
Factor 1 [41%]  
Factor 3 [19%]

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred [CRAN mirror](#).
- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

News:

- [R 2.14.1 prerelease versions](#) will appear starting December 12. Final release is scheduled for December 22, 2011.
- [useR! 2012](#), will take place at Vanderbilt University, Nashville Tennessee, USA, June 12-15, 2012.
- [R version 2.14.0](#) (Great Pumpkin) has been released on 2011-10-31.
- [R version 2.13.2](#) has been released on 2011-09-30.
- [The R Journal Vol.3/1](#) is available.

About R  
[What is R?](#)  
[Contributors](#)  
[Screenshots](#)  
[What's new?](#)

Download, Packages  
[CRAN](#)

Project  
[Foundation](#)  
[Members & Donors](#)  
[Mailing Lists](#)  
[Bug Tracking](#)  
[Developer Page](#)  
[Conferences](#)  
[Search](#)

Documentation  
[Manuals](#)  
[FAQs](#)  
[The R Journal](#)  
[Wiki](#)  
[Books](#)  
[Certification](#)  
[Other](#)

Misc  
[Bioconductor](#)  
[Related Projects](#)  
[User Groups](#)  
[Links](#)

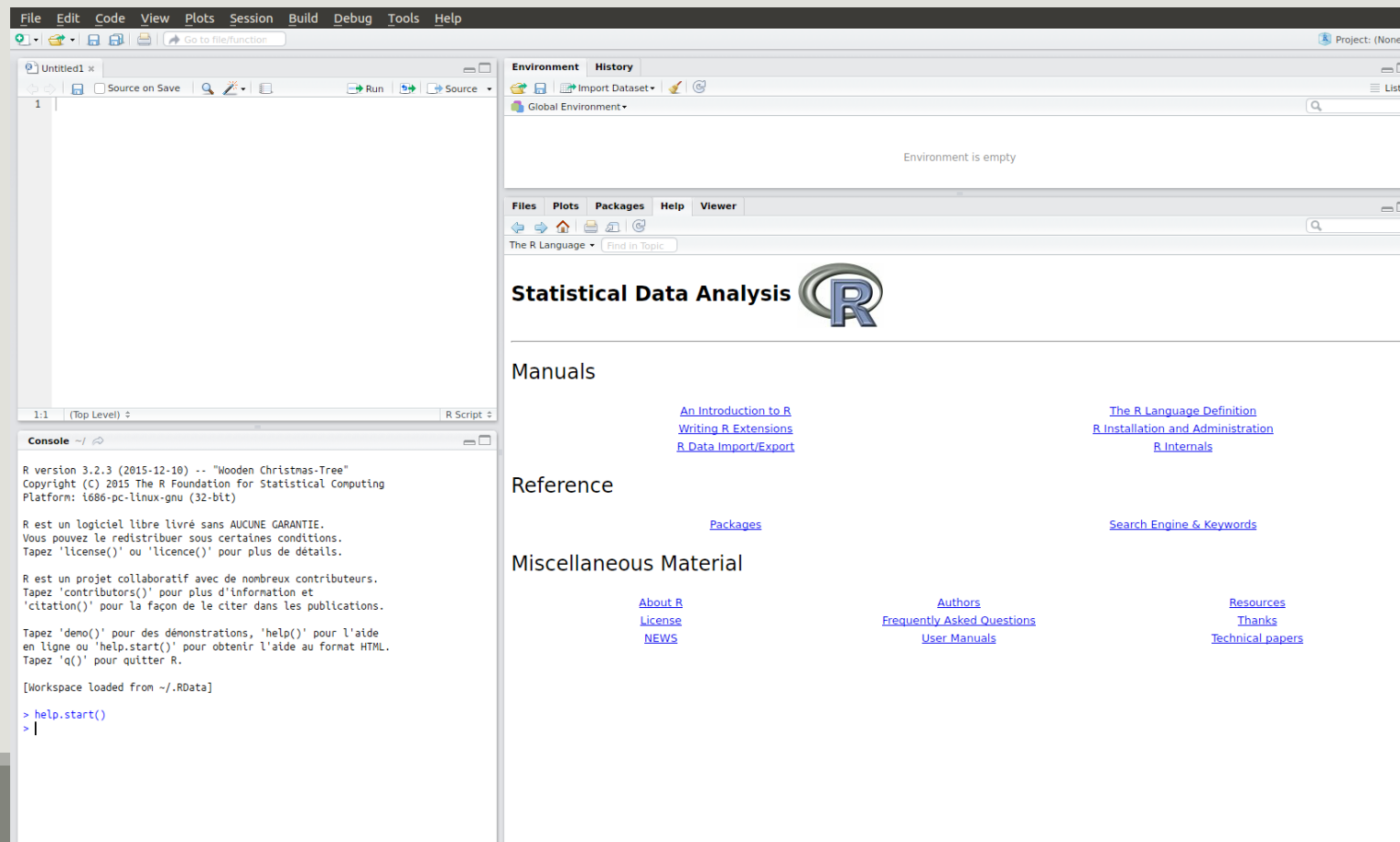
# R and RStudio

---

- 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 panel (script, console, workspace, graphics) :  
<https://www.rstudio.com/>



# 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 R-studio, 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

---

## Getting help

- Widely used packages include detailed help files for the functions they provide.
- `help("function name")` leads to the help page of function name

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

# R and RStudio

---

## Installing and loading packages

- 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("ggplot2")
```

- Loading packages is equally easy:

```
library(ggplot2)
```

- Most packages must be **loaded at each new session** (see the "Packages" tab in R-studio)



# R and Rstudio

---

## Variable assignment

- You can save the value of some R code using one of two ways:
  - the "arrow operator": `<-` (or more rarely `->`)
  - The "equal" sign: `=` (we recommend `<-` over `=`)
- The syntax is simple: `variable_name <- value`.

