Programming for Data Science in R

Efstathios (Stathis) D. Gennatas, MBBS AICSM PhD

2020-10-26

Contents

Welcome

This is the online book for the new UCSF Biostat 213, Fall 2020.
It is being updated regularly.

EDG, San Francisco, CA, October 2020

Introduction

Throughout this book you will see boxes with R code followed by its output, if any. The code (or input) is decorated with a teal border on the left to separate it from its output, like in the following example:

```
x <- rnorm(200)
x[1:20]
```

```
 [1]  1.36733704  1.36923924 -1.80234101  1.39115214 -0.55392449 -0.88017063
 [7]  0.40075800  0.01844506  0.40376335  1.10458735  0.82321280 -1.25445959
[13]  0.96665185  0.92676550 -2.62644808  1.83369051 -0.01109560 -0.95149158
[19] -0.23366163  0.53709415
```

Notice that R adds numbers in brackets in the beginning of each row. This happens when R prints the contents of a vector. The number is the integer index of the first element in that row. Therefore, the first one is always `[1]` and the number of the subsequent rows depends on how many elements fit in each line. If the output is a single element, it will still have `[1]` in front of it.

Also notice that if we enclose the assignment operation of a variable in parentheses, this prints the resulting value of the variable. Therefore, this:

```
(y <- 4)
```

```
[1] 4
```

is equivalent to:

```
y <- 4
y
```

```
[1] 4
```

Note that if you mouse over the input code box, a clickable "Copy to clipboard" appears on the top right of the box allowing you to copy paste into an R session or file.

Lastly, you will see the following Info, Note, or Warning boxes at times:

This is some info

This is a note

This is a warning

This book was created using bookdown[1] (Xie, 2020)

[1] `https://CRAN.R-project.org/package=bookdown`

R setup, Packages & Docs

0.1 R

This book was compiled using R version 4.0.3 (2020-10-10).
Make sure you have the latest version by visiting the R project website[2]

It's a good idea to keep a log of the version of R and installed packages when beginning a new project. An easy way to do this is to save the output of `sessionInfo()`:

```
sessionInfo()
```

```
R version 4.0.3 (2020-10-10)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Catalina 10.15.6

Matrix products: default
BLAS:  /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dy

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.0.3  magrittr_1.5    bookdown_0.20   htmltools_0.5.0
 [5] tools_4.0.3     yaml_2.2.1      stringi_1.5.3   rmarkdown_2.4.4
 [9] knitr_1.29      stringr_1.4.0   digest_0.6.25   xfun_0.17
[13] rlang_0.4.7     evaluate_0.14
```

[2] https://www.r-project.org

0.2 R packages

0.2.1 CRAN

The Comprehensive R Archive Network (CRAN) is the official R package repository and currently hosts 16271 packages (as of 2020-09-13). To install a package from CRAN, use the builtin `install.packages` command:

```
install.packages('glmnet')
```

0.2.1.1 Check for outdated packages

```
old.packages()
```

0.2.1.2 Update installed packages

If you don't set `ask = FALSE`, you will have to accept each package update separately.

```
update.packages(ask = FALSE)
```

0.2.2 GitHub

GitHub contains a large number of R packages, some of which also exist in CRAN, but the GitHub version may be updated a lot more frequently. To install from GitHub, you need to have the `remotes` package from CRAN first:

```
install.packages("remotes")
```

```
remotes::install_github("username/reponame")
```

Note: Running `remotes::install_github("user/repo")` will not reinstall a previously installed package, unless it has been updated.

0.2.3 Bioconductor

Bioconductor is a repository which includes tools for the analysis and comprehension of high-throughput genomic data, among others. To install package from Bioconductor, first install the `BiocManager` package from CRAN:

```
install.packages("BiocManager")
```

and then use that similar to the builtin `install.packages`:

```
BiocManager::install("packageName")
```

0.2.4 Installed packages

List all R packages installed on your system with `installed.packages()` (the following block has not been run to prevent a very long output)

```
installed.packages()
```

List attached packages with `search()`:

```
search()
```

```
[1] ".GlobalEnv"     "package:stats"    "package:graphics"
[4] "package:grDevices" "package:utils"    "package:datasets"
[7] "package:methods"   "Autoloads"        "package:base"
```

List attached packages with their system path:

```
searchpaths()
```

```
[1] ".GlobalEnv"
[2] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/stats"
[3] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/graphics"
[4] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/grDevices"
[5] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/utils"
[6] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/datasets"
[7] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/methods"
[8] "Autoloads"
[9] "/Library/Frameworks/R.framework/Resources/library/base"
```

0.2.5 Dependencies

Most R packages, whether in CRAN, Bioconductor, or GitHub, themselves rely on other packages to run. These are called **dependencies**. Many of these dependencies get installed automatically when you call `install.packages()` or `remotes::install_github()`, etc. This depends largely on whether they are essential for the new package to work. Some packages, especially if they provide a large number of functions that may not all be used by all users, may make some dependencies optional. In that cases, if you try to execute a specific function that depends on uninstalled packages you may get a warning or error or some type of message indicating that you need to install further packages.

0.3 RStudio IDE

RStudio[3] is an Integrated Development Environment (IDE[4]) for R, which can make work in R easier, more productive, and more fun. Make sure to keep your installation up-to-date; new features are added often.

It is recommended to set up a new RStudio project for each data project: Select File > New Project… from the main menu.

0.4 Builtin Documentation

After you've successfully installed R and RStudio, one of the first things to know is how to access and search the builtin documentation.

0.4.1 Get help on a specific item

If you know the name of what you're looking for (an R function most commonly, but possibly also the name of a dataset, or a package itself), just type `?` followed by the name of said function, dataset, etc. in the R prompt:

```
?sample
```

In RStudio, the above example will bring up the documentation for the `sample` function in the dedicated "Help" window, commonly situated at the bottom right (but can be moved by the user freely). If you are running R directly at the system shell, the same information is printed directly at the console.
Try running the above example on your system.

[3] `https://rstudio.com/`
[4] `https://en.wikipedia.org/wiki/Integrated_development_environment`

0.4.2 Search the docs

If you do not know the name of what you are looking for, you can use double question marks, `??`, followed by your query (this is short for the `help.search` command that provides a number of arguments you can look up using `?help.search`):

```
??bootstrap
```

Basic operations

First, before even learning about data types and structures, it may be worth looking at some of the basic mathematical and statistical operations in R.

0.5 Arithmetic

```
x <- 10
y <- 3
```

Standard arithmetic operations are as expected:

```
x + y
```

```
[1] 13
```

```
x - y
```

```
[1] 7
```

```
x * y
```

```
[1] 30
```

```
x / 3
```

```
[1] 3.333333
```

Exponentiation uses ^: (This is worth pointing out, because while ^ is likely the most common way to represent exponentiation, the symbol used for exponentiation varies across languages)

```
x^3
```

```
[1] 1000
```

Square root is `sqrt()`:

```
sqrt(81)
```

```
[1] 9
```

Integer division i.e. *Divide and forget the remainder*

```
x %/% 3
```

```
[1] 3
```

i.e. how many times the denominator fits in the numerator, without taking fractions of the denominator. It can be applied on decimals the same way:

```
9.5 %/% 3.1
```

```
[1] 3
```

Modulo operation i.e. *Divide and return just the remainder*

```
x %% y
```

```
[1] 1
```

```
x <- (-10:10)[-11]
y <- sample((-10:10)[-11], 20)
x - (x %/% y) * y == x %% y
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TR
[16] TRUE TRUE TRUE TRUE TRUE
```

Try to figure out what the following does:

```
x <- rnorm(20)
y <- rnorm(20)
round(x - (x %/% y) * y, 5) == round(x %% y, 5)
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[16] TRUE TRUE TRUE TRUE TRUE
```

`round(x, digits)` rounds `x` to the desired number of digits and is used to overcome rounding errors.

0.6 Logical operations

Logical AND with `&`

```
T & T
```

```
[1] TRUE
```

```
T & F
```

```
[1] FALSE
```

Logical OR with `|`

```
T | F
```

```
[1] TRUE
```

Logical negation with `!`

```
x <- TRUE
!x
```

```
[1] FALSE
```

Exclusive OR with `xor()` (= one or the other is TRUE but not both)

```
a <- c(T, T, T, F, F, F)
b <- c(F, F, T, F, T, T)
a & b
```

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

```
a | b
```

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

```
xor(a, b)
```

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

Test all elements are TRUE with `all()`:

```
all(a)
```

```
[1] FALSE
```

Test if any element is TRUE with `any()`:

```
any(a)
```

```
[1] TRUE
```

0.7 Common descriptive stats

Let's use the `rnorm` function to draw 200 numbers from a random normal distribution:

```
x <- rnorm(200)
```

Basic descriptive stat operations:

```
mean(x)
```

```
[1] 0.07256893
```

```
median(x)
```

```
[1] 0.03077537
```

```
sd(x) # standard deviation
```

```
[1] 0.9328076
```

```
min(x)
```

```
[1] -2.476455
```

```
max(x)
```

```
[1] 2.897475
```

```
range(x)
```

```
[1] -2.476455  2.897475
```

```
summary(x)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
-2.47646 -0.53167  0.03078  0.07257  0.82637  2.89748 
```

Data Types

In R, everything is an object.

Every "action" is a function.

Functions are also objects, which means they can be passed as arguments to functions or returned from other functions.

We shall see the relevance of this, for example, in the Loop Functions chapter.

0.8 Base types

R includes a number of builtin data types.

These are defined by the R core team: users cannot define their own data types, but they can define their own classes - see section on Classes and Object-Oriented Programming.

Some of the more popular data types in R are:

- Logical (a.k.a. Boolean[5])
- Numeric, integer
- Numeric, double[6]
- Character
- Environment
- Closure (i.e. function)

Many errors in R occur because a variable is, or gets coerced to, the wrong type by accident.

[5] https://en.wikipedia.org/wiki/Boolean_data_type
[6] https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Check variable types with `typeof()` and/or `str()`.

0.9 Assignment

Use ← for all assignments

```
x <- 3
# You can add comments within code blocks using the usual "#" prefix
```

Typing the name of an object...

```
x
```

```
[1] 3
```

...is equivalent to printing it

```
print(x)
```

```
[1] 3
```

You can also place any assignment in parentheses and this will perform the assignment and print the object:

```
(x <- 3)
```

```
[1] 3
```

While you *could* use the equal sign '=' for assignment, you should only use it to pass arguments to functions.

You can assign the same value to multiple objects - this can be useful when initializing variables.

```
x <- z <- init <- 0
x
```

```
[1] 0
```

```
z
```

```
[1] 0
```

```
init
```

```
[1] 0
```

Excitingly, R allows assignment in the opposite direction as well:

```
10 -> x
x
```

```
[1] 10
```

We shall see later that the → assignment can be convenient at the end of a pipe[7].

You can even do this, which is fun (?) but unlikely to be useful:

```
x <- 7 -> z
x
```

```
[1] 7
```

```
z
```

```
[1] 7
```

Use `c()` to combine multiple values into a vector - this is one of the most widely used R functions:

```
x <- c(-12, 3.5, 104)
x
```

```
[1] -12.0   3.5 104.0
```

0.10 Initialize - coerce - test (types)

The following summary table lists the functions to *initialize*, *coerce* (=convert), and *test* the core data types, which are shown in more detail in the following paragraphs:

[7] https://class.lambdamd.org/progdatscir/functions.html#the-pipe-operator

Initialize	Coerce	Test
logical(n)	as.logical(x)	is.logical(x)
integer(n)	as.integer(x)	is.integer(x)
double(n)	as.double(x)	is.double(x)
character(n)	as.character(x)	is.character(x)

0.11 Logical

If you are writing code, use TRUE and FALSE.
On the console, you can abbreviate to T and F.

```
a <- c(TRUE, FALSE)
a <- c(T, F)
x <- 4
b <- x > 10
b
```

```
[1] FALSE
```

```
str(b)
```

```
 logi FALSE
```

```
typeof(b)
```

```
[1] "logical"
```

0.12 Integer

Create a range of integers using colon notation `start:end`:

```
(x <- 11:15)
```

```
[1] 11 12 13 14 15
```

```
typeof(x)
```

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

```
str(x)
```

```
 int [1:5] 11 12 13 14 15
```

Note that assigning an integer defaults to type double:

```
x <- 1
typeof(x)
```

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

```
str(x)
```

```
 num 1
```

You can force it to be stored as integer by adding an L suffix:

```
x <- 1L
typeof(x)
```

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

```
str(x)
```

```
 int 1
```

```
x <- c(1L, 3L, 5L)
str(x)
```

```
 int [1:3] 1 3 5
```

0.13 Double

```
x <- c(1.2, 3.4, 10.987632419834556)
x
```

```
[1]  1.20000  3.40000 10.98763
```

```
typeof(x)
```

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

```
str(x)
```

```
 num [1:3] 1.2 3.4 11
```

0.14 Character

A character vector consists of one or more elements, each of which consists of one or more actual characters, i.e. it is **not** a vector of single characters. (The length of a character vector is the number of individual elements, and is not related to the number of characters in each element)

```
x <- "word"
typeof(x)
```

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

```
length(x)
```

```
[1] 1
```

```
(x <- c("a", "b", "gamma", "delta"))
```

```
[1] "a"     "b"     "gamma" "delta"
```

```
typeof(x)
```

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

```
length(x)
```

```
[1] 4
```

