HBioinfo12 R Intorduction

Anastasios Chanalaris

30/09/2019

Contents

What is R?

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

Why R?

One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

There are thousands of packages available (almost 15000 at CRAN) (https://cran.r-project.org/web/packages/available_packages_by_name.html). Chances are that someone has a package that does what you want.

Also there is a dedicated repository of biological data handling packages at bioconductor (1741 at alatest version) (https://www.bioconductor.org/packages/release/BiocViews.html#____Software)

How to install R

Go to https://cran.r-project.org and follow the links and instructions for your OS.

IDE for R

R is normally run through a terminal; either interactively, or through scripts. This is however, cumbersome as ir needs keeping track of a large number of windows and the ability to memorise a large amount of commands. That is where an Integrated Development Environment comes handy, as it can provide at a glance all the active windows and help with command syntax, through a GUI environment. The GUI that comes with R is cumbersome and not very sophisticated. There are numerous IDEs available for R, but the best currently is *RStudio*. To install RStudio go to: https://rstudio.com/products/rstudio/download/ and install the version for your OS. There is also a free version for a server if you wish to install it as a server and access it through your browser.

The RStudio environment

R studio presents with three windows. Top left is the editing environment, where you can interact and build your scripts, notebooks, html files etc. You can pass the commands you create at the editing area to the be executed at the output terminal area (console) to the bottom left with *CTRL-ENTER*. You can also

enter commands directly to the console area and execute them by pressing *ENTER*. The Top right panel is the environment area, where you can see your objects in the environment, history of commands and some graphical outputs like you presentations and markdown documents.

The bottom right panel is another assisting area where you can view your directory, packages available, the help display area and most importantly your plots and graphical output.

The layout of the panels can be customised to your liking through Tools/Global Options/Pane Layout.

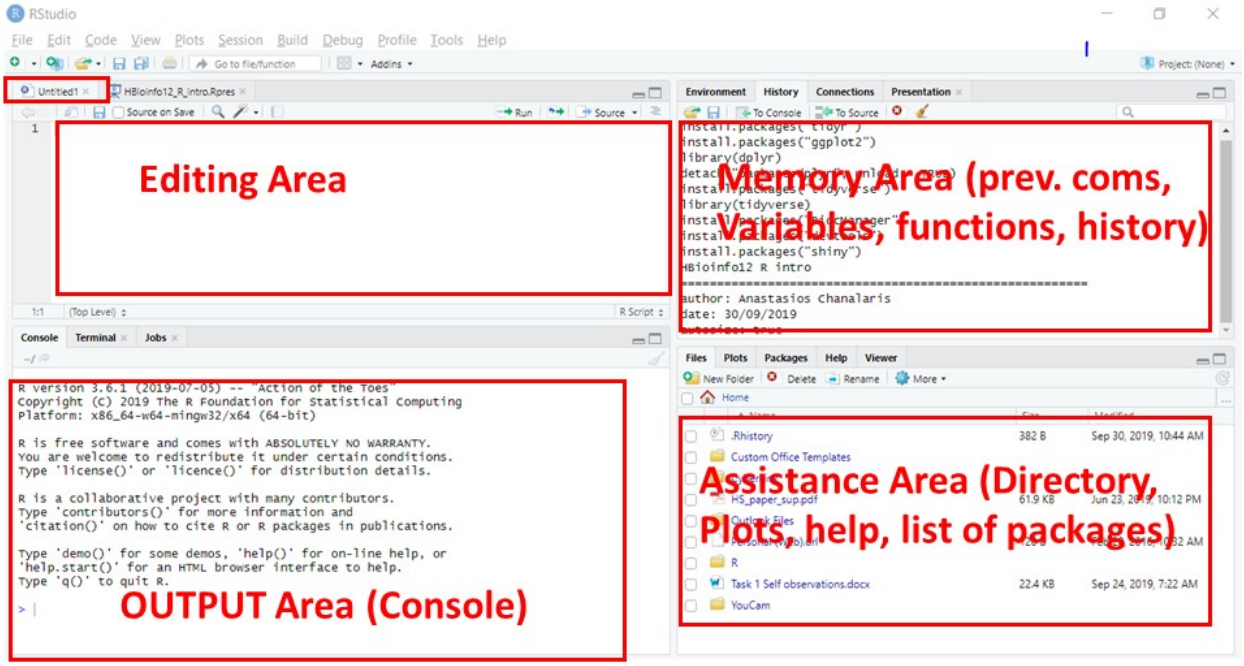


Figure 1. The default pane layout in RStudio. The window is split in 4 areas.

Getting help in R and RStudio

Help is available in HTML format. Getting info on a specific function (in this case the *plot* function):

```
help(plot)
```

or, alternatively

```
?plot
```

For a feature specified by special characters, the argument must be enclosed in quotes. Also necessary for a few words with syntactic meaning (like *if*, *for* and *function*).

```
help("if")
```

```
help("[[")
```

One can get examples of usage of a particular topic by using the command *example()*. For *plot*:

```
example("plot")
```

Help in R

```
library()#packages installed
search()#which packages are loaded
help(package="base")#all the functions available in the *base* package
help("plot")#to see how to use *plot*
RSiteSearch("plot")#Online help about *plot*
apropos("plot")#all functions that contain *plot*
example("plot")#example of using *plot*
demo(graphics)#demonstration on the *graphics* package
data()#what datasets are available
```

R as a calculator

- addition

```
5 + 2
```

```
[1] 7
```

- subtraction

```
5 - 2
```

```
[1] 3
```

- multiplication

```
5 * 2
```

```
[1] 10
```

- division

```
5 / 2
```

```
[1] 2.5
```

- Remainder

```
5 %% 2
```

```
[1] 1
```

- Quotient

```
5 %/% 2
```

```
[1] 2
```

- Power

```
5 ^ 2
```

```
[1] 25
```

```
# or
5 ** 2
```

```
[1] 25
```

R as a logic operator

- Greater than

```
5 > 2
```

```
[1] TRUE
```

- Greater than

```
2 > 5
```

```
[1] FALSE
```

- less than

```
5 < 2
```

```
[1] FALSE
```

- less than

```
2 < 5
```

```
[1] TRUE
```

- equal to

```
5 == 2
```

```
[1] FALSE
```

- not equal to

```
2 != 5
```

```
[1] TRUE
```

- equal to

```
5 == 5
```

```
[1] TRUE
```

- not equal to

```
2 != 2
```

```
[1] FALSE
```

Storing Values in a variable

The output of a command is directed to the screen, and lost. It can be recalled by

```
{.Last.value}
```

but if the answer needs to be used later on it is best if we could store it. That is done by storing it to a variable. Using the assignment symbols <- and = .

- Storing a single values (assign symbol: = or ->)

```
A = 647
```

- storing a calculation to a variable

```
5 *2-(6/4) -> B
```

- Storing a vector

```
C<-c(5,3,7)
```

- Storing a dataframe

```
D=data.frame(Name=c("Jim", "George", "Lisa"), Age= c(31,43,22))
```

- Storing the operation on a number of variables

```
E = A + B
```

Storing strings

We can also store string (alphanumerical series of characters). This is done in a similar manner as with numerical values, but we need to just enclosed the characters in " or '. i.e.:

```
F= "F"
U = "U"
class(F)
```

```
[1] "character"
```

The command *class()* returns the type of the variable. Is it a character, numeric, etc.

```
Nuc<-c("A", "T", "C", "G")
class(Nuc)
```

```
[1] "character"
```

```
my_seq<-sample(Nuc, 50, replace = TRUE) #sample 50 random nucleotides from Nuc and names them my_seq
```

```
head(my_seq, 5)
```

The *head()* command allows us to look at the top of a stored object. The *tail()* displays the end of the stored object.

```
[1] "G" "T" "C" "T" "T"
```

```
tail(my_seq,5)
```

```
[1] "G" "G" "T" "G" "A"
```

The *paste()* function allows to connect, or seperate all elements in an object. Similar funcion to *cat()*.

```
paste(my_seq, sep="", collapse="")->testDNA #from myseq concatenate all the elements
testDNA
```

```
[1] "GTCTTCCGCATCAGTATCGGGGGATACCGCGGGCGCGCTCACCGTGGTGA"
```

String manipulation

```
grep("ATG", testDNA, value = TRUE) # Is there the ATG pattern on the seq?
```

```
character(0)
```

```
gregexpr("ATG", testDNA) # show me the position of ATG
```

```
[[1]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

```
DNA_size=as.numeric(nchar(testDNA))
substring(testDNA, gregexpr("ATG", testDNA, DNA_size))->orf.testDNA #store the sequence from ATG
#to the end of the seq on a variable
print(orf.testDNA)
```

Further string manipulation functions through base R:

Function	Description
is.character()	logic
as.character()	change class to character
readLines()	creates a character vector for each line from a txt file
cat()	cat(. . . , file = "“, sep =” “, fill = FALSE, labels = NULL, append = FALSE) concatenates several objects that like paste() are converted to class character. Outputs the text without””.
format()	for output options (how many digits, justify etc.)
nchar()	number of characters
tolower()	convert to lower case
toupper()	convert to upper case
casefold()	case folding
chartr()	character translation
abbreviate()	abbreviation
substring()	substrings of a character vector
substr()	substrings of a character vector

R, like perl python etc. makes use of regular expressions (regex) to fine tune the search for a pattern that fits our needs. To get more information on regex look at the figure 2.

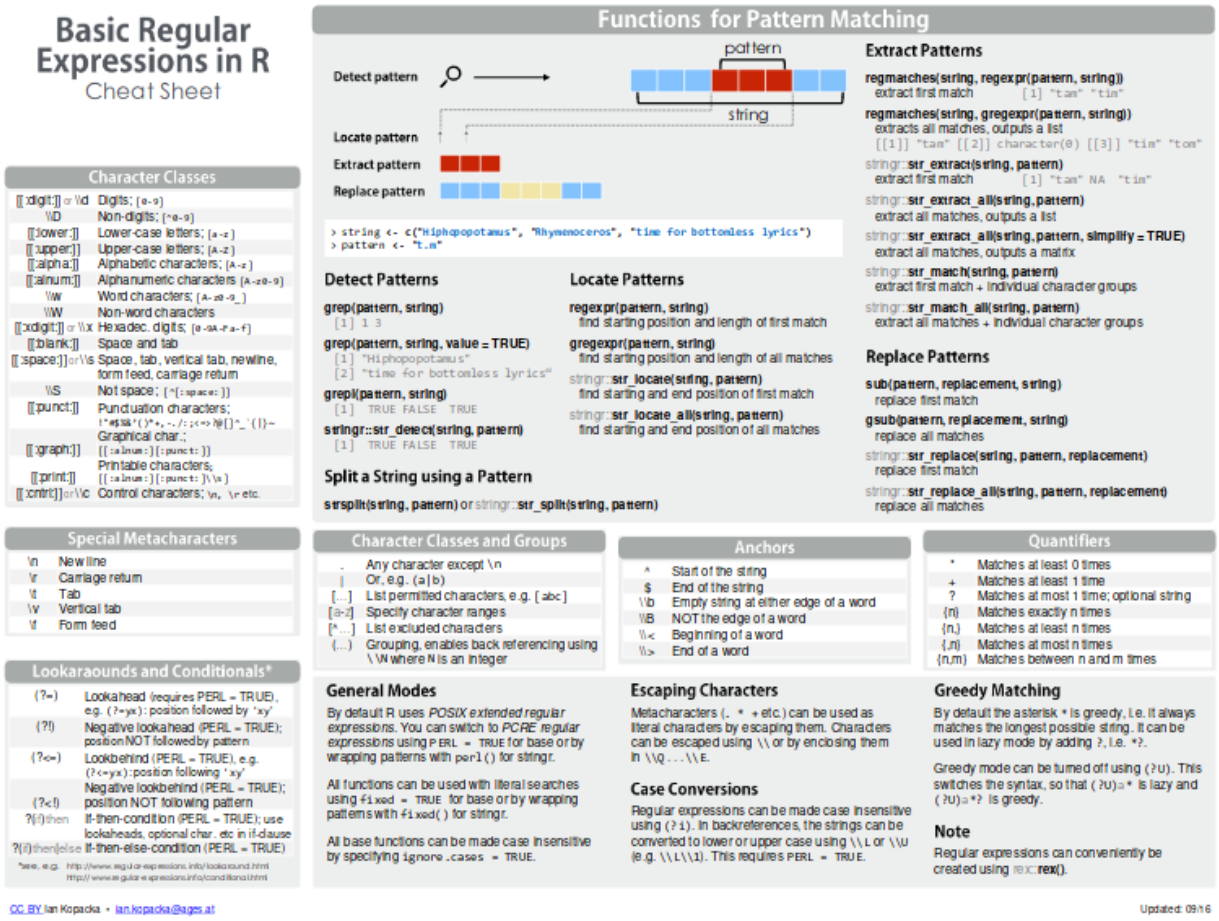


Figure 2. Regular Expressions Cheat sheet from https://rstudio.com/wp-content/uploads/2016/09/RegExCheatsheet.pdf

Recalling values