```
a <- 2*4 ## or a = 2*4 (the # sign signals a comment)
```

- 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 not impact the original object:

```
b <- a; b <- 8
```

```
a; b
```

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

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

# R and RStudio

---

## 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
```

# 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

---

There are several basic modes in R. The most common ones are illustrated below.

| Numeric  | Character  | Logical   |
|--|--|---|
| <pre>x &lt;- 1 class(x) ## [1] "numeric"</pre> | <pre>x &lt;- "hello" class(x) ## [1] "character"</pre> | <pre>x &lt;- TRUE class(x) ## [1] "logical"</pre> |

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

# Data/Variable : conversion

---

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

|  |   |   |
|--|---|---|
| <pre>as.numeric("5")<br/>## [1] 5<br/>as.logical(0.0)<br/>## [1] FALSE</pre> | <pre>as.numeric(TRUE)<br/>## [1] 1<br/>as.character(TRUE)<br/>## [1] "TRUE"</pre> | <pre>as.numeric("5.56")<br/>## [1] 5.56<br/>as.logical(2)<br/>## [1] TRUE</pre> |
|--|---|---|

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

```
as.numeric("INRA")  
## Warning: NAs introduits lors de la conversion automatique  
## [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                        | ## Character                     | ## Logical                        |
|-----------------------------------|----------------------------------|-----------------------------------|
| <code>as.numeric(2/3)</code>      | <code>as.character(2/3)</code>   | <code>as.logical(2/3)</code>      |
| <code>as.numeric(5.67)</code>     | <code>as.character(5.67)</code>  | <code>as.logical(0)</code>        |
| <code>as.numeric(FALSE)</code>    | <code>as.character(FALSE)</code> | <code>as.logical("45")</code>     |
| <code>as.numeric(TRUE)</code>     | <code>as.character(TRUE)</code>  | <code>as.logical("MaIAGE")</code> |
| <code>as.numeric("5.67")</code>   | <code>as.character(5)</code>     |                                   |
| <code>as.numeric("MaIAGE")</code> | <code>as.character(5+7)</code>   |                                   |

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

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



# Data/Variable : length

---

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

```
a <- 2
```

```
length(a)
```

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

```
a
```

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

- In the previous example, `a` is a vector of length 1, with a single element
- Hence the mysterious `[1]` in the output of `a`

# Data/Variable : special value

---

There are two important special values in R :

- **NA** stands for Not Available and is a code for missing data.
- **NULL** is the R code for a null object. It has length 0.

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

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

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

```
x <- NULL; length(x); is.null(x) ## NULL
```

```
## [1] 0
```

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

# Data/Variable : structure

---

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

- vector (1D array)
- factor
- matrix (2D array)
- list
- 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:

|   |  |
|---|--|
| <pre>x[1] ## first element<br/>## [1] 2</pre> | <pre>x[c(3, 5)] ## third and fifth elements<br/>## [1] 8 0</pre> |
|---|--|

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

|   |  |
|---|--|
| <pre>c(3.4, 2, TRUE)<br/>## [1] 3.4 2.0 1.0</pre> | <pre>c(3.4, "MaIAGE", TRUE)<br/>## [1] "3.4" "MaIAGE" "TRUE"</pre> |
|---|--|

# Data/Variable : vector

---

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

```
x <- c("first" = 1, "second" = 4, "third" = 9)
```

```
x
```

```
## first second third
```

```
## 1 4 9
```

|                                       |   |
|---------------------------------------|---|
| <pre>x[1]<br/>## first<br/>## 1</pre> | <pre>x["first"]<br/>## first<br/>## 1</pre> |
|---------------------------------------|---|

# Data/Variable : vector

---

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

```
x <- c(1, 4, 9)
```

```
x
```

```
## [1] 1 4 9
```

```
names(x) <- c("first", "second", "third")
```

```
x
```

```
## first second third
```

```
## 1 4 9
```

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

- Indexing

```
x <- c("O", "L", "H", "L", "E")  
x[c(3, 5, 2, 4, 1)] ## following Marcel Duchamp
```

- Type conversion

```
y <- c(4.5, 3, TRUE, "0.5")  
class(y[3])  
y[3]
```

- Conversion and indexing

```
z <- as.logical(c(4.5, 3, FALSE)) ## converts the whole vector  
z  
z[c(3, 1, 1)]
```



# Data/Variable : matrix

---

Matrices are essentially 2-D vectors: all elements must have the same mode. Indexing works the same way than 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

---

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

```
x[ , c(1, 4, 6)]
```

```
x[c(1, 3), ]
```

```
x[c(1, 3), c(1, 4, 6)]
```

# Data/Variable : factor

---

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

```
x <- factor(c("a", "a", "b", "a", "c"))
```

```
class(x)
```

```
## [1] "factor"
```

Levels can be accessed with levels

```
levels(x)
```

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

Internally, R treats `x` as a integer3 vector and associates each level to a value: here 1 = "a", 2 = "b" and 3 = "c" (alphabetical order by default) so that `x = c(1, 1, 2, 1, 3)`.

# Data/Variable : factor

---

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

```
y <- factor(x, levels = c("b", "a", "c"))
```

```
levels(y)
```

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

Finally since factors are internally coded as integer vectors, conversions can be surprising:

```
as.numeric(x) ## 1="a", 2="b"
```

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

```
as.character(x)
```

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

```
as.integer(y) ## 1="b", 2="a"
```

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

```
as.character(y)
```

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

# Data/Variable : factor

---

Compare the two different codes and try to guess the results. Check with the console.

```
x <- c("a" = 1, "b" = 2, "c" = 3)
y <- c("a", "b", "c")
x[y]
z1 <- factor(y, levels = c("a", "b", "c"))
z2 <- factor(y, levels = c("b", "a", "c"))
z1
x[z1]
x[as.character(z1)]
z2
x[z2]
x[as.character(z2)]
```

Did you guess right? If not, remember that factors are coded as integer vectors and try to guess the representation of z1 and z2 as numeric vectors.

# Data/Variable : data.frame

---

A data.frame is a table-like structure (created with 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"))
```

```
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 are names and can be 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

---

- **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

---

- **position** index elements by position in a `vector/factor` (`x[i]`) or positions (row, column) in a `matrix/data.frame` (`x[i, j]`)
- **name**: index elements by name in a `vector/factor` (`x["first"]`) or positions (row, column) in a `matrix/data.frame` (`x["row", "column"]`)
- **logical index**: use a **logical mask** index the same size as `x` that specifies which elements to keep (`x[index]`)
- **name 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: `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 <- 1:5
```

```
z <- (x < 3); z ## the first command returns a logical vector
```

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

```
z <- (x < 4) & (x > 1); z ## logical AND
```

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

```
z <- (x < 2) | (x > 4); 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 <- (x < 4)
```

```
x[z]
```

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

```
## or equivalently
```

```
x[x < 4]
```

```
## [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   |
|------------------------|---|
| <code>file</code>      | File name, or complete path to file (can be an URL)                         |
| <code>header</code>    | First line = variable names? ( <b>FALSE</b> by default)                     |
| <code>sep</code>       | Field separator character ( <b>white character</b> by default)              |
| <code>dec</code>       | Character used for decimal points ( <b>"."</b> by default)                  |
| <code>na.string</code> | Character vector of strings to be interpreted as NA ( <b>NA</b> by default) |
| <code>row.names</code> | 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 = "nt")
```

To save general objects as R-readable objects (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.