0.15 Environment

Defining your own environments is probably for advanced use only:

```
x <- new.env()
x$name <- "Guava"
x$founded <- 2020
x
```

```
<environment: 0×7fd44e73e818>
```

```
typeof(x)
```

```
[1] "environment"
```

0.16 Closure (function)

Closures are functions - they contain their own variable definitions.
Read more on functions.

```
square <- function(x) x^2
square(3)
```

```
[1] 9
```

```
typeof(square)
```

```
[1] "closure"
```

0.17 Initialize vectors

You can create / initialize vectors of specific type with the `vector` command and specifying a `mode` or directly by calling the relevant function:

```
(xl <- vector(mode = "logical", length = 10))
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
(xd <- vector(mode = "double", length = 10))
```

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

```
(xn <- vector(mode = "numeric", length = 10)) # same as "double"
```

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

```
(xi <- vector(mode = "integer", length = 10))
```

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

```
(xc <- vector(mode = "character", length = 10))
```

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

These are aliases of the `vector` command above (print their source code to see for yourself)

```
xl <- logical(10)
xd <- double(10)
xn <- numeric(10) # same as double
xi <- integer(10)
xc <- character(10)
```

0.18 Explicit coercion

We can explicitly convert objects of one type to a different type using `as.*` functions:

```
(x <- c(1.2, 2.3, 3.4))
```

```
[1] 1.2 2.3 3.4
```

```
(x <- as.logical(x))
```

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

```
(x <- as.double(x))
```

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

```
(x <- as.numeric(x))
```

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

```
(x <- as.integer(x))
```

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

```
(x <- as.character(x))
```

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

Logical vectors are converted to 1s and 0s as expected:

TRUE becomes 1 and FALSE becomes 0

```
x <- c(TRUE, TRUE, FALSE)
as.numeric(x)
```

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

Note that converting from numeric to logical **anything other than zero is TRUE**:

```
x <- seq(-2, 2, .5)
as.logical(x)
```

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

Not all conversions are possible.
There is no meaningful/consistent way to convert a character vector to numeric.
The following outputs NA values and prints a (helpful) error message.

```
x <- c("mango", "banana", "tangerine")
as.numeric(x)
```

```
Warning: NAs introduced by coercion
```

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

0.19 Implicit coercion

Remember, the language tries to make life easier and will often automatically coerce from one class to another to make requested operations possible.

For example, you can sum a logical vector.
It will automatically be converted to numeric as we saw earlier.

```
x <- c(TRUE, TRUE, FALSE)
sum(x)
```

```
[1] 2
```

On the other hand, you cannot sum a factor, for example.
You get an error with an explanation:

```
x <- factor(c("mango", "banana", "mango"))
sum(x)
```

```
Error in Summary.factor(structure(c(2L, 1L, 2L), .Label = c("banana", : 'su
```

Note: We had to add `error = TRUE` in the Rmarkdown's code block's options (not visible in the HTML output), because otherwise compilation of the Rmarkdown document would stop at the error.

If for some reason it made sense, you could explicitly coerce to numeric and then sum:

```
x <- factor(c("mango", "banana", "mango"))
sum(as.numeric(x))
```

```
[1] 5
```

0.20 NA: Missing Values

Missing values in any data type - logical, integer, double, or character - are coded using `NA`.
To check for the presence of `NA` values, use `is.na()`:

```
(x <- c(1.2, 5.3, 4.8, NA, 9.6))
```

```
[1] 1.2 5.3 4.8  NA 9.6
```

```
is.na(x)
```

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

```
(x <- c("mango", "banana", NA, "sugar", "ackee"))
```

```
[1] "mango"  "banana" NA       "sugar"  "ackee"
```

```
is.na(x)
```

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

```
(x <- c(T, T, F, T, F, F, NA))
```

```
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE    NA
```

```
is.na(x)
```

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

`is.na()` works similarly on matrices:

```
x <- matrix(1:20, 5)
x[4, 3] <- NA
is.na(x)
```

```
      [,1]  [,2]  [,3]  [,4]
[1,] FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE
[4,] FALSE FALSE  TRUE FALSE
[5,] FALSE FALSE FALSE FALSE
```

Note that `is.na()` returns a response for each element (i.e. is vectorized) in contrast to `is.numeric()`, `is.logical()`, etc. It makes sense, since the latter are chacking the type of a whole object, while the former is checking individual elements.

`anyNA()` is a very useful function to check if there an any NA values in an object:

```
anyNA(x)
```

```
[1] TRUE
```

Any operations on an NA results in NA

```
x <- c(1.2, 5.3, 4.8, NA, 9.6)
x*2
```

```
[1]  2.4 10.6  9.6   NA 19.2
```

Multiple functions that accept as input an object with multiple values (a vector, a matrix, a data.frame, etc.) will return NA if *any* element is NA:

```
mean(x)
```

```
[1] NA
```

```
median(x)
```

```
[1] NA
```

```
sd(x)
```

```
[1] NA
```

```
min(x)
```

```
[1] NA
```

```
max(x)
```

```
[1] NA
```

```
range(x)
```

```
[1] NA NA
```

First, make sure NA values represent legitimate missing data and not some error. Then, decide how you want to handle it.

In all of the above commands you can pass `na.rm = TRUE` to ignore NA values:

```
mean(x, na.rm = TRUE)
```

```
[1] 5.225
```

```
median(x, na.rm = TRUE)
```

```
[1] 5.05
```

```
sd(x, na.rm = TRUE)
```

```
[1] 3.441293
```

```
min(x, na.rm = TRUE)
```

```
[1] 1.2
```

```
max(x, na.rm = TRUE)
```

```
[1] 9.6
```

```
range(x, na.rm = TRUE)
```

```
[1] 1.2 9.6
```

The chapter on Handling Missing Data describes some approaches to handling missing data in the context of statistics or modeling, commonly supervised learning.

0.21 NaN: Not a number

NaN is a special case of NA and can be the result of undefined mathematical operations:

```
a <- log(-4)
```

```
Warning in log(-4): NaNs produced
```

Note that `class()` returns "numeric":

```
class(a)
```

```
[1] "numeric"
```

To test for `NaN`s, use:

```
is.nan(a)
```

```
[1] TRUE
```

`NaN`s are also `NA`:

```
is.na(a)
```

```
[1] TRUE
```

But the opposite is not true:

```
is.nan(NA)
```

```
[1] FALSE
```

`NaN` can be considered a subtype of `NA`, as such: `is.na(NaN)` is `TRUE`, but `is.nan(NA)` is `FALSE`.

0.22 NULL: the empty object

The `NULL` object represents an empty object.

`NULL` means empty, ***not missing***, and is therefore entirely different from `NA`

`NULL` shows up for example when initializing a list:

```
a <- vector("list", 4)
a
```

```
[[1]]
NULL
```

```
[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

and it can be replaced normally:

```
a[[1]] <- 3
a
```

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

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

0.22.1 Replacing with NULL

You cannot replace one or more elements of a vector/matrix/array with NULL because NULL has length 0 and replacement requires object of equal length:

```
a <- 11:15
a
```

```
[1] 11 12 13 14 15
```

```
a[1] <- NULL
```

```
Error in a[1] <- NULL: replacement has length zero
```

However, in lists and therefore also data frames, replacing an element with NULL ***removes that element***:

```
al <- list(alpha = 11:15,
           beta = rnorm(10),
           gamma = c("mango", "banana", "tangerine"))
al
```

```
$alpha
[1] 11 12 13 14 15

$beta
 [1]  0.34280670  1.07661321 -0.80748006 -0.30374722 -0.02766314 -0.216520
 [7]  0.55670508  0.55847086 -2.35314815  0.90843017

$gamma
[1] "mango"     "banana"    "tangerine"
```

```
al[[2]] <- NULL
al
```

```
$alpha
[1] 11 12 13 14 15

$gamma
[1] "mango"     "banana"    "tangerine"
```

Finally, `NULL` is often used as the default value in a function's argument. The function definition must then determine what the default behavior/value should be.

Data Structures

There are 5 main data structures in R:

- **Vector**: 1-dimensional; homogeneous collection
- **Matrix**: 2-dimensional; homogeneous collection
- **Array**: N-dimensional; homogeneous collection
- **List**: 1-dimensional, but can be nested; heterogeneous collection
- **Data frame**: 2-dimensional: A special type of list; heterogeneous collection of columns

Homogeneous vs. hetereogeneous refers to the kind of data types (integer, double, character, logical, factor, etc.) that a structure can hold. This means a matrix can hold only numbers or only characters, but a data frame can hold different types in different columns. That is why data frames are very popular data structure for statistical work.

Check object class with `class()` and/or `str()`.

0.23 Initialize - coerce - test (structures)

The following summary table lists the functions to *initialize*, *coerce* (=convert), and *test* the core data structures, which are shown in more detail in the following paragraphs:

Initialize	Coerce	Test
`vector(n)`	`as.vector(x)`	`is.vector(x)`
`matrix(n)`	`as.matrix(x)`	`is.matrix(x)`
`array(n)`	`as.array(x)`	`is.array(x)`
`list(n)`	`as.list(x)`	`is.list(x)`
`data.frame(n)`	`as.data.frame(x)`	`is.data.frame(x)`

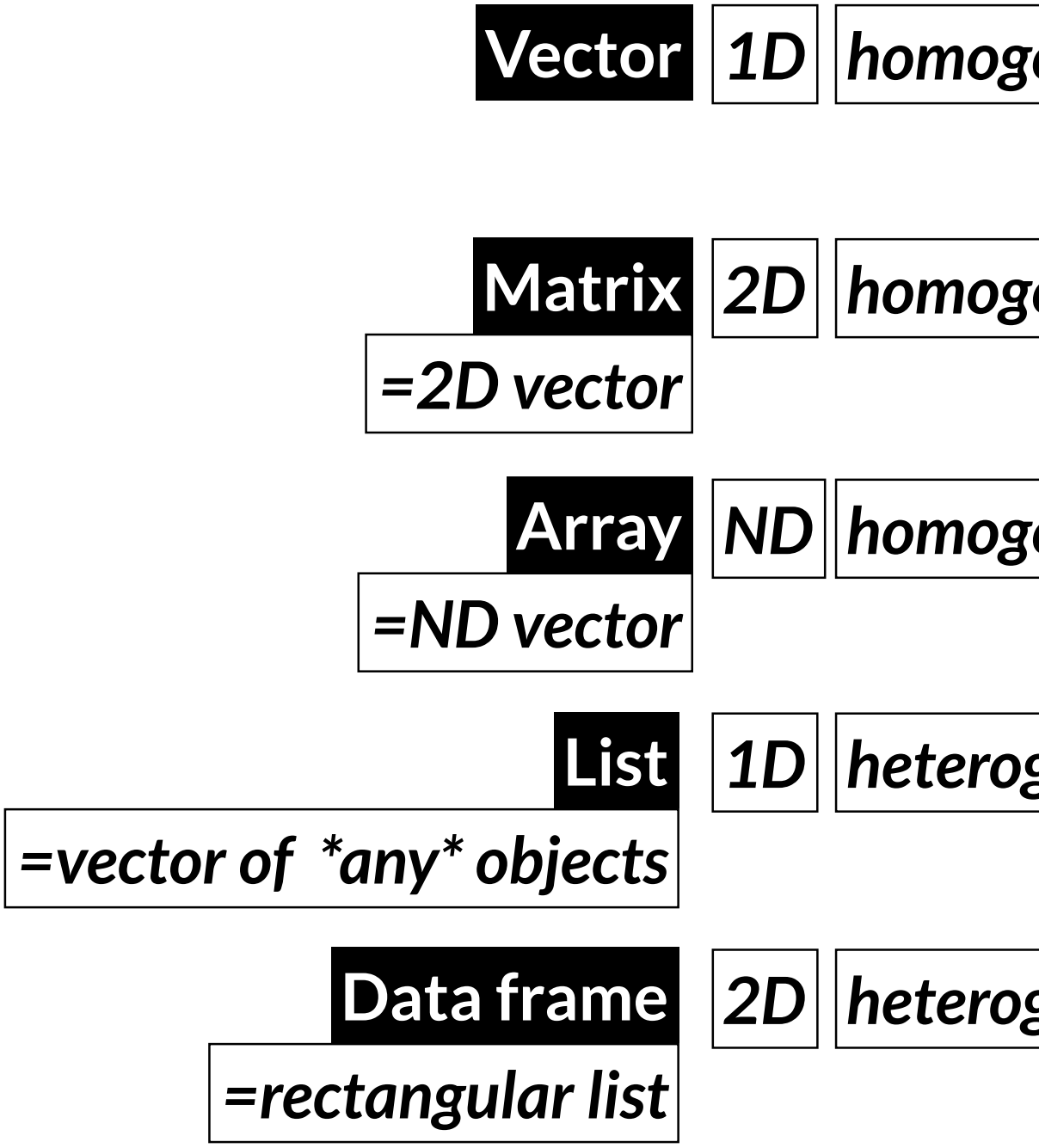


Figure 1: R Data Structure summary - Best to read through this chapter first and then refer back to this figure

0.24 Vectors

A vector is the basic structure that contains data in R. Other structures that contain data are made up of one or more vectors.