```
A
```

```
[1] 647
```

```
B
```

```
[1] 8.5
```

```
C
```

```
[1] 5 3 7
```

```
D
```

```
    Name Age
1    Jim  31
2 George  43
3   Lisa  22
```

```
E
```

```
[1] 655.5
```

Syntax

- R is an expression language.
- It is case sensitive (a != A)
- All alphanumeric symbols allowed (might be locale restricted)
- Names should start with a letter or "." . When "." at the start, second character must **NOT** be a digit.
- Commands consist of expressions or assignments. They are seperated by a ";" or a new line
- Elementary commands can be grouped together into one compound expression by "{}"
- Comments can be added anywhere and start with a "#". Everything to the end of that line is a comment.
- If a command is incomplete at the ed of a line, R changes prompt to "+" on second and subsequent lines and continues to read input, until the command is syntactically complete.

Function

Function is any expression that performs an action to an object. It can be found build in (base R), through packages, or one could build their own. A function is defined by an assignment:

```
my_func <- function(*arg_1, arg_1, ...*) *expression*
```

The *expression* is an R expression that uses the arguments to calculate a value, which is returned by the function. One can call the function by *my_func(expr_1,expr_2,...)*.

Example of function

Let's say we need a function for calling a two sample t test can be. We can do this by:

```
2samT <- function(y1,y2) {
  n1 <- length(y1); n2 <- length(y2)
  yb1 <-mean(y1); yb2 <- mean(y2)
  s1 <- var(y1); s2 <- var(y2)
  s <- ((n1-1) * s1 + (n2-1) * s2)/(n1+n2-2)
  tst <- (yb1 - yb2)/sqrt(s * (1/n1 + 1/n2))
  tst
}
```

to call the function:

```
X <- 2samT(data$treated, data$untreated)
```

Installing packages

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored is called the library. R comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used.

```
.libPaths() # get library location
library()   # see all packages installed
search()    # see packages currently loaded
```

To install packages use:

```
install.packages(pkgs, lib, repos =gerOption("repos"),...)
```

Let's install tidyverse:

```r
install.packages("tidyverse")
library(tidyverse) # load package
```

Batch Processing

You can run R non-interactively with input from infile and send output (stdout/stderr) to another file. Here are examples.

```
# on Linux
R CMD BATCH [options] my_script.R [outfile]

# on Microsoft Windows (adjust the path to R.exe as needed)
"C:\Program Files\R\R-2.13.1\bin\R.exe" CMD BATCH
   --vanilla --slave "c:\my projects\my_script.R"
```

Input/Output

Input

The *source( )* function runs a script in the current session. If the filename does not include a path, the file is taken from the current working directory.

```r
# input a script
source("myfile")
```

Output

The *sink()* function defines the direction of the output.

```r
# direct output to a file
sink("myfile", append=FALSE, split=FALSE)
# return output to the terminal
sink()
```

I/O Graphs

Graphs

sink() will not redirect graphic output. To redirect graphic output use one of the following functions. Use *dev.off()* to return output to the terminal.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Use a full path in the file name to save the graph outside of the current working directory.

```r
# example - output graph to jpeg file
jpeg("c:/mygraphs/myplot.jpg")
plot(x)
```

```
dev.off()
```

A simple session

The output from analyses can easily be saved and used as input to additional analyses.

```
# Example 1
lm(mpg~wt, data=mtcars) # Linear regression of mpg vs car weight from the mtcars data
```

Results are sent to the screen. Nothing is saved.

```
# Example 2
fit <- lm(mpg~wt, data=mtcars) # save results of regression to variable fit
```

No output is sent to the screen.

```
# Example 2 (continued...)
str(fit) # view the contents/structure of "fit"
```

fit is a list

```
# Example 2 (continued again)
# plot residuals by fitted values
plot(fit$residuals, fit$fitted.values)
```

To see what the *lm()* function returns, look at *help(lm)*.

The results can be fed into other functions.

```
# Example 2 (one last time, I promise)
# produce diagnostic plots
plot(fit)
# predict mpg from wt in a new set of data
predict(fit, mynewdata)
# get and save influence statistics
cook <- cooks.distance(fit)
```

Importing Data

Importing data into R is fairly simple. For Stata and Systat, use the foreign package. For SPSS and SAS I would recommend the Hmisc package for ease and functionality. Examples of importing data are provided below.

From A Comma Delimited Text File

```
# first row contains variable names, comma is separator
# assign the variable id to row names
# note the / instead of \ on mswindows systems

mydata <- read.table("c:/mydata.csv", header=TRUE,
   sep=",", row.names="id")
```

From Excel

One of the best ways to read an Excel file is to export it to a comma delimited file and import it using the method above. Alternatively you can use the xlsx package to access Excel files. The first row should contain variable/column names.

```
# read in the first worksheet from the workbook myexcel.xlsx
# first row contains variable names
library(xlsx)
mydata <- read.xlsx("c:/myexcel.xlsx", 1)

# read in the worksheet named mysheet
mydata <- read.xlsx("c:/myexcel.xlsx", sheetName = "mysheet")
```

From SPSS

```
# save SPSS dataset in trasport format
get file='c:\mydata.sav'.
export outfile='c:\mydata.por'.
# in R
library(Hmisc)
mydata <- spss.get("c:/mydata.por", use.value.labels=TRUE)
# last option converts value labels to R factors
```

From SAS

```
# save SAS dataset in trasport format
libname out xport 'c:/mydata.xpt';
data out.mydata;
set sasuser.mydata;
run;
# in R
library(Hmisc)
mydata <- sasxport.get("c:/mydata.xpt")
# character variables are converted to R factors
```

From Stata

```
# input Stata file
library(foreign)
mydata <- read.dta("c:/mydata.dta")
```

From systat

```
# input Systat file
library(foreign)
mydata <- read.systat("c:/mydata.dta")
```

Exporting Data

For SPSS, SAS and Stata, you will need to load the foreign packages. For Excel, you will need the xlsReadWrite package.

To A Tab Delimited Text File

```
write.table(mydata, "c:/mydata.txt", sep="\t")
```

To an Excel Spreadsheet

```
library(xlsx)
write.xlsx(mydata, "c:/mydata.xlsx")
```

To SPSS

```
# write out text datafile and
# an SPSS program to read it
library(foreign)
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sps",   package="SPSS")
```

To SAS

```
# write out text datafile and
# an SAS program to read it
library(foreign)
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sas",   package="SAS")
```

To Stata

```
# export data frame to Stata binary format
library(foreign)
write.dta(mydata, "c:/mydata.dta")
```

Appendix A

GRAPHICAL ANALYSIS IN R: A primer

Packages available for data visualisation in R

- graphics
- ggplot2
- lattice (multiple graphs)
- RGL (3D)
- plottly (interactive)
- many more

Customising graphs on the graphics package

Graphical parameters can be set using the *par()* command with a number of arguments for all plots; par arguments given before the graph:

Titles

- main = ""
- sub= ""
- xlab=""
- ylab=""

Colours

- col = "" or number (RGB, Hex, or index)
- col.axis axis
- col.lab labels
- col.main main title
- col.sub subtite
- fg foreground, sets axes, boxes and col to the same
- bg background

List of colours

From https://github.com/EarlGlynn/colorchart/wiki/Color-Chart-in-R

Text and symbols

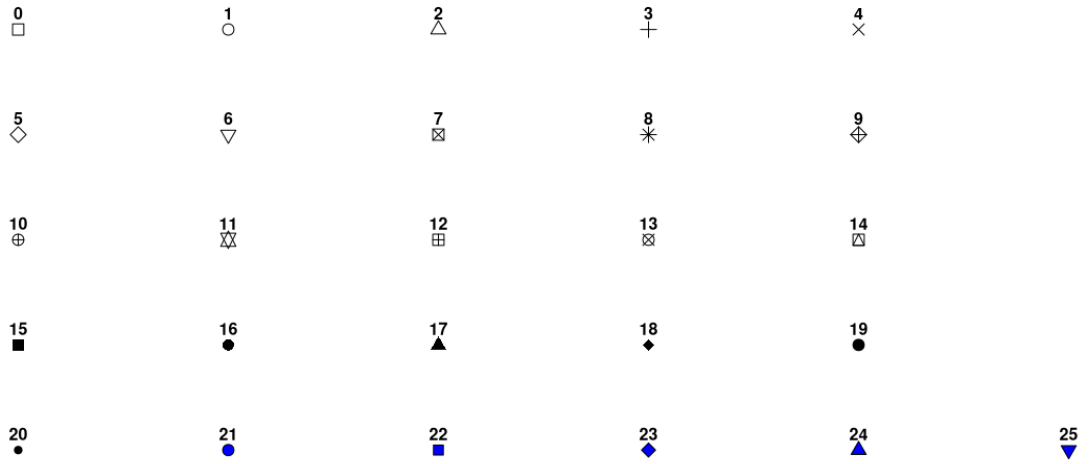


Figure 1: symbols in R

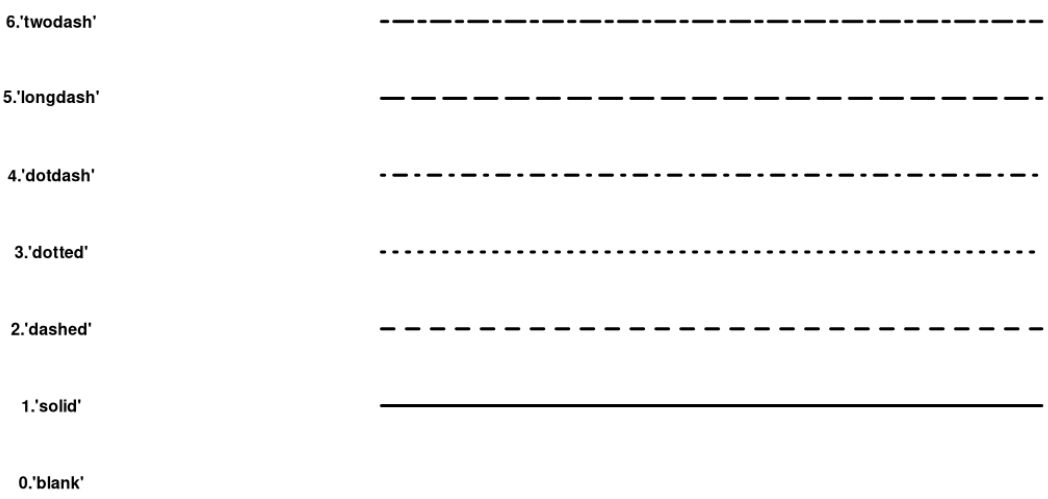


Figure 2: Line types

- cex (scaling factor for text and symbols relative to default (1 = default, 0.5 50% smaller))
- cex.axis = scale axis
- cex.lab = scale labels
- cex.main = scale main title
- cex.sub = scale subtitle

Symbols

pch= a number

Lines

Line type: lty = 0:6

Line width: lwd

Fonts

font = 0:5

1. plain
2. bold
3. italic
4. bold italic
5. symbol

- font.axis
- font.lab
- font.main
- font.sub
- ps font point size (~1/72 inch, text size = ps * cex)
- family ("serif", "sans", "mono", "symbol")

Margins and graph size

- mar vector c(bottom, left, top, right) in lines
- mai as mar, but in inches
- pin plot dimensions width, height in inches

Multiple graphs

You can control the number and size of graphs with - par() and - layout()

With par() mfrow=c(nrows,ncols) creates a matrix of nrows x ncols plots that are populated by row. mfcol=c(nrwos,ncols) populates by columns.

layout() is more versatile, it takes a matrix as an argument that indicates the location of the next N figures in the global plotting space. It is followed by a width and height arguments that allows you to graph plots with different sizes. For quick help see RStudio cheatsheets https://rstudio.com/resources/cheatsheets/

Prerequisites

- Import Data
  - Datasets in R
  - Web / Databases
  - Own data
- Familiarise with the data set
  - summary()
  - structure str()
  - missing values is.na(x)
  - clean up new.data <- na.omit(data)

Data wrangling (dplyr and tidyr)

You can obtain info on these valuable packages from https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf Both are included in the package tidyverse, excellent for data analysis.

Data Wrangling
with dplyr and tidyr
Cheat Sheet

RStudio

Tidy Data - A foundation for wrangling in R

In a tidy data set:

Each **variable** is saved in its own **column**

&

Each **observation** is saved in its own **row**

Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.

Syntax - Helpful conventions for wrangling

dplyr::**tbl_df(iris)**

Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

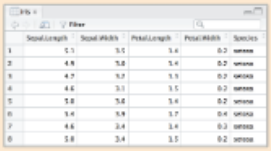
```
Source: local data frame [150 x 5]