# 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](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

---

A ggplot is composed of :

- **data**: must be stored as a `data.frame`
- **aesthetics**: Visual characters that represent the data (position, size, color, ll, etc.)
- **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.)
- **coord**: 2D coordinate systems used to represent the data (cartesian, polar, etc.)
- **stat**: data-smoothing, statistical transformation used to summarize the data
- **facets**: a way to split the data into subsets (e.g. male only/female only) and represent the data as small multiple plots

# 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 : build a plot

---

The ggplot function is used to build the plot layer by layer. The general syntax is

```
## p stands for plot
```

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

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
```

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

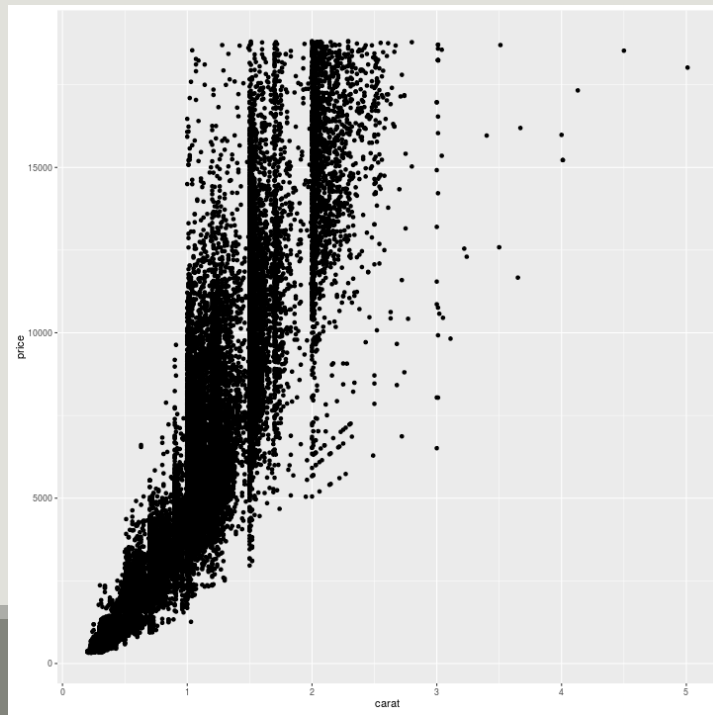
```
names(diamonds) ## documented properties
```

```
## [1] "carat" "cut" "color" "clarity" "depth" "table" ## [8] "x"  
"y" "z"
```

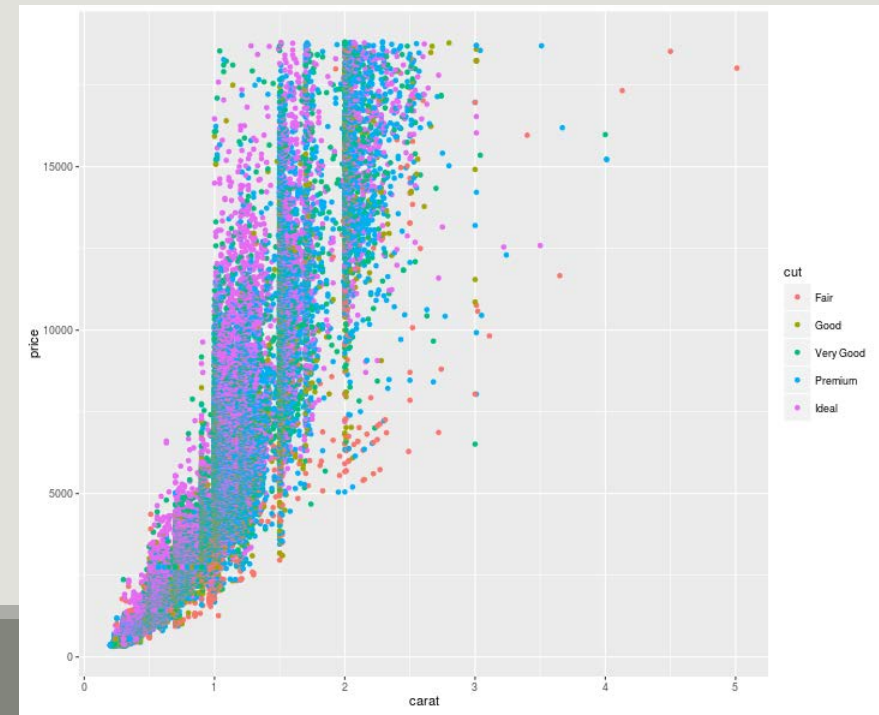
# ggplot2 : build a plot

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

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