```
(x <- c(1, 3, 5, 7))
```

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

```
class(x)
```

```
[1] "numeric"
```

```
typeof(x)
```

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

A vector has `length()` but no `dim()`:

```
length(x)
```

```
[1] 4
```

```
dim(x)
```

```
NULL
```

```
(x2 <- 1:10)
```

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

```
(x3 <- rnorm(10))
```

```
 [1] -0.58371358 -0.89986818 -1.21984954 -1.69780709  0.67515971 -0.46769460
 [7]  0.08998513 -0.47471458 -2.61724883 -0.23993454
```

```
(x4 <- seq(0, 1, .1))
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
seq(10)
```

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

```
(x5 <- sample(seq(100), 20))
```

```
 [1] 74 88 62  3 58 86 50  7 21  6 80 44 14 59 63  9 65  5  1 89
```

0.24.1 Generating sequences with `seq()`

1. from, to, by

```
seq(1, 10, .5)
```

```
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
[16]  8.5  9.0  9.5 10.0
```

2. 1:n

```
(seq(12))
```

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

```
# or
(seq_len(12))
```

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

```
# is same as
1:12
```

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

3. Along the length of another object

```
seq_along(iris)
```

```
[1] 1 2 3 4 5
```

```
1:ncol(iris)
```

```
[1] 1 2 3 4 5
```

4. `from`, `to` with length `n`

```
seq(-5, 12, length.out = 11)
```

```
 [1] -5.0 -3.3 -1.6  0.1  1.8  3.5  5.2  6.9  8.6 10.3 12.0
```

0.24.2 Initializing a vector

```
x <- vector(length = 10)
x <- vector("numeric", 10)
x <- vector("list", 10)
```

0.25 Matrices

A matrix is a **vector with 2 dimensions**.

To create a matrix, you pass a vector to the `matrix()` command and specify number of rows using `nrow` and/or number of columns using `ncol`:

```
x <- matrix(sample(seq(1000), 30),
            nrow = 10, ncol = 3)
x
```

```
      [,1] [,2] [,3]
 [1,]  789  753  375
 [2,]  185  162   64
 [3,]  813  583  275
 [4,]  166  850  777
 [5,]    3  159   34
 [6,]  870  737  436
 [7,]  308   59  507
 [8,]  550   83  338
 [9,]  837  427  426
[10,]  417  421  822
```

```
class(x)
```

```
[1] "matrix" "array"
```

A matrix has length (`length(x)`) equal to the number of all (i, j) elements or nrow * ncol (if `i` is the row index and `j` is the column index) and dimensions (`dim(x)`) as expected:

```
length(x)
```

```
[1] 30
```

```
dim(x)
```

```
[1] 10  3
```

```
nrow(x)
```

```
[1] 10
```

```
ncol(x)
```

```
[1] 3
```

0.25.1 Construct by row or by column

By default, vectors are constructed by column (byrow = FALSE)

```
x <- matrix(1:20, nrow = 10, ncol = 2, byrow = FALSE)
x
```

```
      [,1] [,2]
 [1,]    1   11
 [2,]    2   12
 [3,]    3   13
 [4,]    4   14
 [5,]    5   15
 [6,]    6   16
 [7,]    7   17
```

```
 [8,]    8   18
 [9,]    9   19
[10,]   10   20
```

```
x <- matrix(1:20, nrow = 10, ncol = 2, byrow = TRUE)
x
```

```
      [,1] [,2]
 [1,]    1    2
 [2,]    3    4
 [3,]    5    6
 [4,]    7    8
 [5,]    9   10
 [6,]   11   12
 [7,]   13   14
 [8,]   15   16
 [9,]   17   18
[10,]   19   20
```

0.25.2 Initialize a matrix

```
(x <- matrix(NA, nrow = 6, ncol = 4))
```

```
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
[4,]   NA   NA   NA   NA
[5,]   NA   NA   NA   NA
[6,]   NA   NA   NA   NA
```

```
(x <- matrix(0, nrow = 6, ncol = 4))
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
[5,]    0    0    0    0
[6,]    0    0    0    0
```

0.25.3 Bind vectors by column or by row

Use `cbind` ("column-bind") to convert a set of input vectors to columns of a **matrix**. The vectors must be of the same length:

```
x <- cbind(1:10, 11:20, 41:50)
x
```

```
      [,1] [,2] [,3]
 [1,]    1   11   41
 [2,]    2   12   42
 [3,]    3   13   43
 [4,]    4   14   44
 [5,]    5   15   45
 [6,]    6   16   46
 [7,]    7   17   47
 [8,]    8   18   48
 [9,]    9   19   49
[10,]   10   20   50
```

```
class(x)
```

```
[1] "matrix" "array"
```

Similarly, you can use `rbind` ("row-bind") to convert a set of input vectors to rows of a **matrix**. The vectors again must be of the same length:

```
x <- rbind(1:10, 11:20, 41:50)
x
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
[3,]   41   42   43   44   45   46   47   48   49    50
```

```
class(x)
```

```
[1] "matrix" "array"
```

0.25.4 Combine matrices

`cbind()` and `rbind()` can be used to combine two or more matrices together - or vector and matrices:

```
cbind(matrix(1, 5, 2), matrix(2, 5, 4))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    2    2    2    2
[2,]    1    1    2    2    2    2
[3,]    1    1    2    2    2    2
[4,]    1    1    2    2    2    2
[5,]    1    1    2    2    2    2
```

0.26 Arrays

Arrays are **vectors with dimensions**.
You can have 1D, 2D or any-D, i.e. ND arrays.

0.26.1 1D array

A 1D array is just like a vector but of class `array` and with `dim(x)` equal to `length(x)` (remember, vectors have only `length(x)` and undefined `dim(x)`):

```
x <- 1:10
xa <- array(1:10, dim = 10)
class(x)
```

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

```
is.vector(x)
```

```
[1] TRUE
```

```
length(x)
```

```
[1] 10
```

```
dim(x)
```

```
NULL
```

```
class(xa)
```

```
[1] "array"
```

```
is.vector(xa)
```

```
[1] FALSE
```

```
length(xa)
```

```
[1] 10
```

```
dim(xa)
```

```
[1] 10
```

It is quite unlikely you will need to use a 1D array instead of a vector.

0.26.2 2D array

A 2D array is a matrix:

```
x <- array(1:40, dim = c(10, 4))
class(x)
```

```
[1] "matrix" "array"
```

```
dim(x)
```

```
[1] 10  4
```

0.26.3 ND array

You can build an N-dimensional array:

```
(x <- array(1:60, dim = c(5, 4, 3)))
```

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
```

```
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20

, , 2

     [,1] [,2] [,3] [,4]
[1,]   21   26   31   36
[2,]   22   27   32   37
[3,]   23   28   33   38
[4,]   24   29   34   39
[5,]   25   30   35   40

, , 3

     [,1] [,2] [,3] [,4]
[1,]   41   46   51   56
[2,]   42   47   52   57
[3,]   43   48   53   58
[4,]   44   49   54   59
[5,]   45   50   55   60
```

```
class(x)
```

```
[1] "array"
```

You can provide names for each dimensions using the `dimnames` argument. It accepts a list where each elements is a character vector of legth equal to the dimension length. Using the same example as above, we pass three character vector of length 5, 4, and 3 to match the length of the dimensions:

```
x <- array(1:60,
           dim = c(5, 4, 3),
           dimnames = list(letters[1:5],
                           c("alpha", "beta", "gamma", "delta"),
                           c("x", "y", "z")))
```

3D arrays can be used to represent color images. Here, just for fun, we use `rasterImage` to show how you would visualize such an image:

```
x <- array(sample(1:255, 432, TRUE), dim = c(12, 12, 3))
par("pty")
```

```
[1] "m"
```

```
par(pty = "s")
plot(NULL, NULL,
     xlim = c(0, 100), ylim = c(0, 100),
     axes = F, ann = F, pty = "s")
rasterImage(x/255, 0, 0, 100, 100)
```

0.27 Lists

To define a list, we use `list()` to pass any number of objects.
If these objects are passed as named arguments, the names will rename as element names:

```
x <- list(one = 1:4,
          two = sample(seq(0, 100, .1), 10),
          three = c("mango", "banana", "tangerine"),
          four = median)
class(x)
```

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

```
str(x)
```

```
List of 4
```

```
 $ one  : int [1:4] 1 2 3 4
$ two  : num [1:10] 42 50.9 53.1 11.2 0 47.9 55.4 56.8 64.2 55.2
 $ three: chr [1:3] "mango" "banana" "tangerine"
 $ four :function (x, na.rm = FALSE, ... )
```

0.27.1 Nested lists

Since each element can be any object at all, it is simple to build a nested list:

```
x <- list(alpha = letters[sample(26, 4)],
          beta = sample(12),
          gamma = list(i = rnorm(10),
                       j = runif(10),
                       j = seq(0, 1000, length.out = 10)))
x
```

```
$alpha
[1] "r" "z" "m" "t"

$beta
 [1]  2 10  3  5  1  6  8  9  7 11  4 12

$gamma
$gamma$i
 [1] -0.6668076  2.0549929 -0.4238642 -0.4842669 -0.8476532 -0.5907072
 [7]  0.7108277 -1.8806911 -1.1756815  0.6060033

$gamma$j
 [1] 0.22827651 0.97811864 0.04831228 0.79990941 0.65935084 0.43072323
 [7] 0.63118740 0.65953034 0.06062042 0.09970150

$gamma$j
 [1]   0.0000 111.1111 222.2222 333.3333 444.4444 555.5556 666.6667
 [8] 777.7778 888.8889 1000.0000
```

0.27.2 Initialize a list

```
x <- vector("list", 4)
x
```

```
[[1]]
NULL
```

```
[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

0.27.3 Combine lists

You can combine lists with `c()` (just like vectors):

```
l1 <- list(q = 11:14, r = letters[11:14])
l2 <- list(s = LETTERS[21:24], t = 100:97)
(x <- c(l1, l2))
```

```
$q
[1] 11 12 13 14

$r
[1] "k" "l" "m" "n"

$s
[1] "U" "V" "W" "X"

$t
[1] 100  99  98  97
```

```
length(x)
```

```
[1] 4
```

0.27.4 Mixing types with `c()`

It's best to use `c()` to either combine elements of the same type into a vector, or to combine lists, otherwise you must inspect the outcome to be certain it was as intended.

As we've seen, if all arguments passed to `c()` are of a single type, you get a vector of that type:

```
(x <- c(12.9, 94.67, 23.74, 46.901))
```

```
[1] 12.900 94.670 23.740 46.901
```

```
class(x)
```

```
[1] "numeric"
```

If arguments passed to `c()` are a mix of numeric and character, they all get ***coerced to character***.

```
(x <- c(23.54, "mango", "banana", 75))
```

```
[1] "23.54"  "mango"  "banana" "75"
```

```
class(x)
```

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

If you pass more types of objects (which cannot be coerced to character) you get a list, since it is the only structure that can support all of them together:

```
(x <- c(42, mean, "potatoes"))
```

```
[[1]]
[1] 42

[[2]]
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fe8e21d9350>
<environment: namespace:base>

[[3]]
[1] "potatoes"
```

```
class(x)
```

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

Other than concatenating vectors of the same type or lists into a larger list, it probably best to avoid using `c()` and directly constructing the object you want using, e.g. `list()`.

0.28 Data frames

A data frames is a **special type of list** where each element has the same length and forms a column, resulting in a 2D structure. Unlike matrices, each column can contain a different data type.

```
x <- data.frame(Feat_1 = 1:5,
                Feat_2 = rnorm(5),
                Feat_3 = paste0("rnd_", sample(seq(100), 5)))
x
```

```
  Feat_1     Feat_2 Feat_3
1      1  0.1139031 rnd_32
2      2  0.8674226 rnd_21
3      3 -1.1174872 rnd_15
4      4 -0.4318017 rnd_10
5      5  0.4467293 rnd_46
```

```
class(x)
```

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

```
str(x)
```

```
'data.frame':   5 obs. of  3 variables:
 $ Feat_1: int  1 2 3 4 5
 $ Feat_2: num  0.114 0.867 -1.117 -0.432 0.447
 $ Feat_3: chr  "rnd_32" "rnd_21" "rnd_15" "rnd_10" ...
```

```
class(x$Feat_1)
```

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

```
mat <- matrix(1:100, 10)
length(mat)
```

```
[1] 100
```

```
df <- as.data.frame(mat)
length(df)
```

```
[1] 10
```

0.29 Attributes

R objects may have some builtin attributes but you can add arbitrary attributes to any R object. These are used to store additional information, sometimes called metadata.

0.29.1 Print all attributes

To print an object's attributes, use `attributes`:

```
attributes(iris)
```

```
$names
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

$class
[1] "data.frame"

$row.names
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150
```

This returns a named list. In this case we got names, class, and row.names of the iris data frame.

0.29.2 Get or set specific attributes

You can assign new attributes using `attr`:

```
(x <- c(1:10))
```

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

```
attr(x, "name") <- "Very special vector"
```

Printing the vector after adding a new attribute, prints the attribute name and value underneath the vector itself:

```
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"name")
[1] "Very special vector"
```

Our trusty `str` function will print attributes as well

```
str(x)
```

```
 int [1:10] 1 2 3 4 5 6 7 8 9 10
 - attr(*, "name")= chr "Very special vector"
```

0.29.2.1 A matrix is a vector - a closer look

Let's see how a matrix is literally just a vector with assigned dimensions.
Start with a vector of length 20:

```
x <- 1:20
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

The vector has no attributes - yet:

```
attributes(x)
```

```
NULL
```

To convert to a matrix, we would normally pass our vector to the `matrix()` function and define number of rows and/or columns:

```
xm <- matrix(x, 5)
xm
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

```
attributes(xm)
```

```
$dim
[1] 5 4
```

Just for demonstration, let's instead directly add a dimension attribute to our vector:

```
attr(x, "dim") <- c(5, 4)
x
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

```
class(x)
```

```
[1] "matrix" "array"
```

Just like that, we have a matrix.