   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: Petal.Width (dbl),
  Species (fctr)
```

dplyr::**glimpse(iris)**

Information dense summary of tbl data.

utils::**View(iris)**

View data set in spreadsheet-like display (note capital V).



dplyr::**%>%**

Passes object on left hand side as first argument (or . argument) of function on righthand side.

```
x %>% f(y)  is the same as f(x, y)
y %>% f(x, .., z)  is the same as f(x, y, z)
```

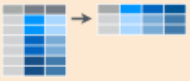
"Piping" with %>% makes code more readable, e.g.

```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

Reshaping Data - Change the layout of a data set

tidyr::**gather(cases, "year", "n", 2:4)**

Gather columns into rows.



tidyr::**spread(pollution, size, amount)**

Spread rows into columns.

tidyr::**separate(storms, date, c("y", "m", "d"))**

Separate one column into several.

tidyr::**unite(data, col, ..., sep)**

Unite several columns into one.

dplyr::**data_frame(a = 1:3, b = 4:6)**

Combine vectors into data frame (optimized).

dplyr::**arrange(mtcars, mpg)**

Order rows by values of a column (low to high).

dplyr::**arrange(mtcars, desc(mpg))**

Order rows by values of a column (high to low).

dplyr::**rename(tb, y = year)**

Rename the columns of a data frame.

Subset Observations (Rows)

dplyr::**filter(iris, Sepal.Length > 7)**

Extract rows that meet logical criteria.

dplyr::**distinct(iris)**

Remove duplicate rows.

dplyr::**sample_frac(iris, 0.5, replace = TRUE)**

Randomly select fraction of rows.

dplyr::**sample_n(iris, 10, replace = TRUE)**

Randomly select n rows.

dplyr::**slice(iris, 10:15)**

Select rows by position.

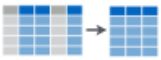
dplyr::**top_n(storms, 2, date)**

Select and order top n entries (by group if grouped data).

Logic in R - ?Comparison, ?base::Logic

<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&, |, !, xor, any, all	Boolean operators

Subset Variables (Columns)



dplyr::**select(iris, Sepal.Width, Petal.Length, Species)**

Select columns by name or helper function.

Helper functions for select - ?select

select(iris, **contains(".")**)
Select columns whose name contains a character string.

select(iris, **ends_with("Length")**)
Select columns whose name ends with a character string.

select(iris, **everything()**)
Select every column.

select(iris, **matches(".t.")**)
Select columns whose name matches a regular expression.

select(iris, **num_range("x", 1:5)**)
Select columns named x1, x2, x3, x4, x5.

select(iris, **one_of(c("Species", "Genus"))**)
Select columns whose names are in a group of names.

select(iris, **starts_with("Sepal")**)
Select columns whose name starts with a character string.

select(iris, **Sepal.Length:Petal.Width**)
Select all columns between Sepal.Length and Petal.Width (inclusive).

select(iris, **-Species**)
Select all columns except Species.

devtools::install_github("rstudio/EDAWR") for data sets

Learn more with browseVignettes(package = c("dplyr", "tidyr")) • dplyr 0.4.0 • tidyr 0.2.0 • Updated: 1/15

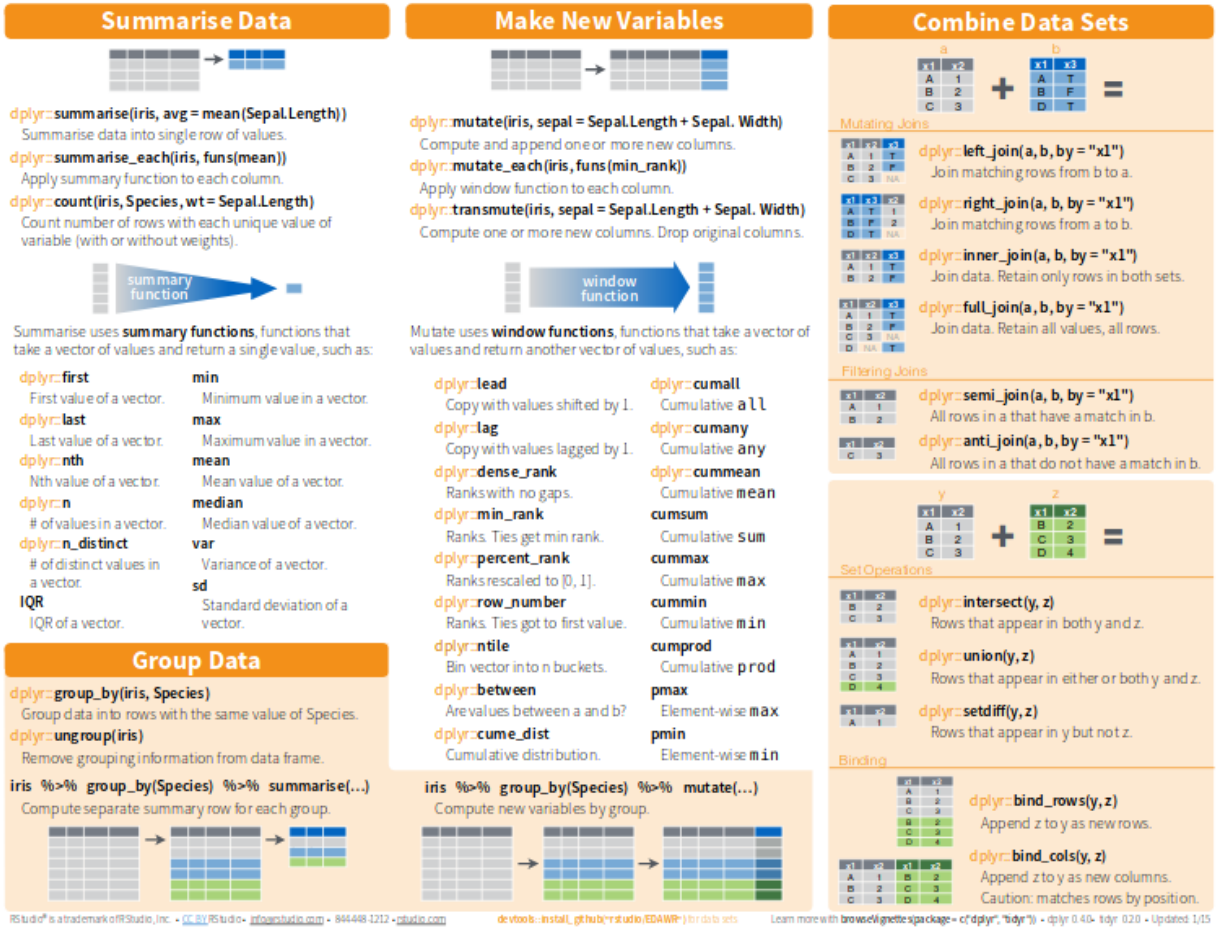


Figure 3. dplyr cheat sheets

Deciding on plots

Data can be:

1. Numerical (Discrete/Continuous)
2. Categorical (Ordinal/Nominal)
3. Mixed
4. Other (Maps, Network, Time Series)

We might need to plot:

A. Single variables

B. Multiple Variables

Visualisations

MAINLY SINGLE VARIABLE - Histogram/ Density plots - qqplots

MAINLY MULTIPLE VARIABLES - Pie Chart - Bar Chart - Boxplot - Scatter plots - Heatmaps - Correlation plots

The Iris Dataset

```
dim(iris)
```

```
[1] 150   5
```

```
str(iris)
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
summary(iris)
```

```
  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
```

```
attach(iris)
```

Seperate the iris data by Species

```
A<-split(iris,iris$Species)
summary(A)
```

```
           Length Class      Mode
setosa     5      data.frame list
versicolor 5      data.frame list
virginica  5      data.frame list
```

Create data frames

```
head(A$setosa,3)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
```

```
setosa<-as.data.frame(A$setosa[,1:4])
versicolor<-as.data.frame(A$versicolor[,1:4])
virginica<-as.data.frame(A$virginica[,1:4])
head(setosa,3)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
```

Histogram of Sepal Length all species

```
hist(iris$Sepal.Length,
  breaks = 14,
  main= "Histogram of Sepal Length",
  sub = "All Species",
  xlim = range(c(4, max(iris$Sepal.Length))),
  xlab="Sepal Length (cm)",
  col="steelblue")
```