```
## Add a layer to represent data as point,  
colored by cut  
p2 <- p + geom_point(aes(color = cut))  
plot(p2)
```



# ggplot2: build a plot

---

- The first command 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`). The aesthetics are extracted from global aesthetics `aes(x = carat, y = price)`.
- The variant `aes(color = cut)` adds a new local aesthetic for the point layer. *cut* value is mapped to the color of the points and both a legend and a color scale are automatically constructed.

# ggplot2: build a plot, aesthetics

---

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

- `shape`
- `size`
- `alpha` (transparency)
- `fill`

The value of each aesthetic can be either

- **identical for all** observation: the argument must be given **outside** of `aes` (e.g. `geom_point(color = "black")`)
- **mapped to a variable** value (here `cut`): the argument must be given **inside** of `aes` (e.g. `geom_point(aes(color = cut))`)



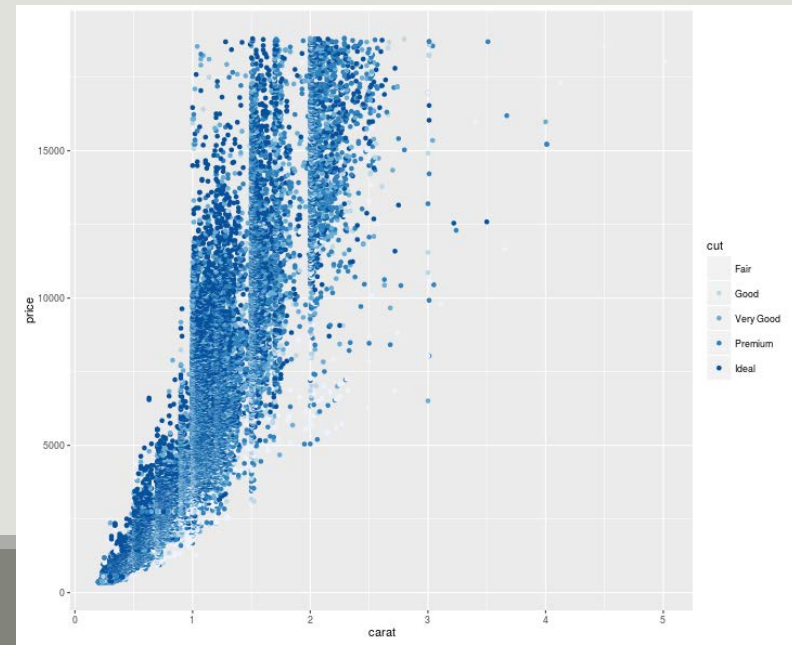
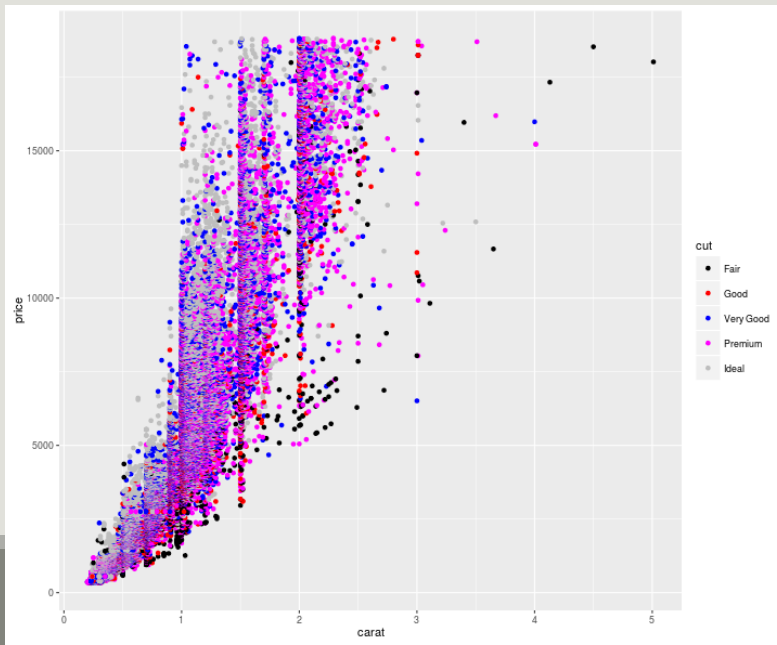
# ggplot2: build a plot, aesthetics

*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")
```

```
## Manual color scale
p2.1 <- p2 +
scale_color_manual(values = palette)
plot(p2.1)
```

```
## Use built-in color palette
p2.2 <- p2 + scale_color_brewer()
plot(p2.2)
```



# 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**) or factor, logical (**discrete**)
  - every scale is build in the following way they all begin with **scale\_**
  - continue with the aesthetic name (**linetype, fill, color**)
  - and end with the name of the scale (**manual, discrete, brewer**)

# ggplot2: build a plot, aesthetics

---

About geom:

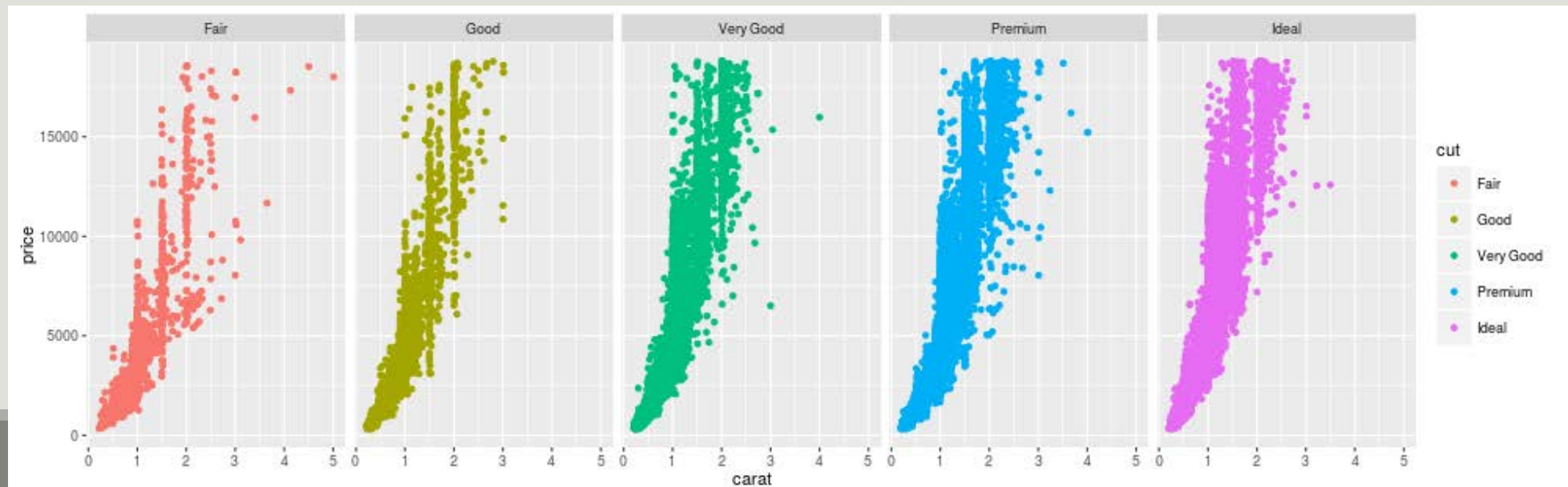
- Here we used `geom_point` to represent data as points. We could have used other geometric representations of the data:
  - `geom_point`
  - `geom_line`
  - `geom_bar`
  - `geom_density`
  - `geom_boxplot`
  - `geom_histogram`
- Each geometry expects and accepts different aesthetics (e.g `linetype`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

# ggplot2: build a plot, facetting

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)