0.29.3 Common builtin attributes

Vectors can have named elements. A new vector has no names, but you can add them:

```
x <- rnorm(10)
names(x)
```

```
NULL
```

```
names(x) <- paste0("Value", seq(x))
x
```

```
    Value1     Value2     Value3     Value4     Value5     Value6 
-0.262001868  0.914904183 -1.721527420 -1.599504301  0.654998648  0.525585
      Value7       Value8       Value9      Value10 
 0.938582676 -0.121320715 -0.519416220 -0.006049031
```

Matrices and data frames can have column names (`colnames`) and row names (`rownames`):

```
x <- matrix(1:15, 5)
colnames(x)
```

```
NULL
```

```
rownames(x)
```

```
NULL
```

```
colnames(x) <- paste0("Feature", seq(3))
rownames(x) <- paste0("Case", seq(5))
x
```

```
      Feature1 Feature2 Feature3
Case1        1        6       11
Case2        2        7       12
Case3        3        8       13
Case4        4        9       14
Case5        5       10       15
```

Lists are vectors so they have `names`. These can be defined when a list is created using the name-value pairs or added/changed at any time.

```
x <- list(HospitalName = "CaliforniaGeneral",
          ParticipatingDepartments = c("Neurology", "Psychiatry", "Neuro
          PatientIDs = 1001:1253)
names(x)
```

```
[1] "HospitalName"           "ParticipatingDepartments"
[3] "PatientIDs"
```

Add/Change names:

```
names(x) <- c("Hospital", "Departments", "PIDs")
x
```

```
$Hospital
[1] "CaliforniaGeneral"

$Departments
[1] "Neurology"    "Psychiatry"   "Neurosurgery"

$PIDs
  [1] 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015
 [16] 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030
 [31] 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045
 [46] 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060
 [61] 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075
 [76] 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090
 [91] 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105
[106] 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120
[121] 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135
[136] 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150
[151] 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165
[166] 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180
[181] 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195
[196] 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210
[211] 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225
[226] 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240
[241] 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253
```

Remember that data a frame is a special type of list. Therefore in data frames `colnames` and `names` are equivalent:

```
colnames(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Note: As we saw, matrices have `colnames` and `rownames`. Using `names` on a matrix will assign names to *individual elements*, as if it was a long vector - this is not usually very useful.

Factors

Factors in R are used to store **categorical variables** and therefore have many important uses in statistics / data science / machine learning.

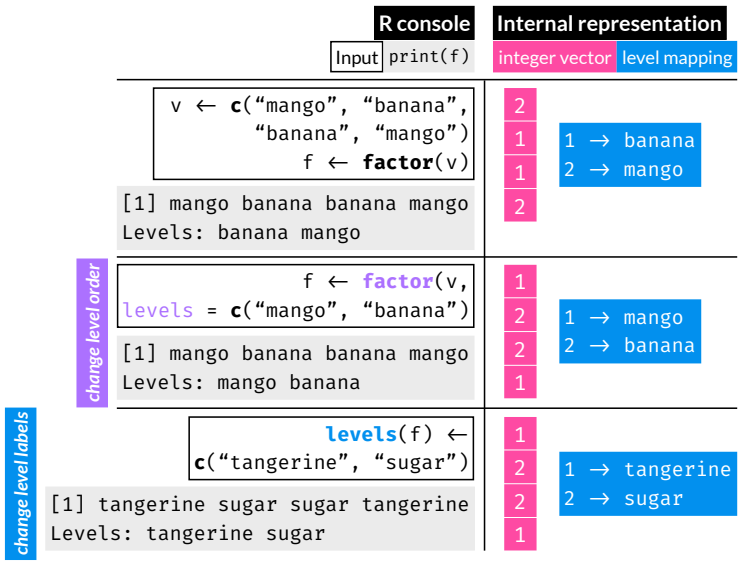


Figure 2: Factors in R - Best to read through this chapter first and then refer back to this figure

You can create a factor by passing a numeric or character vector to `factor()` or `as.factor()`.
The difference is that `as.factor()` does not accept any arguments while `factor()` does - and they can be very important. More on this right below.

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

```
[1] a c d b a a d c
Levels: a b c d
```

```
class(xf)
```

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

```
xftoo <- as.factor(x)
xftoo
```

```
[1] a c d b a a d c
Levels: a b c d
```

```
class(xftoo)
```

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

A factor contains three crucial pieces of information:

1. The underlying **integer vector**
2. The **mapping of integers to labels**
3. Whether the factor is **ordered**

Let's unpack these.

Begin with a simple factor:

```
x <- factor(c("female", "female", "female", "male", "male"))
x
```

```
[1] female female female male   male
Levels: female male
```

Internally, the command sees there are two distinct labels, female and male, and defaults to assigning integer numbers alphabetically, in this case female has been mapped to '1' and male to '2'.

Printing a factor prints the vector of labels followed by the levels, i.e. the unique labels.

0.30 The underlying integer vector

Each level is assigned an integer. (Internally, this is the "data" that forms the elements of a factor vector). You don't see these integers unless you convert the factor to numeric

(`as.numeric()`) or look at the (truncated) output of `str()`

```
as.numeric(x)
```

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

0.31 The mapping of integers to labels

This defines which integer is mapped to which label, i.e. whether 1 is mapped to male or female. You can store the same information regardless which one you choose to call 1 and 2.

To get the mapping you can use `levels()`. It prints the labels in order:

```
levels(x)
```

```
[1] "female" "male"
```

Again, this means that female is mapped to 1 and male is mapped to 2.

```
str(x)
```

```
 Factor w/ 2 levels "female","male": 1 1 1 2 2
```

The above tells you that x is a factor,
it has two levels labeled as "female" and "male", in that order, i.e. female is level 1 and male is level 2.
The last part shows that the first five elements (in this case the whole vector) consists of three elements of level 1 (female) followed by 2 elements of level 2 (male)

0.31.1 Setting new level labels

You can use the `levels()` command with an assignment to assign new labels to a factor (same syntax to how you use `rownames()` or `colnames()` to assign new row or column names to a matrix or data frame)

```
xf <- factor(sample(c("patient_status_positive", "patient_status_negative"), 10,
             levels = c("patient_status_positive", "patient_status_negative"))
xf
```

```
 [1] patient_status_negative patient_status_negative patient_status_negative
 [4] patient_status_positive patient_status_negative patient_status_negative
```

```
 [7] patient_status_negative patient_status_negative patient_status_posit
[10] patient_status_negative
Levels: patient_status_positive patient_status_negative
```

```
levels(xf)
```

```
[1] "patient_status_positive" "patient_status_negative"
```

```
levels(xf) <- c("positive", "negative")
xf
```

```
 [1] negative negative negative positive negative negative negative negativ
 [9] positive negative
Levels: positive negative
```

0.31.2 Defining the mapping of labels to integers

If you want to define the mapping of labels to their integer representation (and not default to them sorted alphabeticaly), you use the `levels` arguments of the `factor()` function.

The vector passed to the `levels` arguments must include at least all unique values passed to `factor()`, otherwise you will get `NA` values

Without defining `levels` they are assigned alphabeticaly:

```
x <- factor(c("alpha", "alpha", "gamma", "delta", "delta"))
x
```

```
[1] alpha alpha gamma delta delta
Levels: alpha delta gamma
```

Define `levels`:

```
x <- factor(c("alpha", "alpha", "gamma", "delta", "delta"),
 levels = c("alpha", "gamma", "delta"))
x
```

```
[1] alpha alpha gamma delta delta
Levels: alpha gamma delta
```

The `table` command has a number of useful applications, in it simplest form, it tabulates number of elements with each unique value found in a vector:

```
table(x)
```

```
x
alpha gamma delta
    2     1     2
```

If you forget (or choose to exclude) a level, all occurences are replaced by NA:

```
x <- factor(c("alpha", "alpha", "gamma", "delta", "delta"),
 levels = c("alpha", "gamma"))
x
```

```
[1] alpha alpha gamma <NA>  <NA>
Levels: alpha gamma
```

If you know that more levels exist, even if no examples are present in your sample, you can includes these extra levels:

```
x <- factor(c("alpha", "alpha", "gamma", "delta", "delta"),
 levels = c("alpha", "beta", "gamma", "delta"))
x
```

```
[1] alpha alpha gamma delta delta
Levels: alpha beta gamma delta
```

```
table(x)
```

```
x
alpha  beta gamma delta
    2     0     1     2
```

0.32 Is the factor ordered

We looked at how you can define the order of levels using the `levels` argument in `factor()`, which affects the integer mapping to each label.

This can affect how some applications treat the different levels.

On top of the order of the mapping, you can further define if there is a *quantitative relationship* among levels of the form `level 1 < level 2 < ... < level n`. This, in turn, can affect how the factor is treated by some functions, like some functions that fit statistical models.

All factors' levels appear in some order or other.

An **ordered factor** indicates that its levels have a quantitative relationship of the form `level 1 < level 2 < ... < level n`.

First an unordered factor:

```
dat <- sample(c("small", "medium", "large"), 10, TRUE)
x <- factor(dat)
x
```

```
 [1] medium small  large  large  small  small  small  medium large  large
Levels: large medium small
```

To make the above into an ordered factor, we need to define the order of the levels with the `levels` arguments and also specify that it is ordered with the `ordered` argument:

```
x <- factor(dat,
            levels = c("small", "medium", "large"),
            ordered = TRUE)
x
```

```
 [1] medium small  large  large  small  small  small  medium large  large
Levels: small < medium < large
```

Note how the levels now include the < sign between levels to indicate the ordering.

0.33 Change (order of levels) or (labels)

We've seen how to create a factor with defined order of levels and how to change level labels already. Because these are prone to serious accidents, let's look at them again, together.

To change the order of levels of an existing factor use `factor()`:

```
x <- factor(c("target", "target", "control", "control", "control"))
x
```

```
[1] target  target  control control control
Levels: control target
```

Change the order so that target is first (i.e. corresponds to 1:

```
x <- factor(x, levels = c("target", "control"))
x
```

```
[1] target  target  control control control
Levels: target control
```

To change the labels of the levels use `levels()`:

```
x
```

```
[1] target  target  control control control
Levels: target control
```

```
levels(x) <- c("hit", "decoy")
x
```

```
[1] hit   hit   decoy decoy decoy
Levels: hit decoy
```

Changing the levels of a factor with `levels()` does not change the internal integer representation but changes every elements labels

0.34 Fatal error to avoid

Example scenario: You receive a dataset for classification where the outcome is a factor of 1s and 0s:

```
outcome <- factor(c(1, 1, 0, 0, 0, 1, 0))
outcome
```

```
[1] 1 1 0 0 0 1 0
Levels: 0 1
```

Some classification procedures expect the first level to be the 'positive' outcome, so you decide to reorder the levels.

You mistakenly use `levels()` instead of `factor(x, levels=c( ... ))` hoping to achieve this.

You end up flipping all the outcome values.

```
levels(outcome) <- c("1", "0")
outcome
```

```
[1] 0 0 1 1 1 0 1
Levels: 1 0
```

All zeros became ones and ones became zeros.

You don't notice because how would you.

Your model does the exact opposite of what you intended.

-> Don't ever do this.

0.35 Factor to numeric

While it often makes sense to have factors with words for labels, they can be any character and that includes numbers (i.e. numbers which are treated as labels)

```
f <- factor(c(3, 7, 7, 9, 3, 3, 9))
f
```

```
[1] 3 7 7 9 3 3 9
Levels: 3 7 9
```

This behaves just like any other factor with all the rules we learned above.

There is a very easy trap to fall into, if you ever decide to convert such a factor to numeric.

The first thing that usually comes to mind is to use `as.numeric()`.

```
# !don't do this!
as.numeric(f)
```

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

But! We already know this will return the integer index, it will not return the labels as numbers.

By understanding the internal representation of the factor, i.e. that a factor is an integer vector indexing a set of labels, you can convert labels to numeric exactly by indexing the set of labels:

```
levels(f)[f]
```

```
[1] "3" "7" "7" "9" "3" "3" "9"
```

The above suggests that used as an index within the brackets, `f` is coerced to integer, therefore to understand the above:

```
levels(f)
```

```
[1] "3" "7" "9"
```

```
levels(f)[as.integer(f)]
```

```
[1] "3" "7" "7" "9" "3" "3" "9"
```

```
# same as
levels(f)[f]
```

```
[1] "3" "7" "7" "9" "3" "3" "9"
```

A different way around this that may be less confusing is to simply convert the factor to character and then to numeric:

```
as.numeric(as.character(f))
```

```
[1] 3 7 7 9 3 3 9
```

0.36 Summary

- Factors in R are **integer vectors** with labels.
- A factor's internal integer values range from 1 to the number of levels, i.e. categories.
- Each integer corresponds to a label.
- Use `factor(levels = c( ... ))` to order levels
- Use `levels(x)` to change levels' labels

Indexing - Subsetting - Slicing

An index is used to select elements of a vector, matrix, array, list or data frame.
You can select (or exclude) one or multiple elements at a time.

An index is one of two types:

- logical index: for each elements in an object specify TRUE if you want to include it, or FALSE to exclude it from the selection.
- integer index: define the position of elements to select.

The main indexing operator in R is the square bracket `[`.
Logical indexes are usually created as the output of a logical operation, e.g. an elemn-twise comparison.
Integer indexing in R is 1-based, meaning the first item of a vector is in position 1.
(If you are wondering why we even have to mention this, know that many programming languages use 0-based indexing[8])

0.37 Vectors

```
x <- 15:24
x
```

```
 [1] 15 16 17 18 19 20 21 22 23 24
```

Get the 5th element of a vector (integer index):

```
x[5]
```

```
[1] 19
```

[8] https://en.wikipedia.org/wiki/Zero-based_numbering#Computer_programming

Get elements 6 through 9 of the same vector (integer index):

```
x[6:9]
```

```
[1] 20 21 22 23
```

Select elements with value greater than 19 (logical index):

```
x[x > 19]
```

```
[1] 20 21 22 23 24
```

0.38 Matrices

Reminder:

- A matrix is a 2D vector and contains elements of one type only (numeric, integer, character, factor).
- A data frame is a 2D list and each column can contain different type of data.

To index a 2D structure, whether a matrix or data frame, we use the form `[row, column]`
The following indexing operations are therefore the same whether applied on a matrix or a data frame.