Histograms of Sepal Length for each species

```
par(mfrow=c(1,3))

hist(setosa$Sepal.Length,
  breaks = 10,
  main= "Setosa",
  xlim = range(4, 8),
  xlab="Sepal Length (cm)",
  col ="red")

hist(versicolor$Sepal.Length,
  breaks = 10,
  main= "Versicolor",
  xlim = range(4, 8),
  xlab="Sepal Length (cm)",
  col ="green")

hist(virginica$Sepal.Length,breaks = 10,
  main= "Virginica",
  xlim = range(4, 8),
  xlab="Sepal Length (cm)",
  col ="yellow")
```

Boxplot

```
par(mfrow=c(2,2))

boxplot(Sepal.Length ~ Species,
  data = iris,
  col = c("red", "green", "yellow"),
  main="Sepal Length", xlab= "Species",
  ylab= "Sepal Length (cm)")

boxplot(Sepal.Width ~ Species,
  data = iris,
  col = c("red", "green", "yellow"),
  main="Sepal Width", xlab= "Species",
```

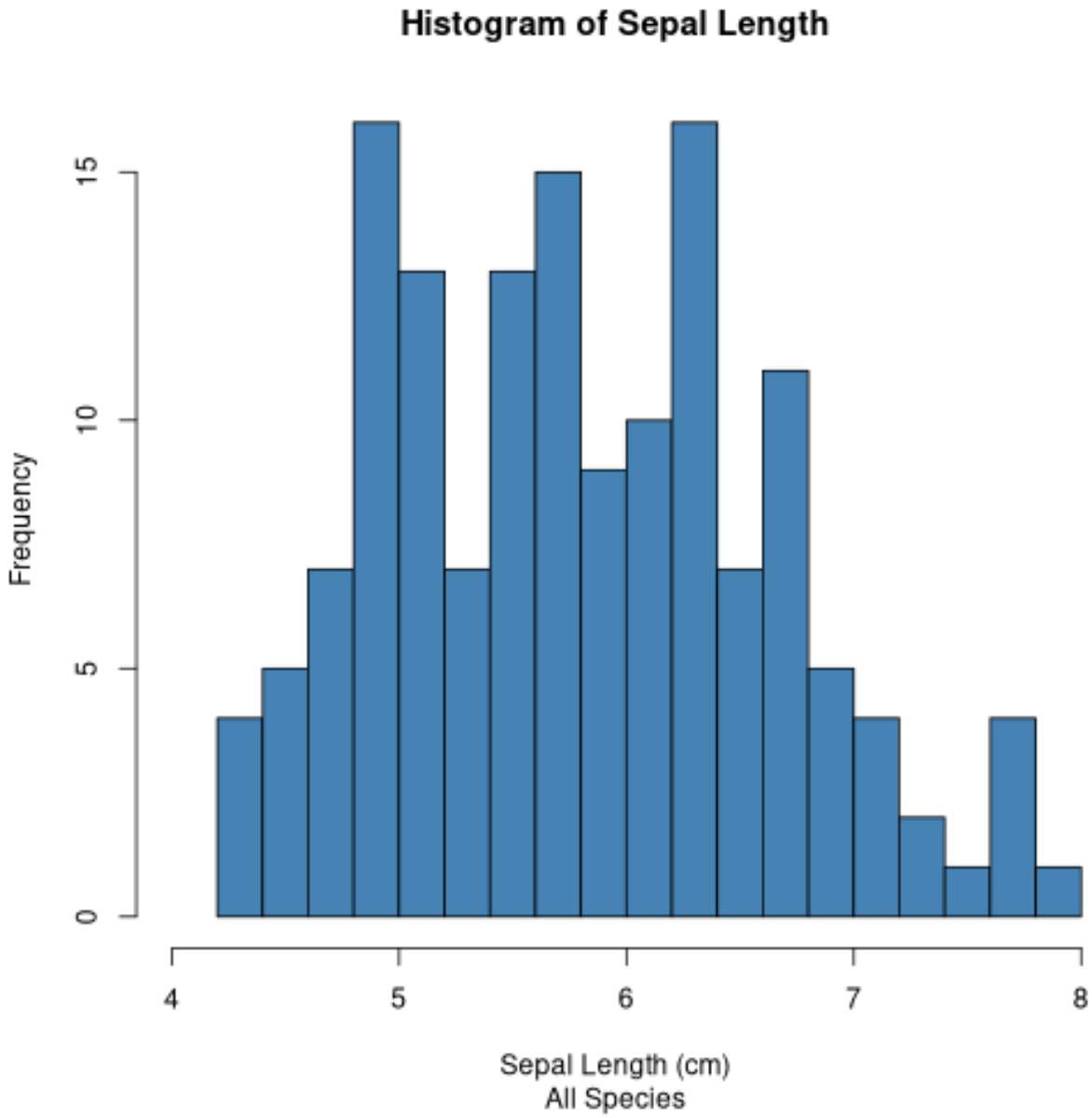


Figure 3: Histogram of Sepal Length for all species combined

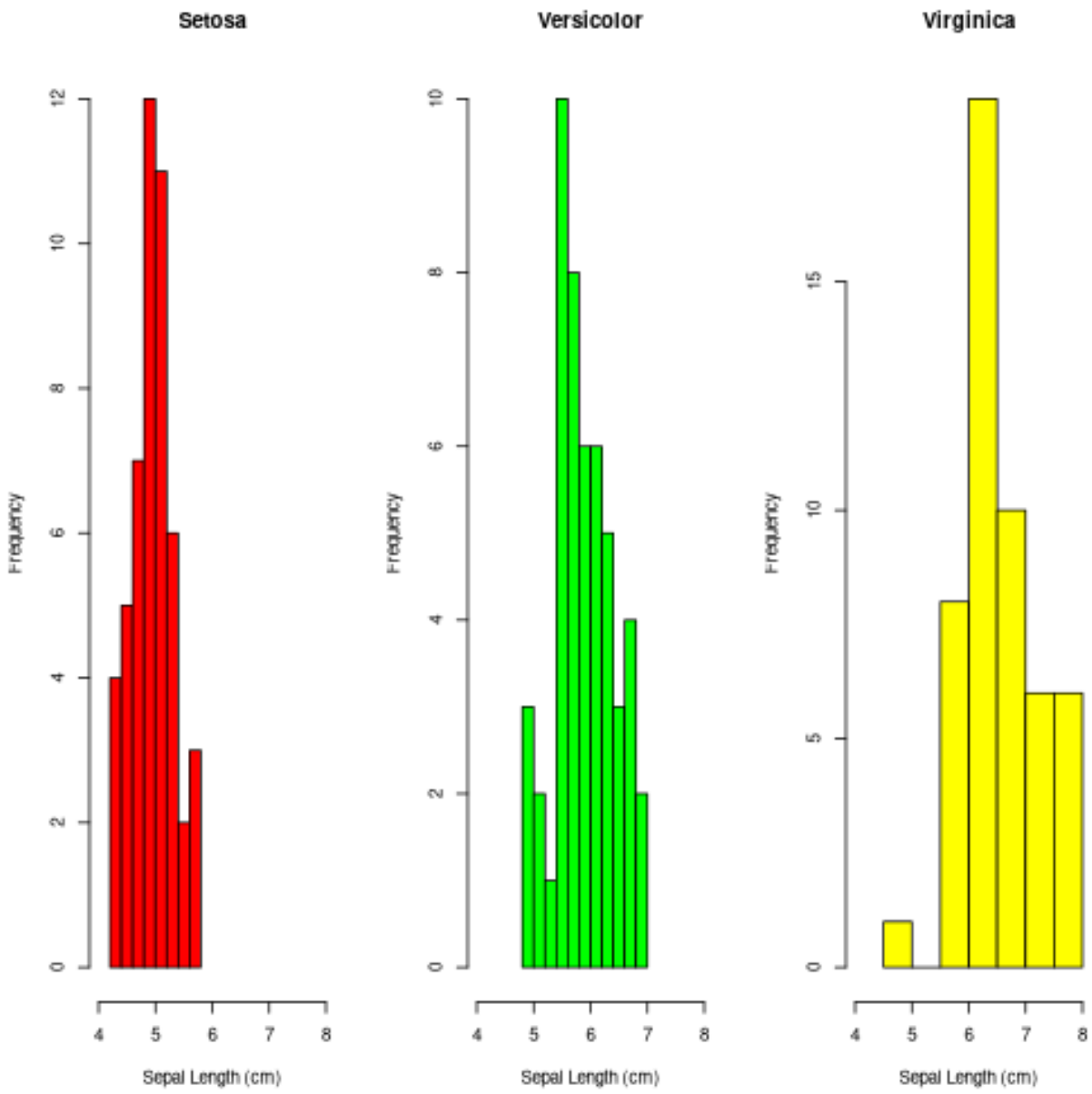


Figure 4: Histograms of Sepal Length for each species

```
  ylab= "Sepal Width (cm)")

boxplot(Petal.Length ~ Species,
  data = iris,
  col = c("red", "green", "yellow"),
  main="Petal Length", xlab= "Species",
  ylab= "Petal Length (cm)")

boxplot(Petal.Width ~ Species,
  data = iris,
  col = c("red", "green", "yellow"),
  main="Petal Width", xlab= "Species",
  ylab= "Petal Width (cm)")
```

It seems that setosa are characterised by smaller flowers.

ScatterPlot

```
par(mfrow=c(1,2))

plot(Sepal.Length ~ Sepal.Width,
  data = iris,
  col=c("red", "green","yellow"))

plot(Petal.Length ~ Petal.Width,
  data = iris,
  col=c("red", "green","yellow"))
```

qqplots

```
par(mfrow=c(3,4))

qqnorm(setosa$Sepal.Length,
  main="Sepal Length", ylab = "SETOSA")

qqline(setosa$Sepal.Length)

qqnorm(setosa$Sepal.Width,
  main="Sepal Width ", ylab='')

qqline(setosa$Sepal.Width)

qqnorm(setosa$Petal.Length,
  main ="Petal Length", ylab='')

qqline(setosa$Petal.Length)

qqnorm(setosa$Petal.Width,
  main = "Petal Width", ylab='')

qqline(setosa$Petal.Width)

qqnorm(versicolor$Sepal.Length,
```