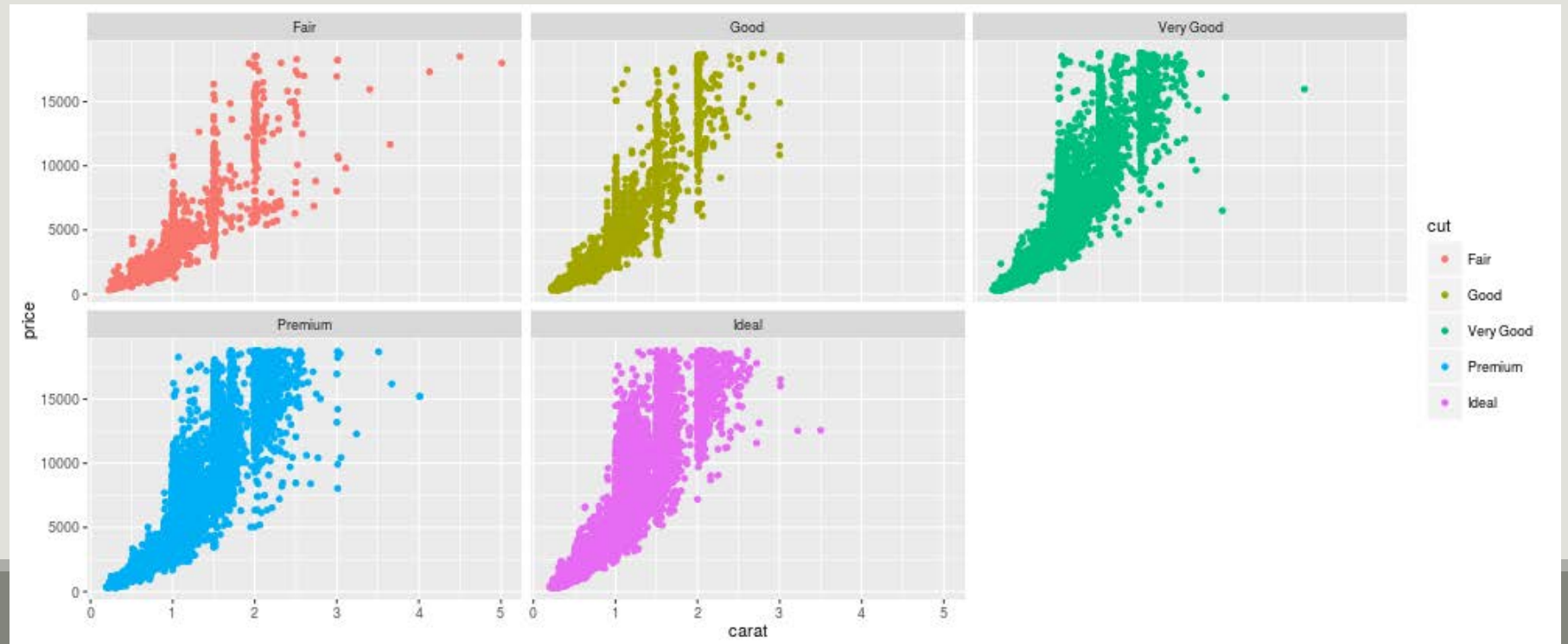
```
## facet along cut, only points from a given cut appear in a facet  
p3 <- p2 + facet_grid(~ cut)  
plot(p3)
```



# ggplot2: build a plot, facetting

Compare facet wrap and facet grid when using only one variable for facetting: facets are organized differently

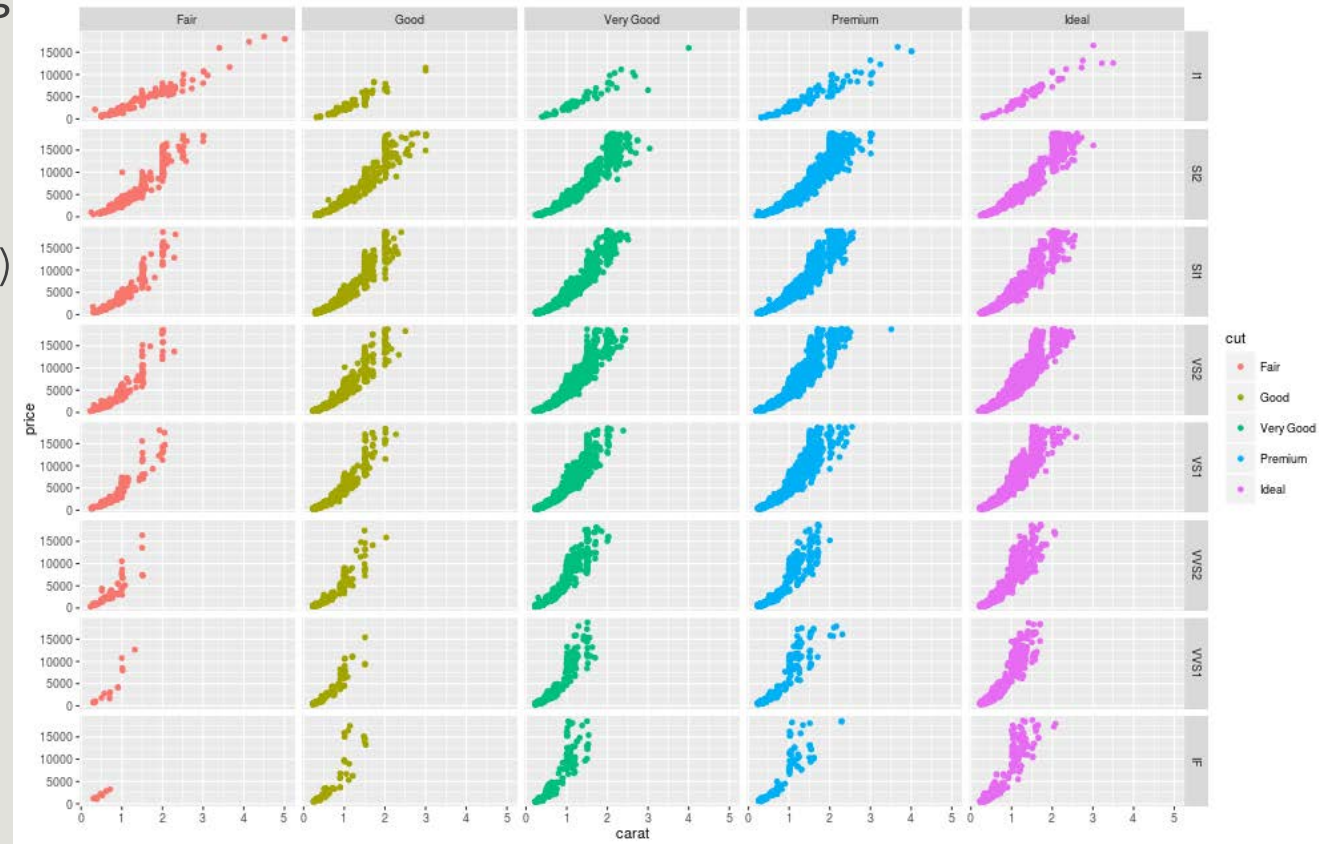
```
## facet along cut  
p3 <- p2 + facet_wrap(~ cut)  
plot(p3)
```