```
mat <- matrix(1:40, 10)
colnames(mat) <- paste0("Feature_", seq(ncol(mat)))
rownames(mat) <- paste0("Row_", seq(nrow(mat)))
mat
```

```
       Feature_1 Feature_2 Feature_3 Feature_4
Row_1          1        11        21        31
Row_2          2        12        22        32
Row_3          3        13        23        33
Row_4          4        14        24        34
Row_5          5        15        25        35
Row_6          6        16        26        36
Row_7          7        17        27        37
Row_8          8        18        28        38
Row_9          9        19        29        39
Row_10        10        20        30        40
```

```
df <- as.data.frame(mat)
df
```

```
       Feature_1 Feature_2 Feature_3 Feature_4
Row_1          1        11        21        31
Row_2          2        12        22        32
Row_3          3        13        23        33
Row_4          4        14        24        34
Row_5          5        15        25        35
Row_6          6        16        26        36
Row_7          7        17        27        37
Row_8          8        18        28        38
Row_9          9        19        29        39
Row_10        10        20        30        40
```

To get the contents of the fifth row, second column:

```
mat[5, 2]
```

```
[1] 15
```

```
df[5, 2]
```

```
[1] 15
```

If you want to select an entire row or an entire column, you leave the row or column index blank, but - necessarily - use a comma:

Get the first row:

```
mat[1, ]
```

```
Feature_1 Feature_2 Feature_3 Feature_4
        1        11        21        31
```

Get the second column:

```
mat[, 2]
```

```
 Row_1  Row_2  Row_3  Row_4  Row_5  Row_6  Row_7  Row_8  Row_9 Row_10
    11     12     13     14     15     16     17     18     19     20
```

Note that colnames and rownames where added to the matrix above for convenience - if they are absent, the labels are not shown above each element.

0.38.1 Range of rows and columns

You can define ranges for both rows and columns:

```
mat[6:7, 2:4]
```

```
      Feature_2 Feature_3 Feature_4
Row_6        16        26        36
Row_7        17        27        37
```

You can return rows and/or columns reversed too if desired:

```
mat[7:6, 4:2]
```

```
      Feature_4 Feature_3 Feature_2
Row_7        37        27        17
Row_6        36        26        16
```

Or use vectors to specify of any rows and columns:

```
mat[c(2, 4, 7), c(1, 4, 3)]
```

```
      Feature_1 Feature_4 Feature_3
Row_2         2        32        22
Row_4         4        34        24
Row_7         7        37        27
```

0.38.2 Matrix of indexes

This is quite less common, but potentially useful. It allows you to specify a series of individual `[i, j]` indexes (i.e. is a way to select multiple non-contiguous elements)

```
idm <- matrix(c(2, 4, 7, 4, 3, 1), 3)
idm
```

```
     [,1] [,2]
[1,]    2    4
[2,]    4    3
[3,]    7    1
```

An n-by-2 matrix can be used to index as a length n vector of `[row, colum]` indexes. Therefore, the above matrix, will return elements `[2, 4]`, `[4, 3]`, `[7, 1]`:

```
mat[idm]
```

```
[1] 32 24  7
```

0.38.3 Logical index

Select all rows with values greater than 15 on the second column:

The logical index for this operation is:

```
mat[, 2] > 15
```

```
 Row_1  Row_2  Row_3  Row_4  Row_5  Row_6  Row_7  Row_8  Row_9 Row_10
 FALSE  FALSE  FALSE  FALSE  FALSE   TRUE   TRUE   TRUE   TRUE   TRUE
```

It can be used directly to index the matrix:

```
mat[mat[, 2] > 15, ]
```

```
       Feature_1 Feature_2 Feature_3 Feature_4
Row_6          6        16        26        36
Row_7          7        17        27        37
Row_8          8        18        28        38
Row_9          9        19        29        39
Row_10        10        20        30        40
```

0.39 Lists

Reminder: A list can contain elements of different class and of different length:

```
(x <- list(one = 1:4,
           two = sample(seq(0, 100, .1), 10),
           three = c("mango", "banana", "tangerine"),
           four = median))
```

```
$one
[1] 1 2 3 4

$two
 [1] 36.8 58.7  2.2 56.3  5.4 66.5 94.8  7.5 74.5 83.0
```

```
$three
[1] "mango"     "banana"    "tangerine"

$four
function (x, na.rm = FALSE, ...)
UseMethod("median")
<bytecode: 0x7fe1e20ae190>
<environment: namespace:stats>
```

You can access a list element with:

- $ followed by name of the element (therefore only works if elements are named)
- using double brackets [[with either name or integer index

To access the third element:

```
x$three
```

```
[1] "mango"     "banana"    "tangerine"
```

```
x[["three"]]
```

```
[1] "mango"     "banana"    "tangerine"
```

```
class(x[["three"]])
```

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

```
x[[3]]
```

```
[1] "mango"     "banana"    "tangerine"
```

To access an element with a name or integer index stored in a variable, only the bracket notation works - therefore programmatically you would always use double brackets to access different elements:

```
idi <- 3
idc <- "three"
x[[idi]]
```

```
[1] "mango"     "banana"    "tangerine"
```

```
x[[idc]]
```

```
[1] "mango"     "banana"    "tangerine"
```

$ or [[return an element.
In contrast, single bracket [indexing of a list returns a pruned list:

```
x[[idi]]
```

```
[1] "mango"     "banana"    "tangerine"
```

```
class(x[[idi]])
```

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

vs.

```
x[idi]
```

```
$three
[1] "mango"     "banana"    "tangerine"
```

```
class(x[idi])
```

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

Extract multiple list elements with single brackets, as expected:

```
x[2:3]
```

```
$two
 [1] 36.8 58.7  2.2 56.3  5.4 66.5 94.8  7.5 74.5 83.0

$three
[1] "mango"     "banana"    "tangerine"
```

```
class(x[2:3])
```

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

Beware (confusing) recursive indexing.
(This is probably rarely used).
Unlike in the single brackets example above, colon notation in double brackets accesses elements recursively at the given position.
The following extracts the 3rd element of the 2nd element of the list:

```
x[[2:3]]
```

```
[1] 2.2
```

You can convert a list to one lone vector containing all the individual components of the original list using `unlist()`. Notice how names are automatically created based on the original structure:

```
(x <- list(alpha = sample(seq(100), 10),
           beta = sample(seq(100), 10),
           gamma = sample(seq(100), 10)))
```

```
$alpha
 [1] 86 77  1 38 36 37 21 44 97 59

$beta
 [1]  7 59 58 64 88 85 34 41 51 87

$gamma
 [1] 41 76 18  5 40 50 28 92 65 93
```

```
unlist(x)
```

```
 alpha1  alpha2  alpha3  alpha4  alpha5  alpha6  alpha7  alpha8  alpha9 alpha1
    86      77       1      38      36      37      21      44      97     59
  beta1   beta2   beta3   beta4   beta5   beta6   beta7   beta8   beta9  beta10
     7      59      58      64      88      85      34      41      51     87
 gamma1  gamma2  gamma3  gamma4  gamma5  gamma6  gamma7  gamma8  gamma9 gamma1
    41      76      18       5      40      50      28      92      65     93
```

0.39.1 Logical index

We can use a logical index on a list as well:

```
x[c(T, F, T, F)]
```

```
$alpha
 [1] 86 77  1 38 36 37 21 44 97 59

$gamma
 [1] 41 76 18  5 40 50 28 92 65 93
```

0.40 Data frames

We've already seen above that a data frame can be indexed in the same ways as a matrix.
At the same time, we know that a data frame is a rectangular list. Like a list, its elements are vectors of any type (integer, double, character, factor, and more) but, unlike a list, they have to be of the same length. A data frame can also be indexed the same way as a list.
Similar to indexing a list, notice that some methods return a smaller data frame, while others return vectors.

You can index a data frame using all the ways you can index a list and all the ways you can index a matrix.

Let's create a simple data frame:

```
x <- data.frame(Feat_1 = 21:25,
                Feat_2 = rnorm(5),
                Feat_3 = paste0("rnd_", sample(seq(100), 5)))
x
```

```
  Feat_1     Feat_2 Feat_3
1     21  0.1865855 rnd_37
2     22 -0.6720045 rnd_38
3     23  1.7228147 rnd_55
4     24 -0.9739142 rnd_58
5     25  0.2165614 rnd_65
```

0.40.1 Extract column(s)

Just like in a list, using the $ operator returns an element, i.e. a **vector**:

```
x$Feat_2
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
class(x$Feat_2)
```

```
[1] "numeric"
```

Accessing a column by name with square brackets, returns a single-column **data.frame**:

```
x["Feat_2"]
```

```
      Feat_2
1  0.1865855
2 -0.6720045
3  1.7228147
4 -0.9739142
5  0.2165614
```

```
class(x["Feat_2"])
```

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

Again, similar to a list, if you double the square brackets, you access the element within the data.frame, which is a vector:

```
x[["Feat_2"]]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

Accessing a column by `[row, column]` either by position or name, return a vector by default:

```
x[, 2]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
class(x[, 2])
```

```
[1] "numeric"
```

```
x[, "Feat_2"]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
class(x[, "Feat_2"])
```

```
[1] "numeric"
```

The above happens, because by default the argument `drop` is set to `TRUE`. Set it to `FALSE` to return a `data.frame`:

```
class(x[, 2, drop = FALSE])
```

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

```
class(x[, "Feat_2", drop = FALSE])
```

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

As in lists, with the exception of the $ notation, all other indexing and slicing operations work with a variable holding either a column name of or an integer location:

```
idi <- 2
idc <- "Feat_2"
x[idi]
```

```
      Feat_2
1  0.1865855
2 -0.6720045
3  1.7228147
4 -0.9739142
5  0.2165614
```

```
x[idc]
```

```
      Feat_2
1  0.1865855
2 -0.6720045
3  1.7228147
4 -0.9739142
5  0.2165614
```

```
x[[idi]]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
x[[idc]]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
x[, idi]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
x[, idc]
```

```
[1]  0.1865855 -0.6720045  1.7228147 -0.9739142  0.2165614
```

```
x[, idi, drop = F]
```

```
      Feat_2
1  0.1865855
2 -0.6720045
3  1.7228147
4 -0.9739142
5  0.2165614
```

```
x[, idc, drop = F]
```

```
      Feat_2
1  0.1865855
2 -0.6720045
3  1.7228147
4 -0.9739142
5  0.2165614
```

Extracting multiple columns returns a data frame:

```
x[, 2:3]
```

```
      Feat_2 Feat_3
1  0.1865855 rnd_37
2 -0.6720045 rnd_38
3  1.7228147 rnd_55
4 -0.9739142 rnd_58
5  0.2165614 rnd_65
```

```
class(x[, 2:3])
```

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

0.40.2 Extract rows

A row is a small data.frame, since it contains multiple columns:

```
x[1, ]
```

```
  Feat_1    Feat_2 Feat_3
1     21 0.1865855 rnd_37
```

```
class(x[1, ])
```

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

Convert into a list using `c()`:

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

```
$Feat_1
[1] 21

$Feat_2
[1] 0.1865855

$Feat_3
[1] "rnd_37"
```

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

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

Convert into a (named) vector using `unlist()`:

```
unlist(x[1, ])
```

```
            Feat_1            Feat_2            Feat_3
             "21" "0.186585475386487"          "rnd_37"
```

```
class(unlist(x[1, ]))
```

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

0.40.3 Logical index

```
x[x$Feat_1 > 22, ]
```

```
  Feat_1     Feat_2 Feat_3
3     23  1.7228147 rnd_55
4     24 -0.9739142 rnd_58
5     25  0.2165614 rnd_65
```

0.41 Logical <-> Integer indexing

As we saw, there are two types of indexes/indices: integer and logical.

- A logical index needs to be of the same dimensions as the object it is indexing (unless you really want to recycle values - see chapter on vectorization): you are specifying whether to include or exclude each element
- An integer index will be shorter than the object it is indexing: you are specifying which subset of elements to include (or with a − in front, which elements to exclude)

It's easy to convert between the two types.

For example, start with a sequence of integers:

```
x <- 21:30
x
```

```
 [1] 21 22 23 24 25 26 27 28 29 30
```

Let's create a logical index based on two inequalities:

```
logical_index <- x > 23 & x < 28
logical_index
```

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

0.41.1 Logical to integer index with `which()`:

The common mistake is to attempt to convert a logical index to an integer index using `as.integer()`. This results in a vector of 1's and 0's, NOT an integer index.
`which()` converts a logical index to an integer index.