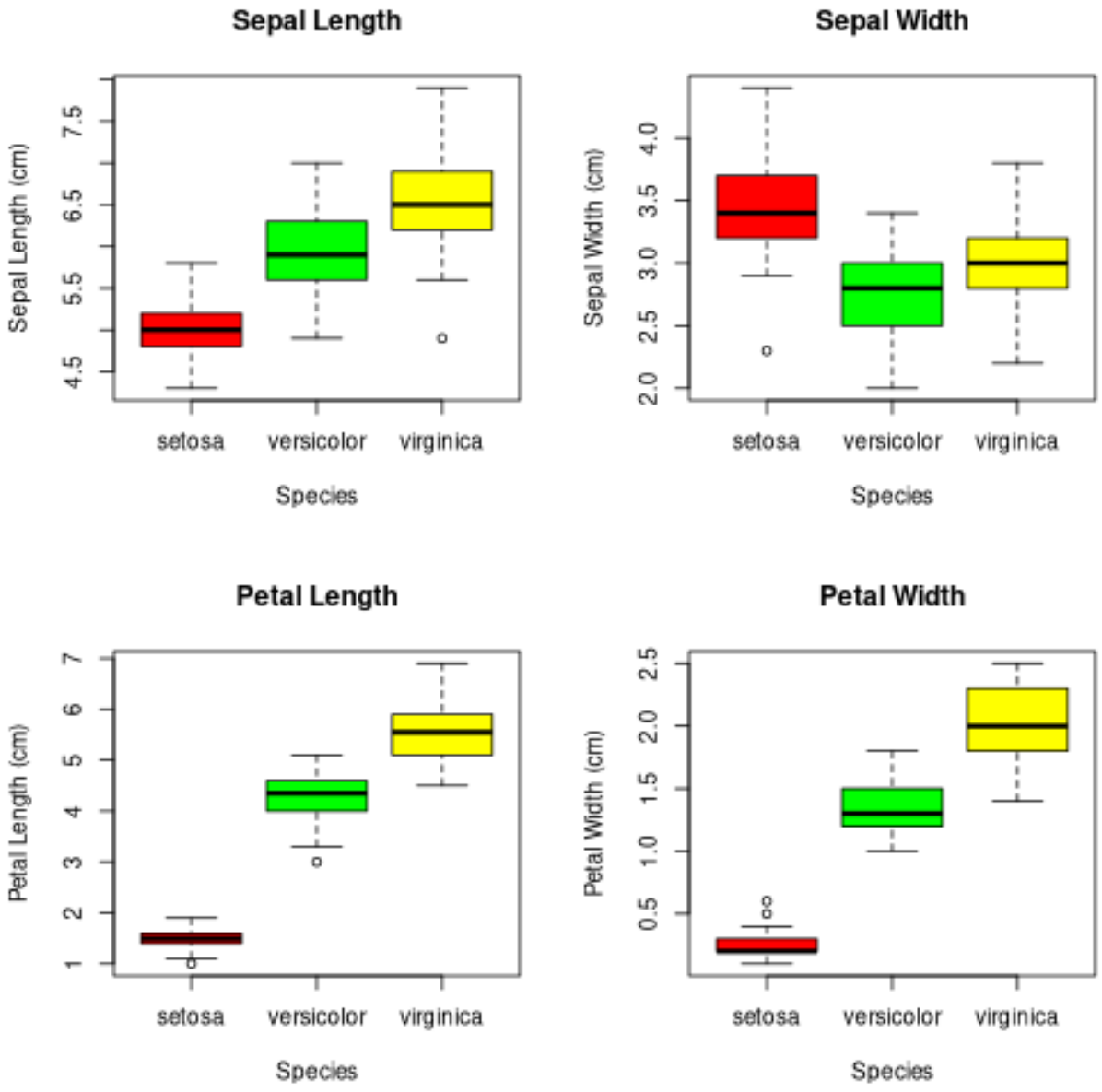


Figure 5: Boxplots for all variables in iris dataset per species

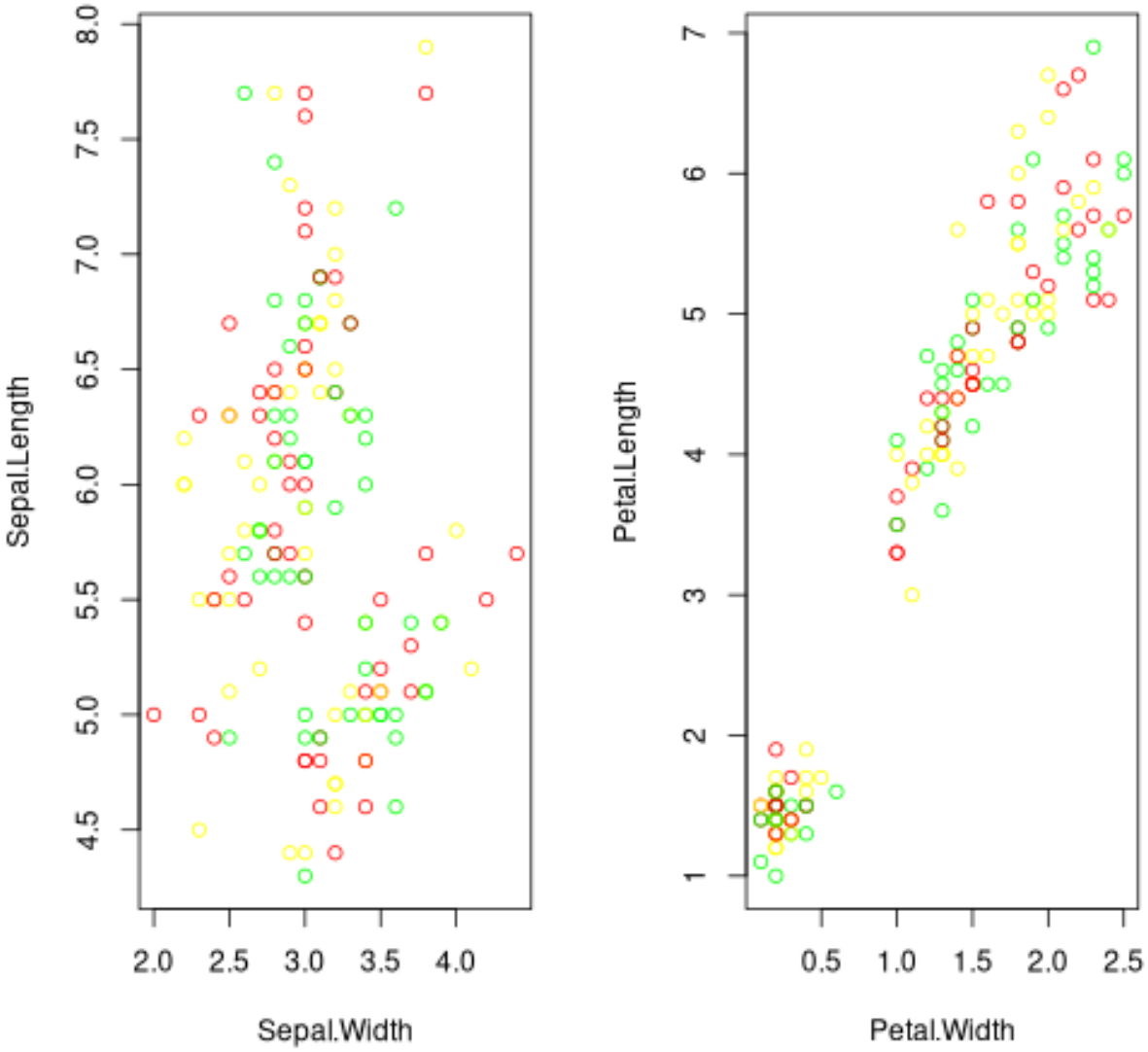


Figure 6: Scatter Plots for all species Length vs Width for Sepals and Petals

```
  main='', ylab = "VERSICOLOR")

qqline(versicolor$Sepal.Length)

qqnorm(versicolor$Sepal.Width,
  main='', ylab='')

qqline(versicolor$Sepal.Width)

qqnorm(versicolor$Petal.Length,
  main ='', ylab='')

qqline(versicolor$Petal.Length)

qqnorm(versicolor$Petal.Width,
  main = '', ylab='')

qqline(versicolor$Petal.Width)

qqnorm(virginica$Sepal.Length,
  main='', ylab = "VIRGINICA")

qqline(virginica$Sepal.Length)

qqnorm(virginica$Sepal.Width,
  main='', ylab='')

qqline(virginica$Sepal.Width)

qqnorm(virginica$Petal.Length,
  main ='', ylab='')

qqline(virginica$Petal.Length)

qqnorm(virginica$Petal.Width,
  main = '', ylab='')

qqline(virginica$Petal.Width)
```

Most are **"Normally"** distributed

For the plots the commands qqnorm(species*variable*); *qqline*(*species*variable) is used

Pie Chart

```
pie(table(iris$Species),
  labels=paste(names(table(iris$Species)),
  "\n", table(iris$Species), sep=""),
  col =c("red","green","yellow"),
  main="Number of observations per species")
```