# ggplot2: build a plot, facetting

facet grid is most useful when splitting the data along two factors

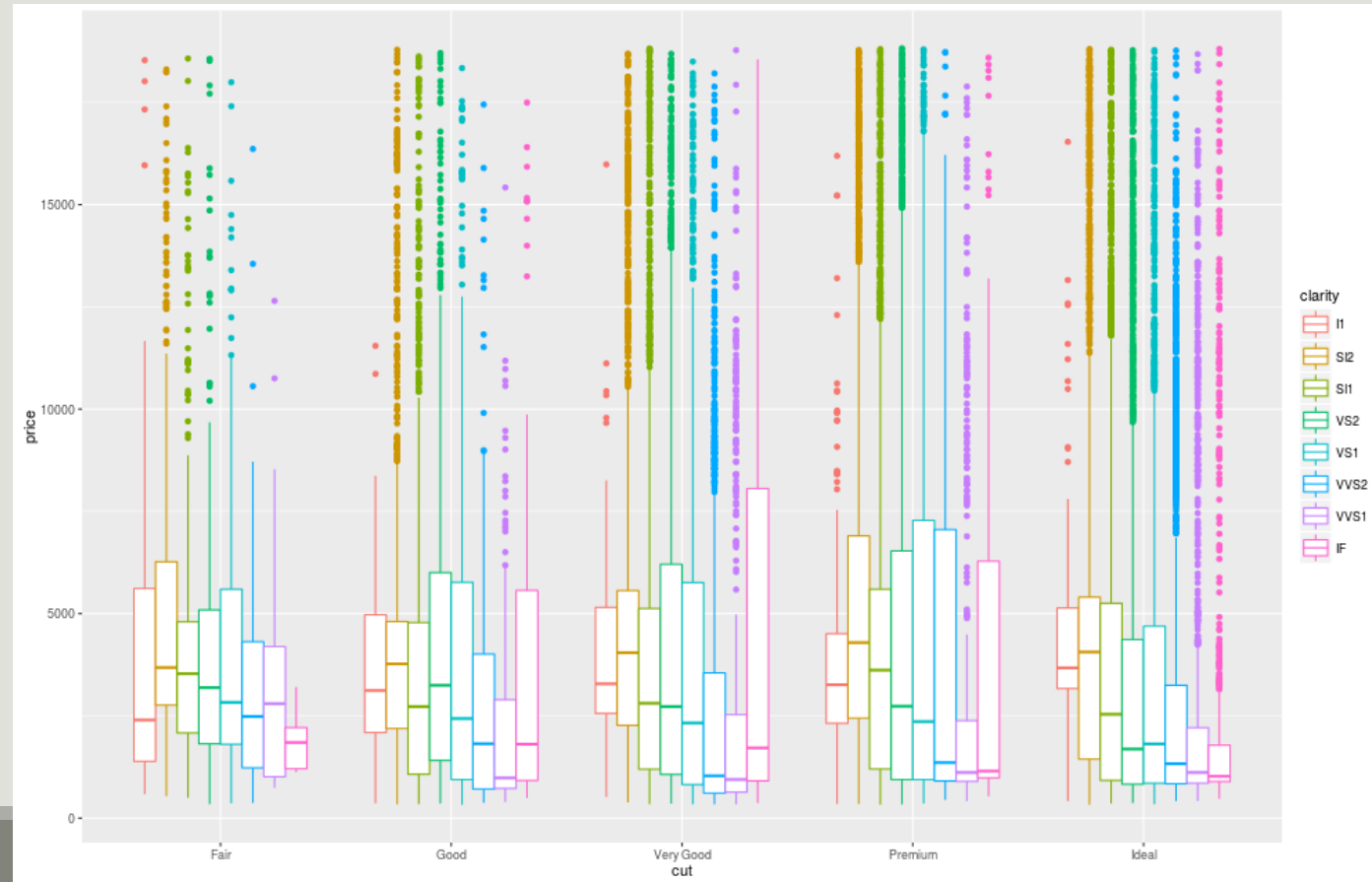
```
## facet along clarity(rows) *  
cut(column)  
p3 <- p2 + facet_grid(clarity ~ cut)  
plot(p3)
```



# ggplot2: build a plot, facetting

Sometimes, facetting wastes spaces. Imagine we want to compare the distribution (using boxplot) of prices (y) for different cuts (x) and highlight (with color) the differences between different clarities