`which()` literally gives the position of all `TRUE` elements in a vector, thus converting a logical to an integer index:

```
integer_index <- which(logical_index)
integer_index
```

```
[1] 4 5 6 7
```

i.e. positions 4, 5, 6, 7 of the `logical_index` are TRUE

A logical and an integer index are equivalent if they select the exact same elements

Let's check than when used to index `x`, they both return the same result:

```
x[logical_index]
```

```
[1] 24 25 26 27
```

```
x[integer_index]
```

```
[1] 24 25 26 27
```

```
all(x[logical_index] == x[integer_index])
```

```
[1] TRUE
```

0.41.2 Integer to logical index

On the other hand, if we want to convert an integer index to a logical index, we can begin with a logical vector of the same length or dimension as the object we want to index with all FALSE values:

```
logical_index_too <- vector(length = length(x))
logical_index_too
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

And use the integer index to replace the corresponding elements to TRUE:

```
logical_index_too[integer_index] <- TRUE
logical_index_too
```

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

This, of course, is the same as the logical index we started with.

```
all(logical_index == logical_index_too)
```

```
[1] TRUE
```

0.42 Exclude cases using an index

Very often, we want to use an index, whether logical or integer, to exclude cases instead of to select cases.
To do that with a logical integer, we simply use an exclamation point in front of the index to negate each element (convert each TRUE to FALSE and each FALSE to TRUE):

```
logical_index
```

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

```
!logical_index
```

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

```
x[!logical_index]
```

```
[1] 21 22 23 28 29 30
```

To exclude elements using an integer index, R allows you to use negative indexing:

```
x[-integer_index]
```

```
[1] 21 22 23 28 29 30
```

To get the complement of an index, you negate a logical index (`!logical_index`) or you subtract an integer index (`-integer_index`):

0.43 subset()

`subset()` allows you to filter cases that meet certain conditions using the `subset` argument, and optionally also select columns using the `select` argument:

(`head()` returns the first few lines of a data frame. We use it to avoid printing too many lines)

```
head(iris)
```

```
  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
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
iris_sl.gt.med <- subset(iris, Sepal.Length > median(Sepal.Length))
```

Note: You can use the column name Sepal.Length directly, i.e. unquoted and you don't need to use `iris$Sepal.Length`. (This is called Non-Standard Evaluation, NSE)

```
x <- data.frame(one = 1:10,
                two = rnorm(10),
                group = c(rep("alpha", 4),  rep("beta", 6)))
subset(x, subset = two > 0, select = two)
```

```
        two
3 0.3229268
4 0.8261063
5 0.5371031
9 0.3025462
```

```
subset(x, two > 0, -one)
```

```
        two group
3 0.3229268 alpha
4 0.8261063 alpha
5 0.5371031  beta
9 0.3025462  beta
```

```
subset(x, two > 0, two:one)
```

```
        two one
3 0.3229268   3
4 0.8261063   4
5 0.5371031   5
9 0.3025462   9
```

```
subset(x, two > 0, two:group)
```

```
        two group
3 0.3229268 alpha
4 0.8261063 alpha
5 0.5371031  beta
9 0.3025462  beta
```

0.44 split()

Split a data frame into multiple data frames by groups defined by a factor:

```
x_by_group <- split(x, x$group)
```

0.45 with()

Within a `with()` expression, you can access data.frame columns without quoting or using the $ operator:

```
with(x, x[group == "alpha", ])
```

```
  one         two group
1   1 -0.02064675 alpha
2   2 -0.46339186 alpha
3   3  0.32292683 alpha
4   4  0.82610626 alpha
```

```
with(x, x[two > 0, ])
```

```
  one       two group
3   3 0.3229268 alpha
4   4 0.8261063 alpha
5   5 0.5371031  beta
9   9 0.3025462  beta
```

Vectorized Operations

Most built-in R functions are vectorized and many functions from external packages are as well.

A vectorized function operates on all elements of a vector at the same time.

Vectorization is very efficient: it can save both human (your) time and machine time. In many cases, applying a function on all elements simultaneously may seem obvious or expected behavior, but since not all functions are vectorized, make sure to check the documentation if unsure.

0.46 Operations between vectors of equal length

Such operations are applied between corresponding elements of each vector:

```
x <- 1:10
z <- 11:20
x
```

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

```
z
```

```
 [1] 11 12 13 14 15 16 17 18 19 20
```

```
x + z
```

```
 [1] 12 14 16 18 20 22 24 26 28 30
```

i.e. the above is equal to `c(x[1] + z[1], x[2] + z[2], ..., x[n] + z[n])`

0.47 Operations between a vector and a scalar

In this cases, the scalar is essentially recycled, i.e. repeated to match the length of the vector:

```
(x + 10)
```

```
 [1] 11 12 13 14 15 16 17 18 19 20
```

```
(x * 2)
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
(x / 10)
```

```
 [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
(x ^ 2)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

0.48 Operations between vectors of unequal length: value recycling

Operations between a vector and a scalar are a special case of operations between vectors of unequal length. Whenever you perform an operation between two objects of different length, ***the shorter object's elements are recycled***:

```
x + c(2:1)
```

```
 [1]  3  3  5  5  7  7  9  9 11 11
```

Operations between objects of unequal length can occur by mistake. If the shorter object's length is a multiple of the longer object's length, there will be *no error or warning*, as above. Otherwise, there is a warning (which may be confusing at first) BUT ***recycling still happens and is highly unlikely to be intentional***:

```
x + c(1, 3, 9)
```

```
Warning in x + c(1, 3, 9): longer object length is not a multiple of shorter
object length
```

```
 [1]  2  5 12  5  8 15  8 11 18 11
```

0.49 Vectorized matrix operations

Operations between matrices are similarly vectorized, i.e. performed between corresponding elements:

```
a <- matrix(1:4, 2)
b <- matrix(11:14, 2)
a
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
b
```

```
     [,1] [,2]
[1,]   11   13
[2,]   12   14
```

```
a + b
```

```
     [,1] [,2]
[1,]   12   16
[2,]   14   18
```

```
a * b
```

```
     [,1] [,2]
[1,]   11   39
[2,]   24   56
```

```
a / b
```

```
           [,1]      [,2]
[1,] 0.09090909 0.2307692
[2,] 0.16666667 0.2857143
```

0.50 Vectorized functions

Some examples of common mathematical operations that are vectorized:

```
log(x)
```

```
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
```

```
sqrt(x)
```

```
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.82842
 [9] 3.000000 3.162278
```

```
sin(x)
```

```
 [1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243 -0.2794155
 [7]  0.6569866  0.9893582  0.4121185 -0.5440211
```

```
cos(x)
```

```
 [1]  0.5403023 -0.4161468 -0.9899925 -0.6536436  0.2836622  0.9601703
 [7]  0.7539023 -0.1455000 -0.9111303 -0.8390715
```

0.51 `ifelse()`

`ifelse()` is vectorized and can be a great and compact alternative to a more complicated expression:

```
a <- 1:10
(y <- ifelse(a > 5, 11:20, 21:30))
```

```
 [1] 21 22 23 24 25 16 17 18 19 20
```

so what did this do?

It is equivalent to:

```
idl <- a > 5
yes <- 11:20
no <- 21:30
out <- vector("numeric", 10)
for (i in seq(a)) {
  if (idl[i]) {
    out[i] <- yes[i]
  } else {
    out[i] <- no[i]
  }
}
out
```

```
 [1] 21 22 23 24 25 16 17 18 19 20
```

i.e.

- Create a logical index using `test`
- for each element `i` in `test`:
  - if the element `i` is TRUE, return `yes[i]`, else `no[i]`

Data Input/Output

0.52 R datasets

0.52.1 Datasets included with R (in package 'datasets')

List builtin datasets with `data` and no arguments:

```
data()
```

These builtin datasets are normally readily available in the R console (because the `datasets` package is automatically loaded)
You can check if this is the case using `search()`

```
search()
```

```
[1] ".GlobalEnv"     "package:stats"    "package:graphics"
[4] "package:grDevices" "package:utils"    "package:datasets"
[7] "package:methods"   "Autoloads"        "package:base"
```

0.52.2 Datasets included with other packages

List a dataset included with some R package:

```
data(package = "glmnet")
data(package = "MASS")
data(package = "mlbench")
```

Load a dataset from some R package:

```
data(Sonar, package = "mlbench")
```

Note: quotes around "Sonar" in the `data()` command above are optional.

0.53 System commands

Get working directory with `getwd()`

```
getwd()
```

You can set a different working directory with `setwd()`

List files in current directory:

```
dir()
```

You can execute a command of you operating system (OS) -i.e. MacOS, Linux, Windows- from within R using the `system()` function:

```
system("uname -a")
```

Note: See issue here[9]

0.54 Data I/O

0.54.1 Read local CSV

`read.table()` is the core function that reads data from formatted text files in R, where cases correspond to lines and variables to columns. Its many arguments allow to read different formats.
`read.csv()` is an alias for `read.table()` that defaults to commas as separators and dots for decimal points. (Run `read.csv` in the console to print its source and see for yourself and read the documentation with `?read.table`).

Some important arguments for `read.table()` listed here with their default values for `read.csv()`:

[9] `https://stackoverflow.com/questions/27388964/rmarkdown-not-outputting-results-of-system-command-to-html-file`

- `sep = ","`: Character that separate entries. Default is a comma; use `""` for tab-separated files (default setting in `read.delim()`)
- `dec = "."`: Character for the decimal point. Defaultis a dot; in some cases where a comma is used as the decimal point, the entry separator `sep` may be a semicolon (default setting in `read.csv2()`)
- `na.strings = "NA"`: Character vector of strings to be coded as "NA"

```
men <-  read.csv("../Data/pone.0204161.s001.csv")
```

0.54.2 Read data from the web

`read.csv()` can directly read an online file. In the second example below, we also define that missing data is coded with `?` using the `na.strings` argument:

```
parkinsons <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-database

sleep <- read.csv("https://www.openml.org/data/get_csv/53273/sleep.arff",
                  na.strings = "?")
```

The above files are read from two very popular online data repositories[10]. Confusingly, neither file ends in `.csv`, but they both work with `read.csv()`. Always look at the plain text file first to determine if it can work with `read.table()`/`read.csv()` and what settings to use.

0.54.3 Read zipped data from the web

0.54.3.1 using base gzcon and csv.read

`read.table()`/`read.csv()` also accepts a "connection" as input.
Here we define a connection to a zipped file by nesting `gzcon()` and `url()`:

```
con <- gzcon(url("https://github.com/EpistasisLab/pmlb/raw/master/datasets/breas
             text = TRUE)
```

We read the connection and specify the file is tab-separated, or call `read.delim()`:

```
bcw <- read.csv(con, header = TRUE, sep = "\t")
```

[10] `https://rtemis.lambdamd.org/resources.html#datasets`

```
#same as
bcw <- read.delim(con, header = TRUE)
```

0.54.3.2 using data.table's `fread()`

You can also use **data.table**'s `fread()`, which will directly handle zipped files:

```
library(data.table)
bcw2 <- fread("https://github.com/EpistasisLab/penn-ml-benchmarks/raw/ma
```

If you want to stick to using data frames, set the argument `data.table` to `FALSE`:

```
bcw2 <- fread("https://github.com/EpistasisLab/penn-ml-benchmarks/raw/ma
              data.table = FALSE)
```

0.54.4 Write to CSV

Use the `write.csv()` function to write an R object (usually data frame or matrix) to a CSV file. Setting `row.names = FALSE` is usually a good idea. (Instead of storing data in rownames, it's usually best to create a new column.)

```
write.csv(iris, "../Data/iris.csv", row.names = FALSE)
```

Note that in this case we did not need to save row names (which are just integers 1 to 150 and would add a useless extra column in the output)

0.54.5 Read .xslx using `openxlsx::read.xlsx()`

As an example, we can read the csv we saved earlier into Excel and then save it as a .xlsx file.

```
iris.path <- normalizePath("../Data/iris.xlsx")
iris2 <- openxlsx::read.xlsx(iris.path)
```

Note: `openxlsx::read.xlsx()` does not work with a relative path like `"./Data/iris.xlsc"`. Therefore we used the `normalizePath()` function to give us the full path of the file without having to type it out.

Check that the data is still identical:

```
all(iris == iris2)
```

0.54.6 Write an R object to RDS

You can write any R object directly to file so that you can recover it at any time, share it, etc. Remember that since a list can contain any number of objects of any type, you can save any collection of objects as an RDS file. For multiple objects, see also the `save.image` command below.

```
saveRDS(iris, "iris.rds")
```

To load an object saved in an rds file, assign it to an object usig `readRDS`:

```
iris_fromFile <- readRDS("iris.rds")
all(iris == iris_fromFile)
```

0.54.7 Write multiple R objects to RData file using `save`

```
mat1 <- sapply(seq_len(10), function(i) rnorm(500))
mat2 <- sapply(seq_len(10), function(i) rnorm(500))
save(mat1, mat2, file = "./mat.RData")
```

Note: we will learn how to use `sapply` later under "Loop functions"

To load the variables in the `.RData` file you saved, use the `load` command:

```
load("./Rmd/mat.RData")
```

Note that `load` adds the objects to your workspace using with their original names. You do not assign them to a new object, unlike with the `readRDS` call above.