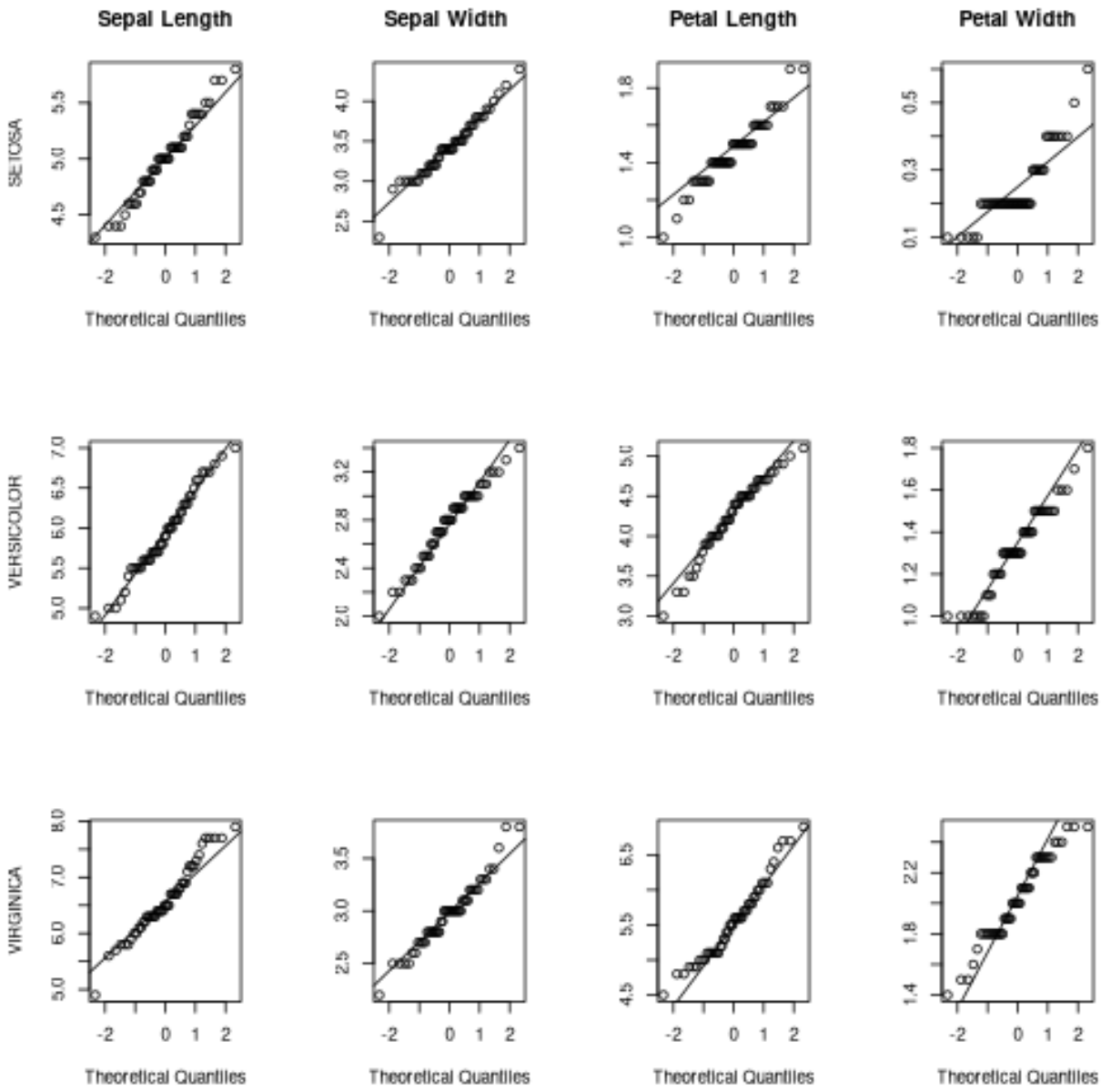


Figure 7: qqplots with normality line

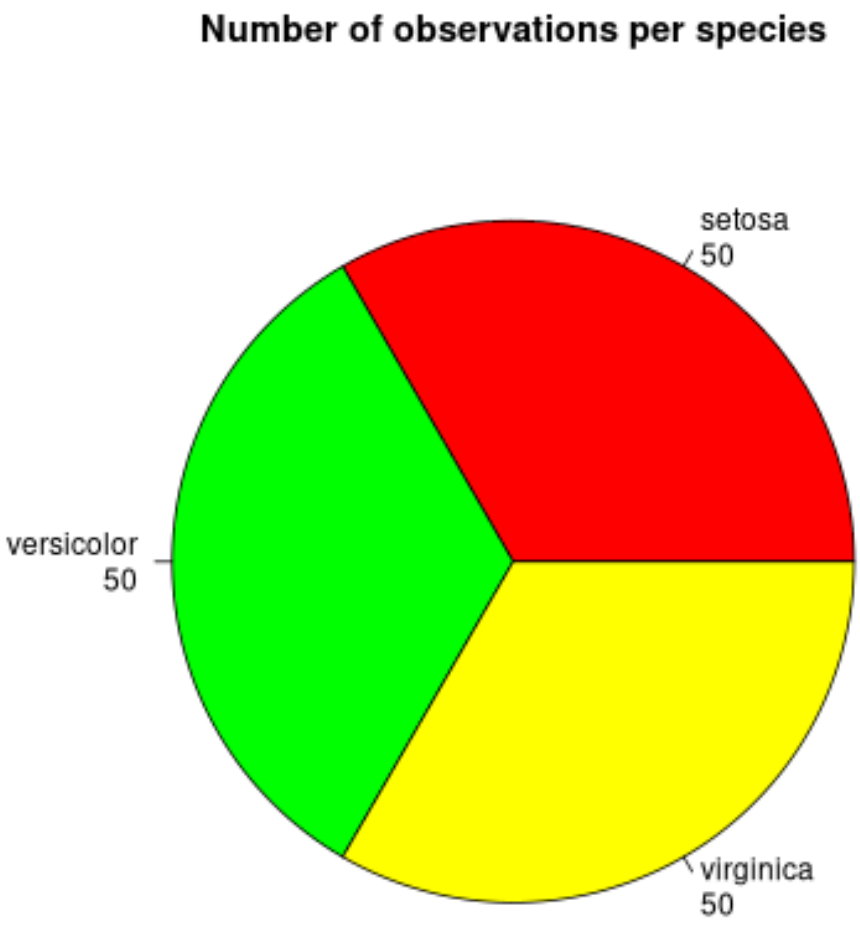


Figure 8: Pie chart of the number of observations per species on the iris dataset

Paired Scatter Plots

```
## put histograms on the diagonal
panel.hist <- function(x, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col = "steelblue", ...)
}
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits = digits)[1]
    txt <- paste0(prefix, txt)
    if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex.cor * r)
} # pearson correlation function for the upper panel
pairs(iris[1:4], main = "Iris data set paired plots",
  panel = panel.smooth,
  cex = 1.5, pch = 21,
  bg = c("red", "green", "yellow")[unclass(iris$Species)],
  diag.panel=panel.hist, cex.labels = 2, font.labels =2,
  upper.panel=panel.cor, gap = 0, rowlattop =FALSE)
```

A combination of plots that shows:

- Scatter plots on the lower panel with each species labeled in different colour
- Histograms on the diagonal
- Pearson correlation coefficients with font size proportionate to the coefficient value on the upper panel Functions for the plots generated with code from the R help for the pairs function.

Flower size can seperate the three species particularly in combination with Sepal Length. Sepal Width does not appear to vary between the three species.

Correlation plots

```
library(corrplot)

cor.m<-cor(iris[1:4])

col <- colorRampPalette(c("#BB4444",
  "#EE9988",
  "#FFFFFF",
  "#77AADD",
  "#4477AA"))

corrplot(cor.m,
  method = "color",
  col = col(200),
  type = "upper",
  order = "hclust",
```

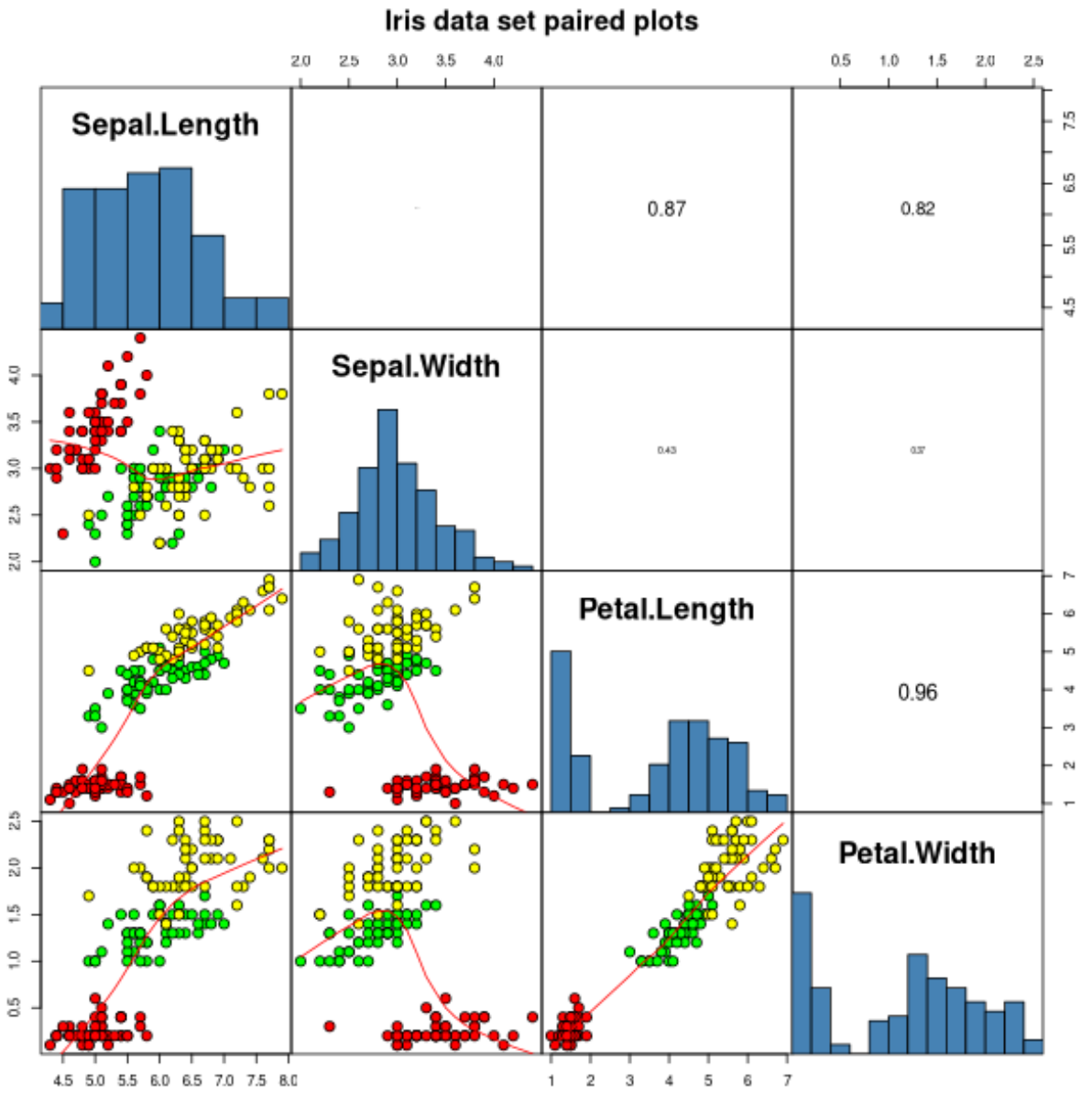


Figure 9: Paired scatter plot of the iris dataset. Lower panel: Scatter plots, Diagonal: Histograms, Upper panel: Pearson correlations.

```
  number.cex = 1.2,
  addCoef.col = "black",
  tl.col = "black",
  tl.srt = 90,
  p.mat = cor.mtest(iris[1:4])$p,
  sig.level = 0.05,
  insig = "blank",
  diag = FALSE)
```

Heatmap

```
library(gplots)

heatmap.2(t(iris[, 1:4]),
  trace="none",
  scale="row",
  key=TRUE,
  mar=c(2, 8),
  cexRow=1,
  ColSideColors=c("red",
  "green",
  "yellow")[iris$Species])
```

PCA

```
library(pca3d)

pca.iris<-prcomp(iris[-5], scale = TRUE)

pca2d(pca.iris,
  group = factor(iris$Species),
  legend ="topright",
  biplot = TRUE,
  show.centroids=FALSE,
  show.ellipses=TRUE,
  ellipse.ci=0.75,
  show.shadows= FALSE,
  show.group.labels = FALSE)
```

PCA demonstrates the differences between setosa and the two other species, with smaller petals and shorter sepals.

ggplot2

It's hard to succinctly describe how ggplot2 works because it embodies a deep philosophy of visualisation. However, in most cases you start with ggplot(), supply a dataset and aesthetic mapping (with aes()). You then add on layers (like geom_point() or geom_histogram()), scales (like scale_colour_brewer()), faceting specifications (like facet_wrap()) and coordinate systems (like coord_flip()). `{r, echo=FALSE} library(tidyverse) ggplot(data=iris) + geom_point(mapping = aes(x = Petal.Length, y =Petal.Width, color = Species)) + geom_smooth(mapping = aes(x = Petal.Length, y =Petal.Width, color = Species))`