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



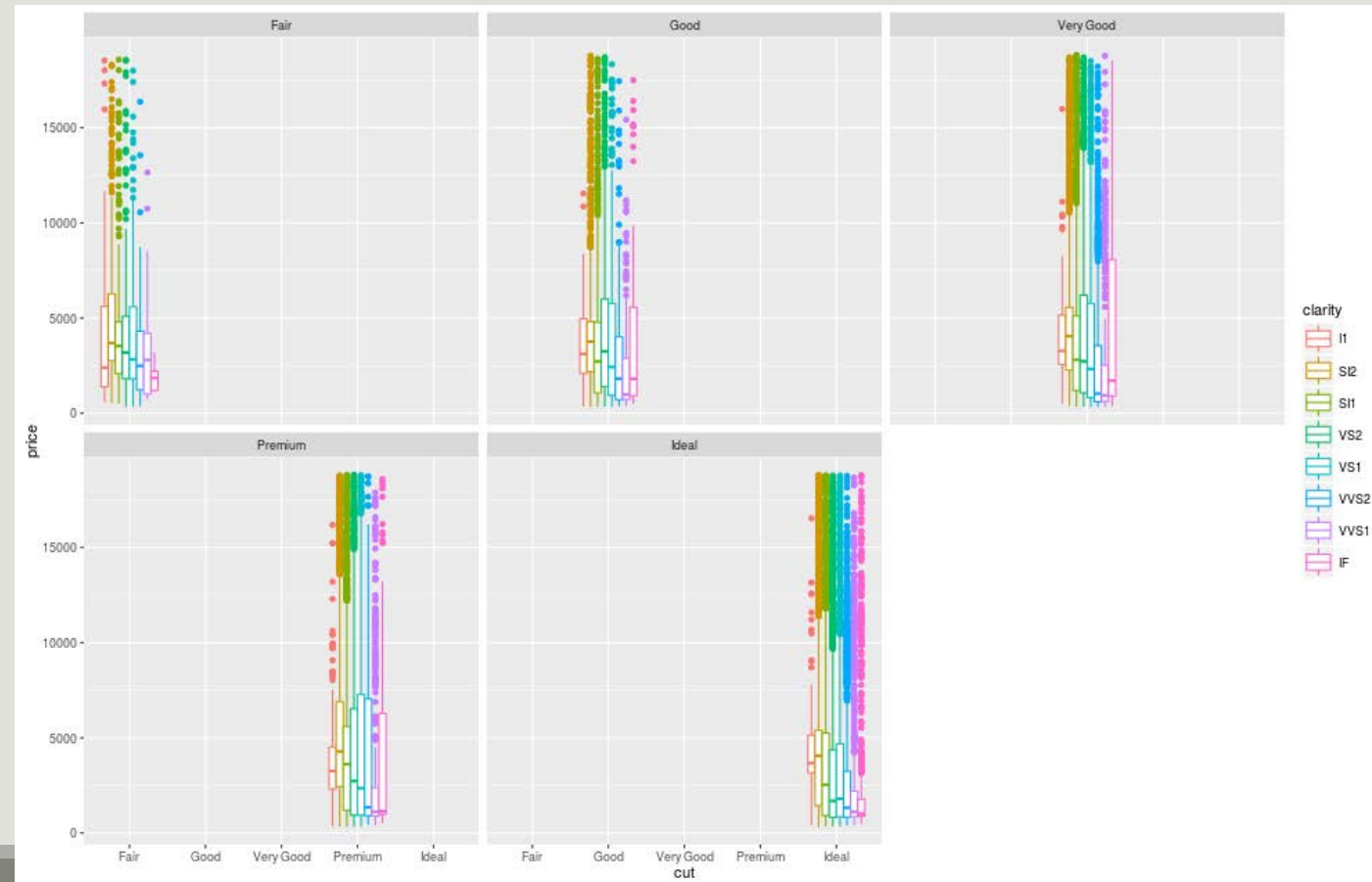


# ggplot2: build a plot, facetting

We may want to facet by cut to make the plot easier to read

```
p1 <- p + facet_wrap(~cut)
plot(p1)
```

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





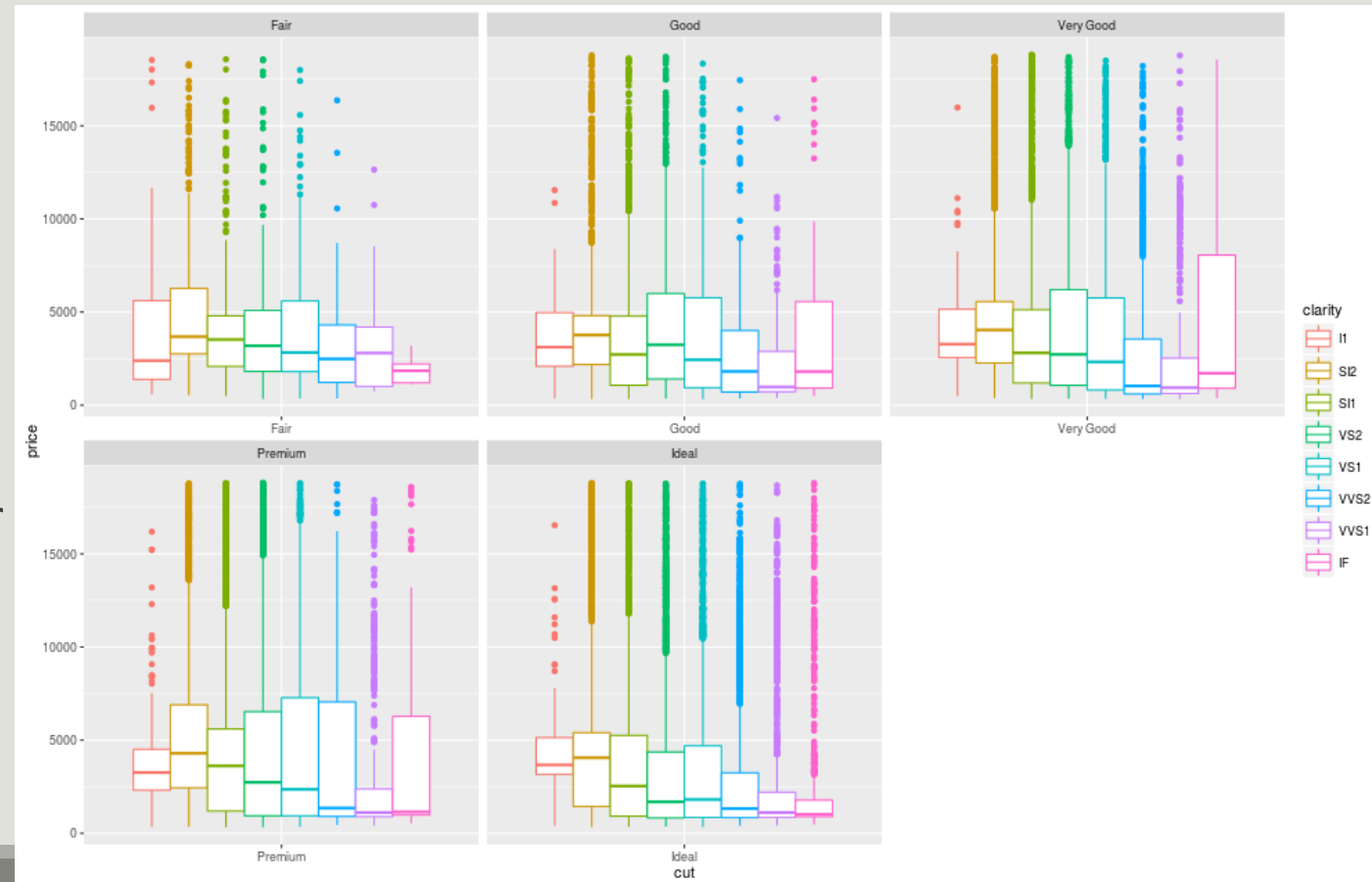
# ggplot2: build a plot, facetting

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

```
p2 <- p + facet_wrap(~cut,  
scales = "free_x")  
  
plot(p2)
```