0.54.8 Write your entire workspace to a RData image using `save.image`

You can save your entire workspace to a RData file using the `save.image` function.

```
save.image("workspace_10_05_2020.RData")
```

Same as above, to re-load the workspace saved in the `.RData` file, use the `load` command:

```
load("workspace_10_05_2020.RData")
```

Control flow

Code is often not executed linearly (i.e. line-by-line). Control flow (or flow of control) operations define the order in which code segments are executed.

Execution is often conditional (`if - then - else` or `switch`).

Segments of code may be repeated multiple times (`for`) or as long as certain conditions are met (`while`).

Control flow operations form some of the fundamental building blocks of programs. Each operation is very simple - combine enough of them and you can build up to any amount of complexity.

- `if` [Condition] `then` [Expression] `else` [Alternate Expression]
- `for` [Variable in Sequence] do [Expression]
- `while` [Condition] do [Expression]
- `repeat` [Expression] until `break`
- `break`: break out of `for`, `while` or `repeat` loop
- `next`: skip current iteration and proceed to next

0.55 `if-then-else`:

```
a <- 4
if (a < 10) {
  cat("a is not that big")
} else {
  cat("a is not too small")
}
```

```
a is not that big
```

0.56 `if-then-else` `if-else`:

```
a <- sample(seq(-2, 2, .5), 1)
a
```

```
[1] 0.5
```

```
if (a > 0) {
  result <- "positive"
} else if (a == 0) {
  result <- "zero"
} else {
  result <- "negative"
}
result
```

```
[1] "positive"
```

0.57 Conditional assignment with `if-else`:

You can use an `if` statement as part of an assignment:

```
a <- 8
y <- if (a > 5) {
  10
} else {
  0
}
```

0.58 Conditional assignment with `ifelse`:

```
a <- 3
(y <- ifelse(a > 5, 10, 0))
```

```
[1] 0
```

`ifelse` is vectorized:

```
a <- 1:10
(y <- ifelse(a > 7, a^2, a))
```

```
 [1]   1   2   3   4   5   6   7  64  81 100
```

0.59 `for` loops

Use `for` loops to repeat execution of a block of code a certain number of times.

The syntax is `for (var in vector) expression`.
The `expression` is usually surrounded by curly brackets and can include any number of lines, any amount of code:

```
for (i in 1:5) {
  print("I love coffee")
}
```

```
[1] "I love coffee"
[1] "I love coffee"
[1] "I love coffee"
[1] "I love coffee"
[1] "I love coffee"
```

The loop executes for `length(vector)` times.
At iteration `i`, `var = vector[i]`.
You will often use the value of `var` inside the loop (but you don't have to):

```
for (i in seq(10)) {
  cat(i^2, "\n")
}
```

```
1
4
9
16
25
36
49
64
81
100
```

`letters` is a built-in constant that includes all 26 lowercase letters of the Roman alphabet; `LETTERS` similarly includes all 26 uppercase letters.

```
for (letter in letters[1:5]) {
  cat(letter, "is a letter!\n")
}
```

```
a is a letter!
b is a letter!
c is a letter!
d is a letter!
e is a letter!
```

0.59.1 Nested `for` loops

```
a <- matrix(1:9, 3)
for (i in seq(3)) {
  for (j in seq(3)) {
    cat("  a[", i, ",", j, "] is ", a[i, j], "\n", sep = "")
  }
}
```

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

0.60 Select one of multiple alternatives with `switch`

Instead of using multiple `if` - `else if` statements, we can build a more compact call using `switch`. (It is best suited for options that are of type character, rather than numeric)

```
y <- sample(letters[seq(8)], 1)
y
```

```
[1] "h"
```

```
output <- switch(y,                          # 1. Some expression
                 a = "Well done",            # 2. The possible values of the express
                 b = "Not bad",              #    followed by the `=` and the condit
                 c = "Nice try",
                 d = "Not a nice try",
                 e = "This is bad",
                 f = "Fail",
                 "This is not even a possible grade") # 3. An optional last argu
                                                      #    value, if there is no
output
```

```
[1] "This is not even a possible grade"
```

```
a <- rnorm(1)
a
```

```
[1] -0.5205781
```

```
out <- switch(as.integer(a > 0),
              `1` = "Input is positive",
              `0` = "Input is not positive")
out
```

```
NULL
```

```
a <- rnorm(1)
a
```

```
[1] 0.004367281
```

```
out <- switch(as.character(a > 0),
              `TRUE` = "Input is positive",
              `FALSE` = "Input is not positive")
out
```

```
[1] "Input is positive"
```

0.60.1 `switch` example: HTTP Status Codes

```
status <- sample(400:410, 1)
status
```

```
[1] 406
```

```
response <- switch(as.character(status),
                   `400` = "Bad Request",
                   `401` = "Unauthorized",
                   `402` = "Payment Required",
                   `403` = "Forbidden",
                   `404` = "Not Found",
                   `405` = "Method Not Allowed",
                   `406` = "Not Acceptable",
                   `407` = "Proxy Authentication Required",
                   `408` = "Request Timeout",
                   `409` = "Conflict",
                   `410` = "Gone")
response
```

```
[1] "Not Acceptable"
```

0.61 `while` loops

```
a <- 10
while (a > 0) {
  a <- a - 1
  cat("a is equal to", a, "\n")
}
```

```
a is equal to 9
a is equal to 8
a is equal to 7
a is equal to 6
a is equal to 5
a is equal to 4
a is equal to 3
a is equal to 2
a is equal to 1
```

```
a is equal to 0
```

```
cat("when all is said and done, a is", a)
```

```
when all is said and done, a is 0
```

0.62 `break` stops execution of a loop:

```
for (i in seq(10)) {
  if (i == 5) break()
  cat(i, "squared is", i^2, "\n")
}
```

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

0.63 `next` skips the current iteration:

```
for (i in seq(7)) {
  if (i == 5) next()
  cat(i, "squared is", i^2, "\n")
}
```

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
6 squared is 36
7 squared is 49
```

0.64 `repeat` loops

`repeat` initiates an infinite loop and you **must** use `break` to exit. Probably best to use any other type of loop instead.

```
i <- 10
repeat {
 i <- i - 1
 if (i == 0) break()
 cat("i is", i, "\n")
}
```

```
i is 9
i is 8
i is 7
i is 6
i is 5
i is 4
i is 3
i is 2
i is 1
```

Loop Functions

Loop functions are some of the most widely used R functions. They replace longer expressions created with a `for` loop, for example.
They can result in more compact and readable code and are often faster to execute than a `for` loop.

- `apply()`: Apply function over array margins (i.e. over one or more dimensions)
- `lapply()`: Return a *list* where each element is the result of applying a function to each element of the input
- `sapply()`: Same as `lapply()`, but returns the simplest possible R object (instead of always returning a list)
- `vapply()`: Same as `sapply()`, but you pre-specify the return type: this is safer and may also be faster
- `tapply()`: Apply a function to elements of groups defined by a factor
- `mapply()`: Multivariate version of `sapply()`: Apply a function using the first elements of the inputs vectors, then using the second, third, and so on

Before starting to use the above functions, we need to learn about anonymous functions, which are often used within the apply functions.

0.65 `apply()`

`apply()` applies a function over one or more dimensions of an array of 2 dimensions or more (this includes matrices) or a data frame:

```
apply(array, MARGIN, FUN)
```

`MARGIN` can be an integer vector or character indicating the dimensions over which 'FUN' will be applied.

By convention, rows come first (just like in indexing), therefore:
`MARGIN = 1`: apply function on each ***row*** `MARGIN = 2`: apply function on each ***column***

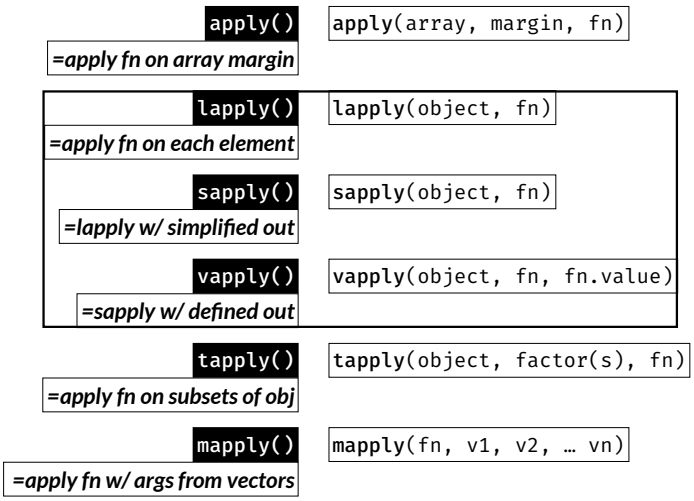


Figure 3: '*apply()' function family summary (Best to read through this chapter first and then refer back to this figure)

Let's calculate the mean value of each of the first four columns of the iris dataset:

```
x <- iris[, -5]
iris_column_mean <- apply(x, MARGIN = 2, FUN = mean)
iris_column_mean
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

Hint: It is possibly easiest to think of the "MARGIN" as the ***dimension you want to keep***.
In the above case, we want the mean for each variable, i.e. we want to keep columns and collapse rows.

The above is equivalent to:

```
iris_column_mean <- numeric(ncol(x))
names(iris_column_mean) <- names(x)

for (i in seq(x)) {
  iris_column_mean[i] <- mean(x[, i])
}
iris_column_mean
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

If you wanted to get the mean of the rows (makes little sense in this case):

```
head(apply(x, 1, mean))
```

```
[1] 2.550 2.375 2.350 2.350 2.550 2.850
```

`apply()` only works on objects with defined (i.e. non-NULL) `dim()`, i.e. arrays.

Try to think why you can't use `apply()` to apply a function `fn()` on a vector `v`.

...

...

Because that would be `fn(v)`

0.66 lapply()

`lapply()` applies a **function** on **each element of its input** and returns a **list** of the outputs.

Note: The 'elements' of a data frame are its columns (remember, a data frame is a list with equal-length elements). The 'elements' of a matrix are each cell one by one, by column. Therefore `lapply()` has a very different effect on a data frame and a matrix. `lapply()` is commonly used to iterate over the columns of a data frame.

`lapply()` is the only function of the `*apply()` family that always returns a list.

```
iris.median <- lapply(iris[, -5], median)
iris.median
```

```
$Sepal.Length
[1] 5.8

$Sepal.Width
[1] 3
```

```
$Petal.Length
[1] 4.35

$Petal.Width
[1] 1.3
```

The above is equivalent to:

```
iris.median <- vector("list", 4)
names(iris.median) <- colnames(iris[, -5])
for (i in 1:4) {
  iris.median[[i]] <- median(iris[, 1])
}
```

0.67 sapply()

`sapply()` is an alias for `lapply()`, followed by a call to `simplify2array()`. (Check the source code for `sapply()` by typing `sapply` at the console and hitting Enter).

Unlike `lapply()`, the output of `sapply()` is variable: it is the simplest R object that can hold the data type(s) resulting from the operations, i.e. a vector, matrix, data frame, or list.

```
iris.median <- sapply(iris[, -5], median)
iris.median
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
        5.80         3.00         4.35         1.30
```

```
iris.summary <- data.frame(Mean = sapply(iris[, -5], mean),
                           SD = sapply(iris[, -5], sd))
iris.summary
```

```
                Mean        SD
Sepal.Length 5.843333 0.8280661
Sepal.Width  3.057333 0.4358663
Petal.Length 3.758000 1.7652982
Petal.Width  1.199333 0.7622377
```

0.68 vapply()

Much less commonly used (possibly *underused*) than `lapply()` or `sapply()`, `vapply()` allows you to specify what the expected output looks like - for example a numeric vector of length 2, a character vector of length 1.

This can have two advantages:

- It is safer against errors
- It will sometimes be a little faster

You add the argument `FUN.VALUE` which must be of the correct **type** and **length** of the expected result ***of each iteration***.

```
vapply(iris[, -5], median, FUN.VALUE = .1)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
        5.80         3.00         4.35         1.30
```

Here, each iteration returns the median of each column, i.e. a numeric vector of length 1. Therefore `FUN.VALUE` can be any numeric scalar.

For example, if we instead returned the range of each column, `FUN.VALUE` should be a numeric vector of length 1:

```
vapply(iris[, -5], range, FUN.VALUE = rep(.1, 2))
```

```
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]          4.3         2.0          1.0         0.1
[2,]          7.9         4.4          6.9         2.5
```

If `FUN.VALUE` does not match the returned value, we get an informative error:

```
vapply(iris[, -5], range, FUN.VALUE = .1)
```

```
Error in vapply(iris[, -5], range, FUN.VALUE = 0.1): values must be length 1,
 but FUN(X[[1]]) result is length 2
```

0.69 tapply()

`tapply()` is one way (of many) to apply a function on **subgroups of data** as defined by one or more factors.
In the following example, we calculate the mean Sepal.Length by species on the iris dataset:

```
mean_Sepal.Length_by_Species <- tapply(iris$Sepal.Length, iris$Species,
mean_Sepal.Length_by_Species
```

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

The above is equivalent to:

```
species <- levels(iris$Species)
mean_Sepal.Length_by_Species <- vector("numeric", length(species))
names(mean_Sepal.Length_by_Species) <- species

for (i in seq(species)) {
  mean_Sepal.Length_by_Species[i] <-
    mean(iris$Sepal.Length[iris$Species == species[i]])
}
mean_Sepal.Length_by_Species
```

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

0.70 mapply()

The above functions all work well when you iterating over elements of a single object. `mapply()` allows you to execute a function that accepts two or more inputs, say `fn(x, z)` using the i-th element of each input, and will return: `fn(x[1], z[1])`, `fn(x[2], z[2])`, ..., `fn(x[n], z[n])`

Let's create a simple function that accepts two numeric arguments, and two vectors length 5 each:

```
raise <- function(x, power) x^power
x <- 2:6
p <- 6:2
```

Use mapply to raise each `x` to the corresponding `p`:

```
out <- mapply(raise, x, p)
out
```

```
[1]  64 243 256 125  36
```

The above is equivalent to:

```
out <- vector("numeric", 5)
for (i in seq(5)) {
  out[i] <- raise(x[i], p[i])
}
out
```

```
[1]  64 243 256 125  36
```

0.71 Iterating over a sequence instead of an object

With `lapply()`, `sapply()` and `vapply()` there is a very simple trick that may often come in handy:
Instead of iterating over elements of an object, you can iterate over an integer index of whichever elements you want to access and use it accordingly within the anonymous function.
This alternative approach is much closer to how we would use an integer sequence in a `for` loop.
It will be clearer through an example:

Get the mean of the first four columns of iris:

```
# original way: iterate through elements i.e. columns:
sapply(iris[, -5], function(i) mean(i))
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

```
# alternative way: iterate over integer index of elements:
sapply(1:4, function(i) mean(iris[, i]))
```

```
[1] 5.843333 3.057333 3.758000 1.199333
```

```
# equivalent to:
for (i in 1:4) {
  mean(iris[, i])
}
```

Notice that in the alternative approach, since you are not passing the object (iris, here) as the input to `lapply()`, therefore it needs to be specified within the anonymous function.