For more info I attach the cheat sheet from RStudio for ggplot2

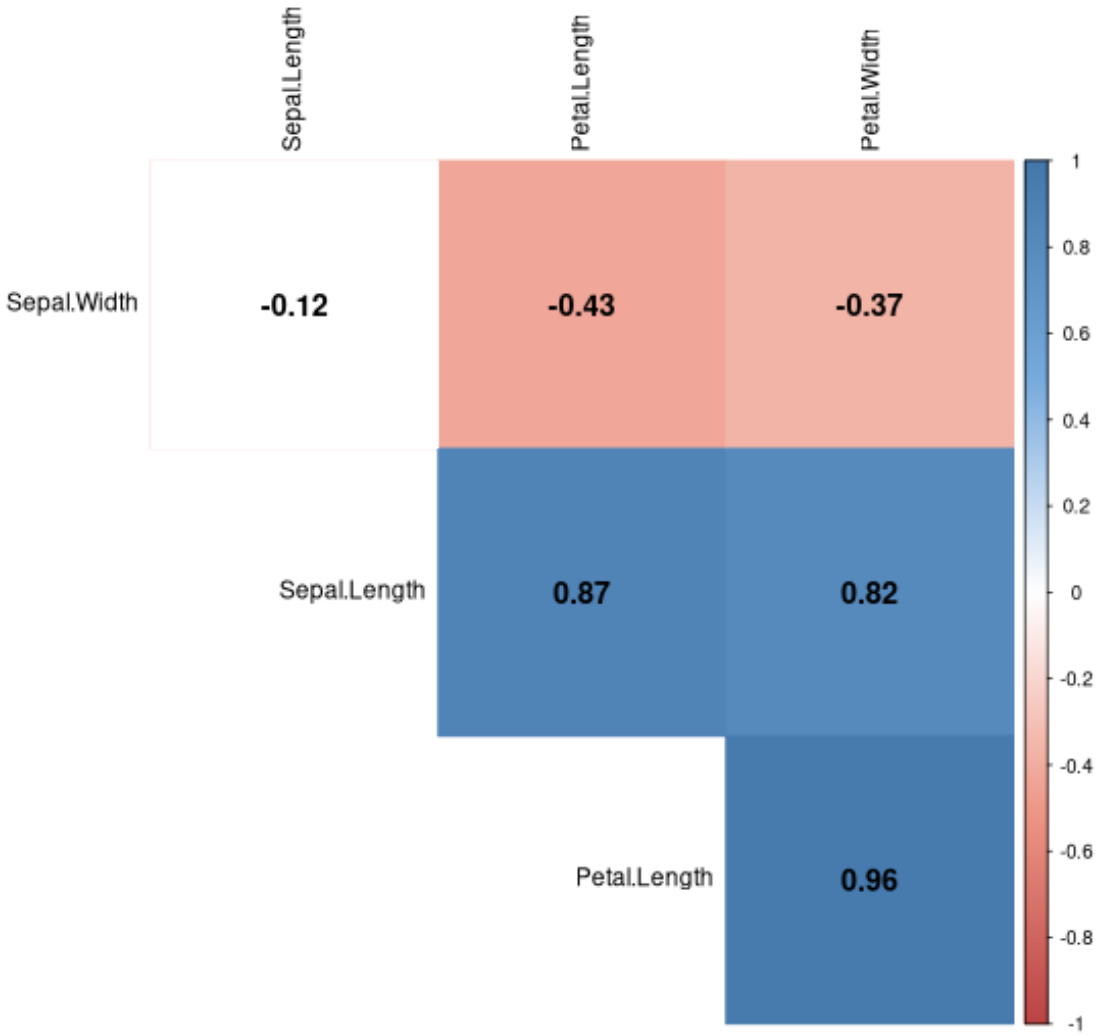


Figure 10: Correlation plot generated with the corrplot package on the iris dataset. Variables are ordered according to their euclidean distance.

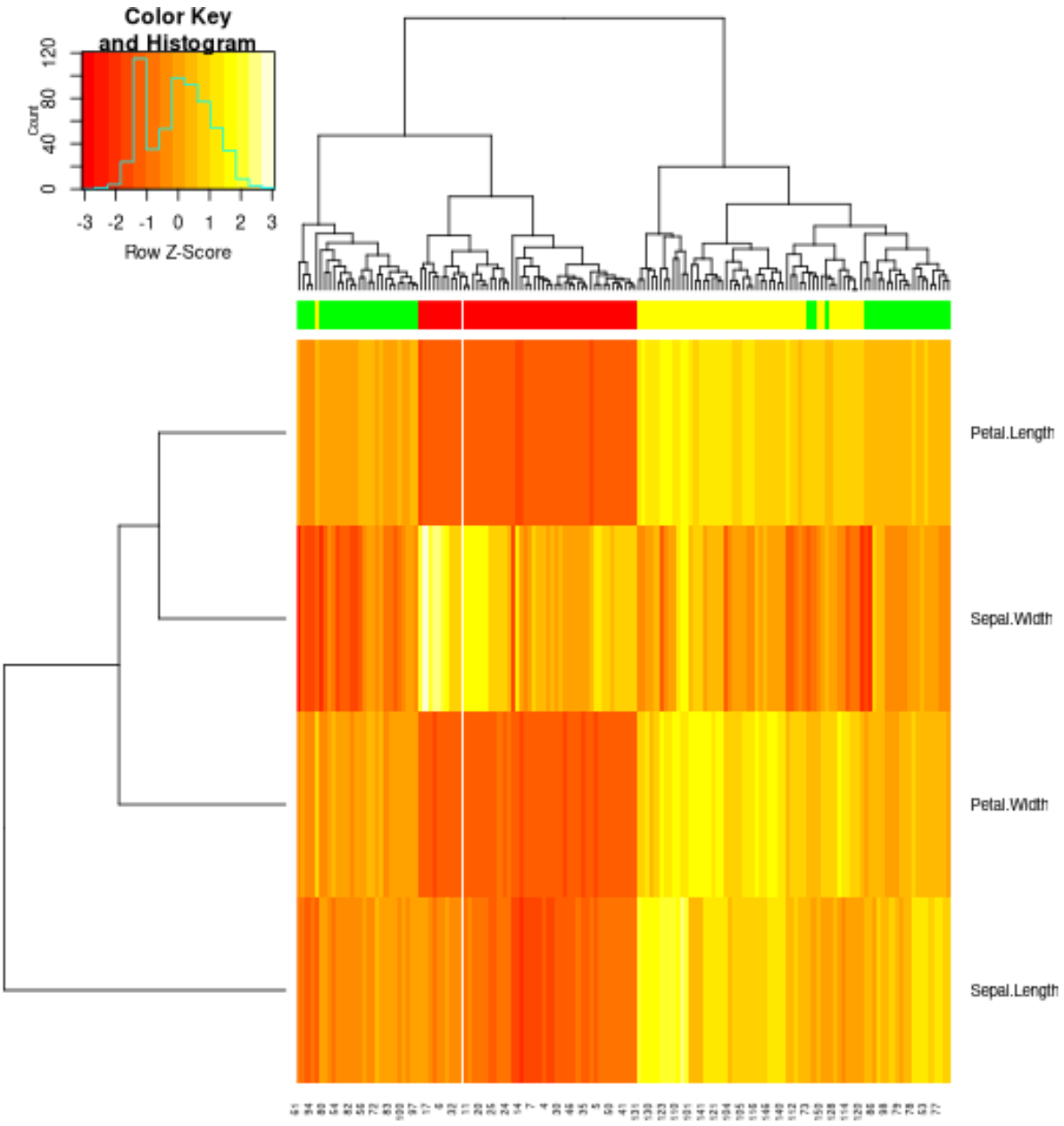


Figure 11: Heatmap for the iris dataset generated with gplots package.Setosa:red, Versicolor: green, Virginica: Yellow