scales = "free\_y" would lead to one y-scale per facet

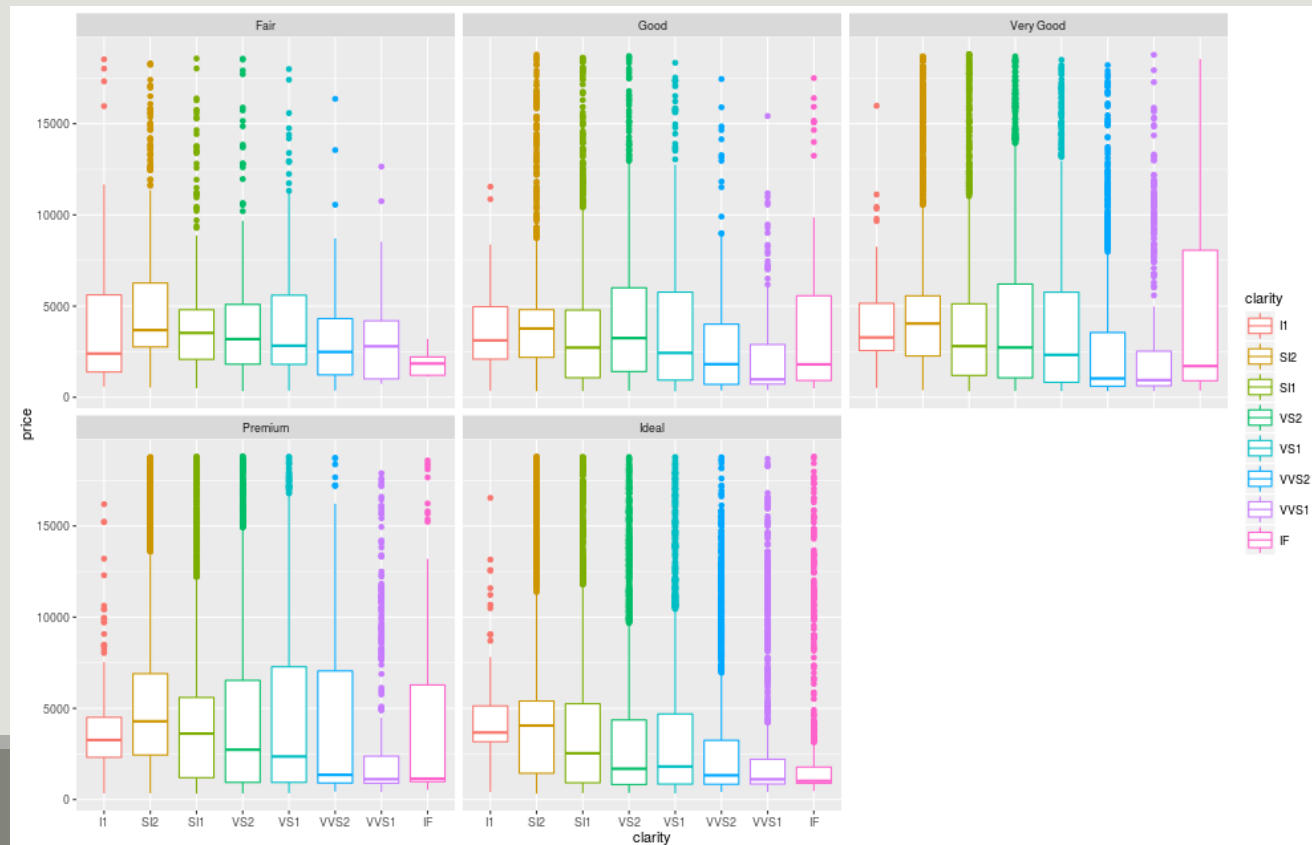
scales = "free" to one y-scale and one x-scale per facet



# ggplot2: build a plot, facetting

Finally, note that the same graph can be obtained in many different ways.

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

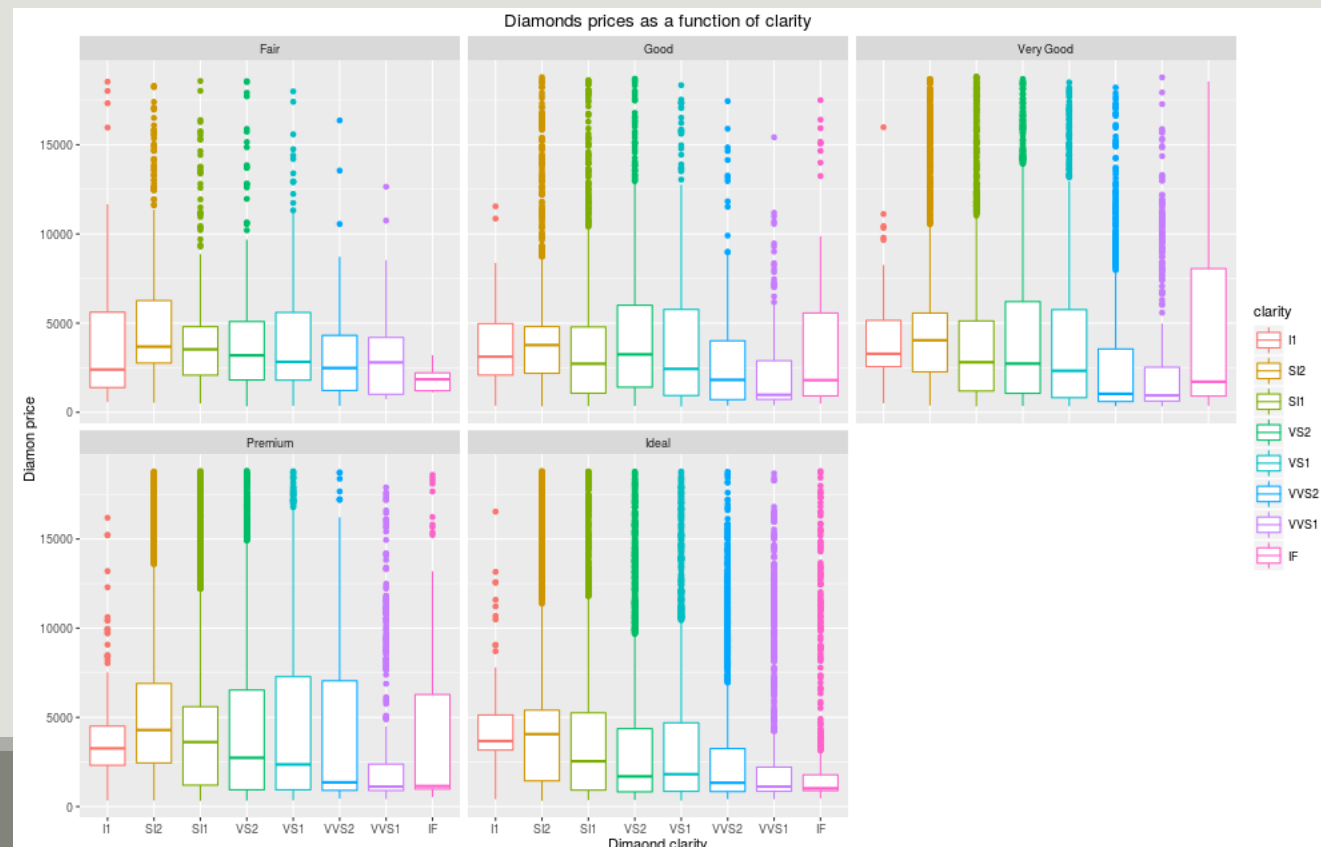


# ggplot2: build a plot, title and labels

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

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

```
plot(p)
```



# 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

---

- <http://had.co.nz/ggplot2/>
- <http://groups.google.com/group/ggplot2>
- <http://cran.r-project.org/web/packages/ggplot2/index.html>
- Wickman, H. 2009 { ggplot2. Elegant graphics for data analysis. Springer, 212p.