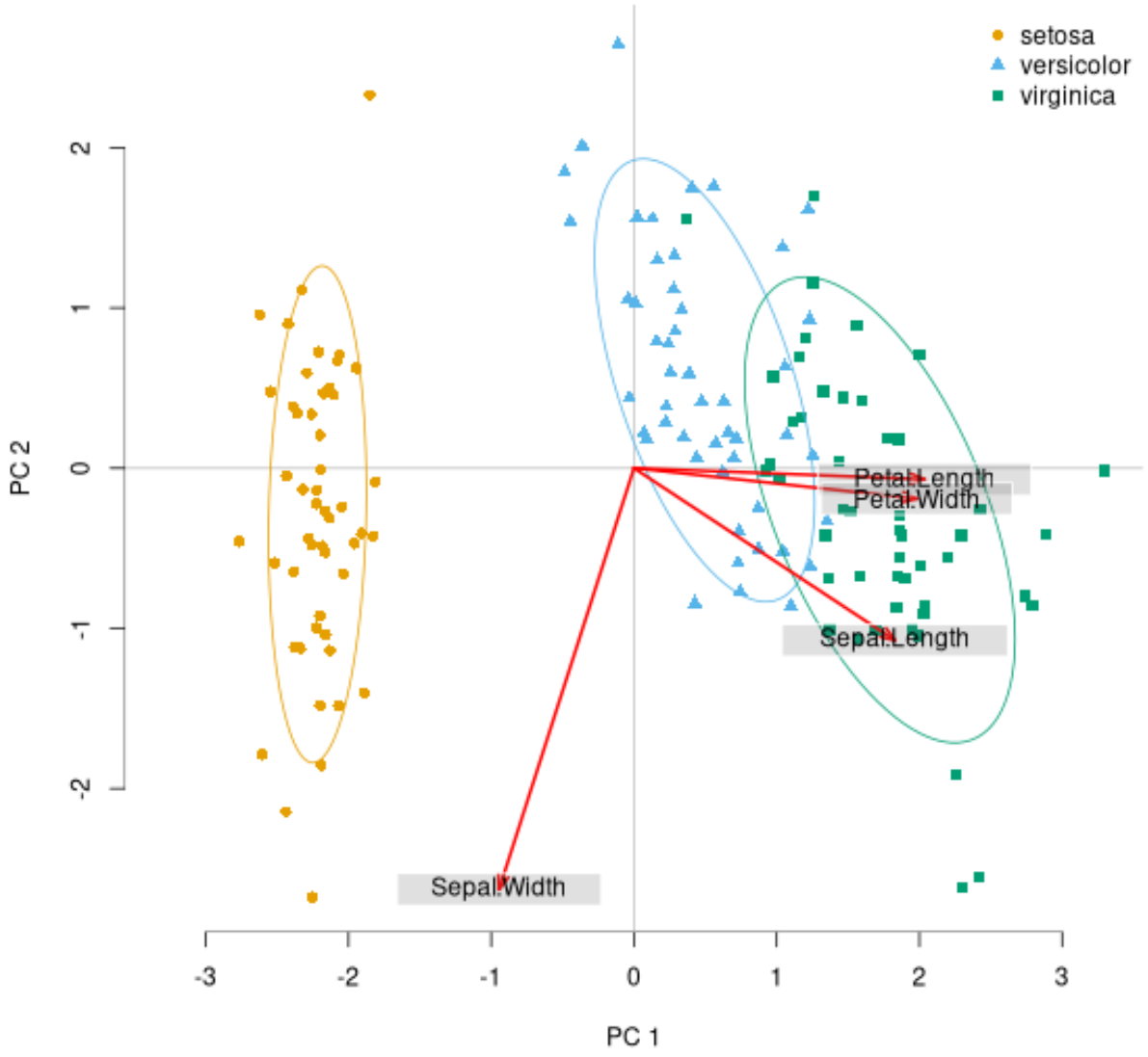


Figure 12: PCA of the iris datasheet generated with the pca3d package

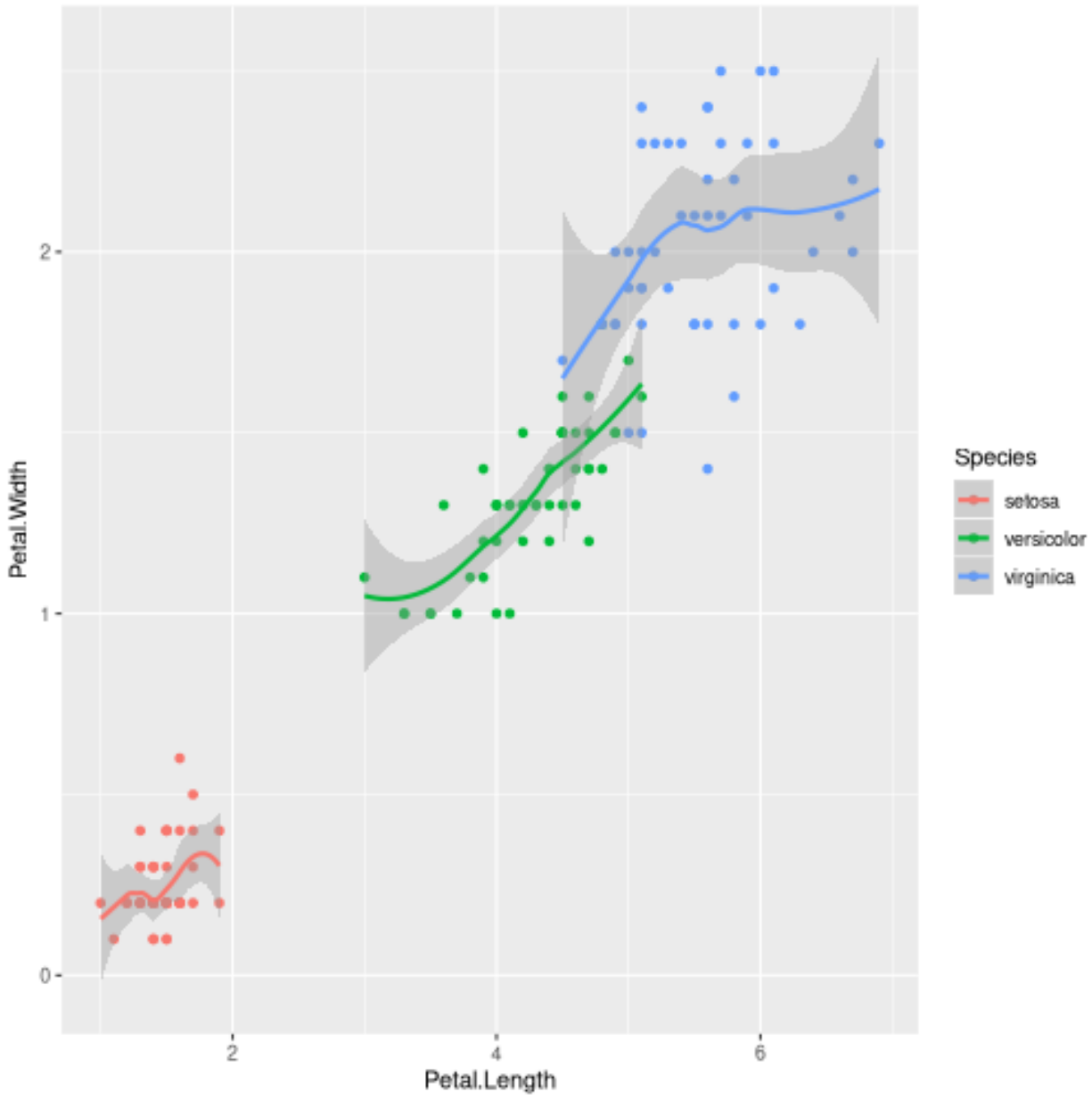


Figure 13: Scatter Plot with trendlines seperated by Species with the ggplot2 package

Data Visualization with ggplot2

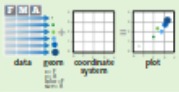
Cheat Sheet

RStudio

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.

To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.



Complete the template below to build a graph.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
    mapping = aes(<MAPPINGS>),
    stat = <STAT>,
    position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION> +
  <SCALE_FUNCTION> +
  <THEME_FUNCTION>
```

Required

Not required, sensible defaults supplied

ggplot(data = mpg, aes(x = cty, y = hwy))
Begins a plot that you finish by adding layers to. Add one geom function per layer.

aesthetic mappings · data · geom

qplot(x = cty, y = hwy, data = mpg, geom = "point")
Creates a complete plot with given data, geom, and mappings. Supplies many useful defaults.

last_plot()
Returns the last plot.

ggsave("plot.png", width = 5, height = 5)
Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Geoms - Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

Graphical Primitives

```
a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))
```

a + geom_blank()
(Useful for expanding limits)

b + geom_curve(aes(yend = lat + 1, xend = long+1,curvature=z)**)** - x,xend, y,yend, alpha, angle, color, curvature, linetype, size

a + geom_path(lineend="butt", linejoin="round", linemitre=1**)**
x, y, alpha, color, group, linetype, size

a + geom_polygon(aes(group = group)**)**
x, y, alpha, color, fill, group, linetype, size

b + geom_rect(aes(xmin = long, ymin=lat, xmax= long + 1, ymax= lat + 1)**)** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin=unemploy - 900, ymax=unemploy + 900)**)** - x, ymax, ymin
alpha, color, fill, group, linetype, size

Line Segments

common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept=0, slope=1)**)**
b + geom_hline(aes(yintercept = lat)**)**
b + geom_vline(aes(xintercept = long)**)**
b + geom_segment(aes(yend=lat+1, xend=long+1)**)**
b + geom_spoke(aes(angle = 1:1155, radius = 1)**)**

One Variable

Continuous

```
c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)
```

c + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian"**)**
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5**)**
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy)**)**
x, y, alpha, color, fill, linetype, size, weight

Discrete

```
d <- ggplot(mpg, aes(fl))
```

d + geom_bar()
x, alpha, color, fill, linetype, size, weight

Two Variables

Continuous X, Continuous Y

```
e <- ggplot(mpg, aes(cty, hwy))
```

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1, check_overlap = TRUE**)**
x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e + geom_jitter(height = 2, width = 2**)**
x, y, alpha, color, fill, shape, size

e + geom_point()
x, y, alpha, color, fill, shape, size, stroke

e + geom_quantile()
x, y, alpha, color, group, linetype, size, weight

e + geom_rug(sides = "bl"**)**
x, y, alpha, color, linetype, size

e + geom_smooth(method = lm**)**
x, y, alpha, color, fill, group, linetype, size, weight

e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1, check_overlap = TRUE**)**
x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

Discrete X, Continuous Y

```
f <- ggplot(mpg, aes(class, hwy))
```

f + geom_col()
x, y, alpha, color, fill, group, linetype, size

f + geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f + geom_dotplot(binaxis = "y", stackdir = "center"**)**
x, y, alpha, color, fill, group

f + geom_violin(scale = "area"**)**
x, y, alpha, color, fill, group, linetype, size, weight

Discrete X, Discrete Y

```
g <- ggplot(diamonds, aes(cut, color))
```

g + geom_count()
x, y, alpha, color, fill, shape, size, stroke

Continuous Bivariate Distribution

```
h <- ggplot(diamonds, aes(carat, price))
```

h + geom_bin2d(binwidth = c(0.25, 500)**)**
x, y, alpha, color, fill, linetype, size, weight

h + geom_density2d()
x, y, alpha, colour, group, linetype, size

h + geom_hex()
x, y, alpha, colour, fill, size

Continuous Function

```
i <- ggplot(economics, aes(date, unemploy))
```

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

i + geom_step(direction = "hv"**)**
x, y, alpha, color, group, linetype, size

Visualizing error

```
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit-se, ymax = fit+se))
```

j + geom_crossbar(fatten = 2**)**
x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar()
x, ymax, ymin, alpha, color, group, linetype, size, width (also **geom_errorbarh()**)

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange()
x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

Maps

```
data <- data.frame(murder = USArrests$Murder,
state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))
```

k + geom_map(aes(map_id = state), map = map**) +**
expand_limits(x = mapSlong, y = mapSlat**)**
map_id, alpha, color, fill, linetype, size

Three Variables

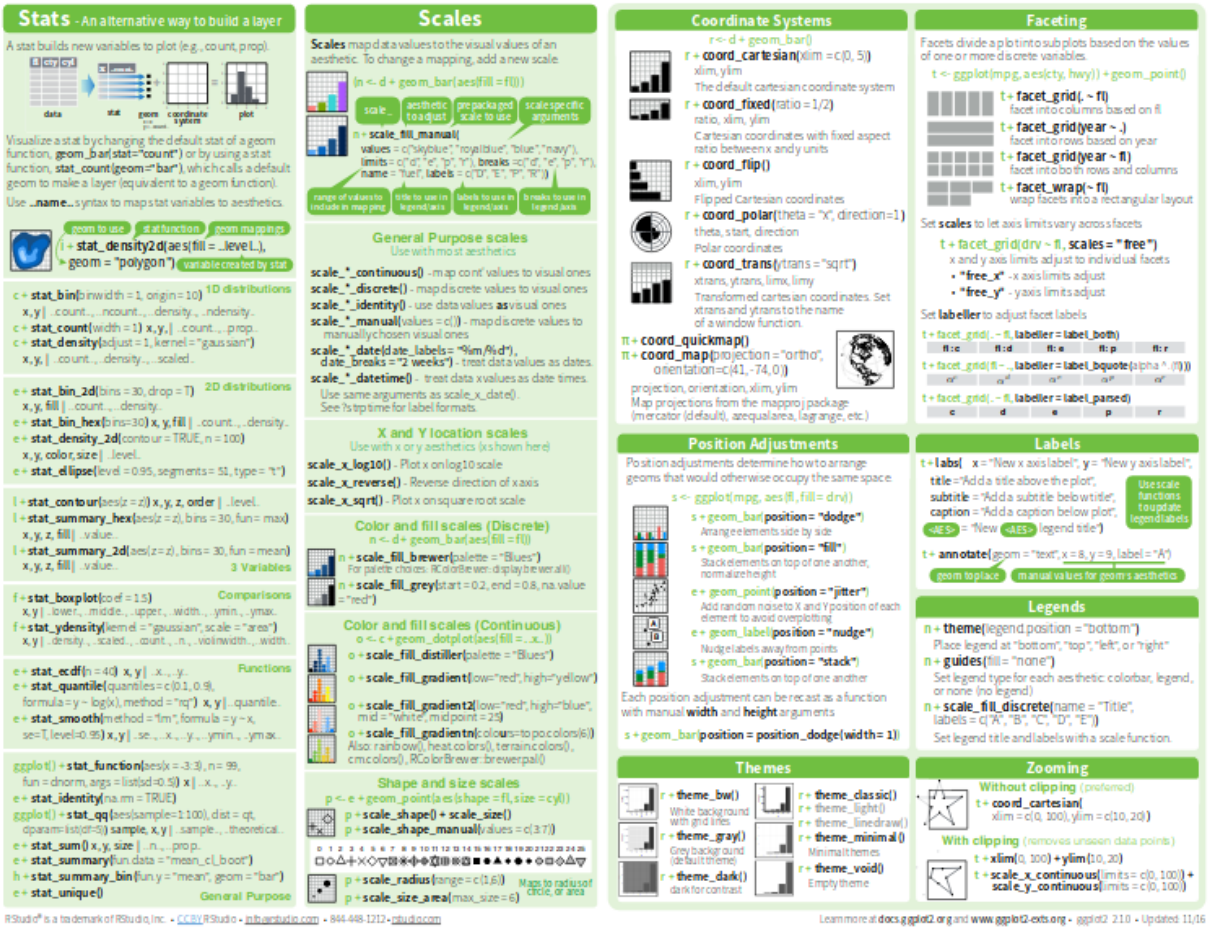
```
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2))
l <- ggplot(seals, aes(long, lat))
```

l + geom_contour(aes(z = z)**)**
x, y, z, alpha, colour, group, linetype, size, weight

l + geom_raster(aes(fill = z), hjust=0.5, vjust=0.5, interpolate=FALSE**)**
x, y, alpha, fill

l + geom_tile(aes(fill = z)**)**
x, y, alpha, color, fill, linetype, size, width

Learn more at **docs.ggplot2.org** and **www.ggplot2-exts.org** • ggplot2 2.1.0 • Updated: 11/16



Cheat sheets for ggplot2 package

RESOURCES

https://cran.r-project.org/

https://www.rdocumentation.org/

https://www.bioconductor.org/help/

https://rstudio.com/resources/cheatsheets/

https://www.tidyverse.org/articles/

https://www.gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf

https://togaware.com/onepager/

https://www.statmethods.net/index.html

Garett Grolemund O'Reilly Medi, Inc., Hands-On Programming with R 2014