0.72 *apply()ing on matrices vs. data frames

To consolidate some of what was learned above, let's focus on the difference between working on a matrix vs. a data frame.
First, let's create a matrix and a data frame with the same data:

```
amat <- matrix(21:70, 10)
colnames(amat) <- paste0("Feature_", 1:ncol(amat))
amat
```

```
      Feature_1 Feature_2 Feature_3 Feature_4 Feature_5
 [1,]        21        31        41        51        61
 [2,]        22        32        42        52        62
 [3,]        23        33        43        53        63
 [4,]        24        34        44        54        64
 [5,]        25        35        45        55        65
 [6,]        26        36        46        56        66
 [7,]        27        37        47        57        67
 [8,]        28        38        48        58        68
 [9,]        29        39        49        59        69
[10,]        30        40        50        60        70
```

```
adf <- as.data.frame(amat)
adf
```

```
   Feature_1 Feature_2 Feature_3 Feature_4 Feature_5
1         21        31        41        51        61
2         22        32        42        52        62
3         23        33        43        53        63
4         24        34        44        54        64
5         25        35        45        55        65
6         26        36        46        56        66
7         27        37        47        57        67
8         28        38        48        58        68
9         29        39        49        59        69
10        30        40        50        60        70
```

We've seen that with `apply()` we specify the dimension to operate on and it works the same way on both matrices and data frames:

```
apply(amat, 2, mean)
```

```
Feature_1 Feature_2 Feature_3 Feature_4 Feature_5
     25.5      35.5      45.5      55.5      65.5
```

```
apply(adf, 2, mean)
```

```
Feature_1 Feature_2 Feature_3 Feature_4 Feature_5
     25.5      35.5      45.5      55.5      65.5
```

However, `sapply()` (and `lapply()`, `vapply()`) acts on ***each element*** of the object, therefore it is not meaningful to pass a matrix to it:

```
sapply(amat, mean)
```

```
 [1] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
[26] 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
```

The above returns the mean of each element, i.e. the element itself, which is pointless.

Since a data frame is a list, and its columns are its elements, it works great for column operations on data frames:

```
sapply(adf, mean)
```

```
Feature_1 Feature_2 Feature_3 Feature_4 Feature_5
     25.5      35.5      45.5      55.5      65.5
```

If you want to use `sapply()` on a matrix, you could iterate over an integer sequence as shown in the previous section:

```
sapply(1:ncol(amat), function(i) mean(amat[, i]))
```

```
[1] 25.5 35.5 45.5 55.5 65.5
```

This is shown to help emphasize the differences between the function and the data structures. In practice, you would use `apply()` on a matrix.

0.73 Anonymous functions

Anonymous functions are just like regular functions but they are not assigned to an object - i.e. they are not "named".
They are usually passed as arguments to other functions to be used once, hence no need to name them.
In R, anonymous functions are often used with the apply family of functions.

Example of a simple regular function:

```
squared <- function(x) {
  x^2
}
```

Because this is a short function definition, it can also be written in a single line without curly brackets:

```
squared <- function(x) x^2
```

The equivalent anonymous function is the same, but omitting the assignment:

```
function(x) x^2
```

```
function(x) x^2
```

Let's use the `squared()` function within `sapply()` to square the first four columns of the iris dataset. In these examples, we often wrap functions around `head()` which prints the first few lines of an object to avoid:

```
head(iris[, 1:4])
```

```
  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
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
```

```
iris_sq <- sapply(iris[, 1:4], squared)
head(iris_sq)
```

```
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]        26.01       12.25         1.96        0.04
[2,]        24.01        9.00         1.96        0.04
[3,]        22.09       10.24         1.69        0.04
[4,]        21.16        9.61         2.25        0.04
[5,]        25.00       12.96         1.96        0.04
[6,]        29.16       15.21         2.89        0.16
```

Let's do the same as above, but this time using an anonymous function:

```
iris_sqtoo <- sapply(iris[, 1:4], function(x) x^2)
head(iris_sqtoo)
```

```
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]        26.01       12.25         1.96        0.04
[2,]        24.01        9.00         1.96        0.04
[3,]        22.09       10.24         1.69        0.04
[4,]        21.16        9.61         2.25        0.04
[5,]        25.00       12.96         1.96        0.04
[6,]        29.16       15.21         2.89        0.16
```

The entire anonymous function definition is passed in the function argument (FUN in the R documentation).

Summarizing Data

Let's read in a dataset from OpenML:

```
heart <- read.csv("https://www.openml.org/data/get_csv/51/dataset_51_heart-h.arf
                  na.strings = "?")
```

0.74 Get summary of an R object with `summary()`

R includes `summary()` methods for a number of different objects.

```
summary(heart)
```

```
    age            sex            chest_pain          trestbps
 Min.   :28.00  Length:294         Length:294         Min.   : 92.0
 1st Qu.:42.00  Class :character   Class :character   1st Qu.:120.0
 Median :49.00  Mode  :character   Mode  :character   Median :130.0
 Mean   :47.83                                       Mean   :132.6
 3rd Qu.:54.00                                       3rd Qu.:140.0
 Max.   :66.00                                       Max.   :200.0
                                                     NA's   :1
     chol           fbs             restecg             thalach
 Min.   : 85.0  Length:294         Length:294         Min.   : 82.0
 1st Qu.:209.0  Class :character   Class :character   1st Qu.:122.0
 Median :243.0  Mode  :character   Mode  :character   Median :140.0
 Mean   :250.8                                       Mean   :139.1
 3rd Qu.:282.5                                       3rd Qu.:155.0
 Max.   :603.0                                       Max.   :190.0
 NA's   :23                                          NA's   :1
   exang              oldpeak          slope                ca
 Length:294         Min.   :0.0000   Length:294         Min.   :0
 Class :character   1st Qu.:0.0000   Class :character   1st Qu.:0
```

```
 Mode  :character  Median :0.0000  Mode  :character  Median :0
               Mean   :0.5861                    Mean   :0
               3rd Qu.:1.0000                    3rd Qu.:0
               Max.   :5.0000                    Max.   :0
                                                 NA's   :291
     thal                num
 Length:294         Length:294
 Class :character   Class :character
 Mode  :character   Mode  :character
```

0.75 Fast builtin column and row operations

We saw in Loop Functions how we can apply functions on rows, columns, or other subsets of our data. R has optimized builtin functions for some very common operations, with self-explanatory names:

- `colSums()`: column sums
- `rowSums()`: row sums
- `colMeans()`: column means
- `rowMeans()`: row means

```
a <- matrix(1:20, 5)
a
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

```
colSums(a)
```

```
[1] 15 40 65 90
```

```
# same as
apply(a, 2, sum)
```

```
[1] 15 40 65 90
```

```
rowSums(a)
```

```
[1] 34 38 42 46 50
```

```
# same as
apply(a, 1, sum)
```

```
[1] 34 38 42 46 50
```

```
colMeans(a)
```

```
[1]  3  8 13 18
```

```
# same as
apply(a, 2, mean)
```

```
[1]  3  8 13 18
```

```
rowMeans(a)
```

```
[1]  8.5  9.5 10.5 11.5 12.5
```

```
# same as
apply(a, 1, mean)
```

```
[1]  8.5  9.5 10.5 11.5 12.5
```

0.76 Optimized matrix operations with matrixStats

While the builtin operations above are already optimized and faster than the equivalent calls, the **matrixStats** package (Bengtsson, 2019) offers a number of futher optimized matrix operations, including drop-in replacements of the above. These should be prefered when dealing with bigger data:

```
library(matrixStats)
colSums2(a)
```

```
[1] 15 40 65 90
```

```
rowSums2(a)
```

```
[1] 34 38 42 46 50
```

```
colMeans2(a)
```

```
[1]  3  8 13 18
```

```
rowMeans2(a)
```

```
[1]  8.5  9.5 10.5 11.5 12.5
```

Note: **matrixStats** provides replacement functions named almost identically to their base counterpart - so they are easy to find - but are different - so they don't mask the base functions (this is important and good software design).

0.77 Grouped summary statistics with `aggregate()`

`aggregate()` is a powerful way to apply functions on splits of your data. It can replicate functionality of the `*apply()` family, but can be more flexible/powerful and supports a formula input.

```
aggregate(iris[, -5], by = list(iris$Species), mean)
```

```
   Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
1    setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

Alternatively, the more compact **formula notation** can be used here to get the same result.
The `.` on the left hand side represents all features, excluding those on the right hand side:

```
aggregate(. ~ Species, iris, mean)
```

```
   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1    setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

To define multiple specific variables on the left hand side of the formula within `aggregate()`, use `cbind()`:

```
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species, iris, mean)
```

```
     Species Sepal.Length Sepal.Width
1     setosa        5.006       3.428
2 versicolor        5.936       2.770
3  virginica        6.588       2.974
```

Let's make up a second grouping:

```
irisd <- iris
irisd$Group2 <- rep(1:2, 3, each = 25)
irisd$Group2
```

```
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 [38] 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [75] 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
[112] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[149] 2 2
```

This "Group2" divides each iris Species in the first and last 25 cases.
Let's aggregate by both Species and Group2:

```
aggregate(irisd[, -c(5, 6)],
          by = list(Species = irisd$Species, Group2 = irisd$Group2),
          mean)
```

```
    Species Group2 Sepal.Length Sepal.Width Petal.Length Petal.Width
1    setosa      1        5.028       3.480        1.460       0.248
2 versicolor      1        6.012       2.776        4.312       1.344
3  virginica      1        6.576       2.928        5.640       2.044
4    setosa      2        4.984       3.376        1.464       0.244
5 versicolor      2        5.860       2.764        4.208       1.308
6  virginica      2        6.600       3.020        5.464       2.008
```

The compact formula notation can be very convenient here:

```
aggregate(. ~ Species + Group2, irisd, mean)
```

```
    Species Group2 Sepal.Length Sepal.Width Petal.Length Petal.Width
1    setosa      1        5.028       3.480        1.460       0.248
2 versicolor      1        6.012       2.776        4.312       1.344
```

```
3  virginica    1     6.576     2.928     5.640     2.044
4     setosa    2     4.984     3.376     1.464     0.244
5 versicolor    2     5.860     2.764     4.208     1.308
6  virginica    2     6.600     3.020     5.464     2.008
```

Note: Using aggregate with the `by = list()` argument is easier to code with. The formula notation might be easier to work with in real time on the console. You *can* code with the formula notation, but if there is an alternative it's unlikely to be worth the extra steps.

Functions

Writing functions is a core part of programming. When should you write a function?
Whenever you find yourself repeating pieces of code.
Why is it important?
Writing functions helps reduce the total amount of code, which reduces the chances of error, and makes your code more readable.

Reminder: Functions in R are "first class objects".
This means you can pass them in and out of other functions or objects like any other R structure.
For example, we have seen that you can use a command like `apply(mat, 2, mean)`

Functions in R are *for the most part* like mathematical functions: they have one or more inputs and one output. The inputs are known as the function arguments. If you want to return multiple outputs, you can return a list containing any number of R objects.

0.78 Simple functions

Let's start with a very simple function: single argument with no default value:

```
square <- function(x) {
  x^2
}

square(3)
```

```
[1] 9
```

Notice above that `x^2` is automatically returned by the function. It is the same as explicitly returning it with `return()`:

```
square <- function(x) {
  out <- x^2
  return(out)
}

square(4)
```

```
[1] 16
```

which is the same as:

```
square <- function(x) {
  out <- x^2
  out
}

square(5)
```

```
[1] 25
```

A function returns either:

- an object passed to `return()`
- the value of the last expression within the function definition such as `out` or `x^2` above.

Note that the following function definition does not return anything:

```
sq <- function(x) {
  out <- x^2
}

sq(5)
```

`return()` is a way to end evaluation early:

```
square.pos <- function(x) {
  if (x > 0) {
    return(x^2)
  } else {
    x
  }
  cat("The input was left unchanged\n")
}
```

```
x <- sample(-10:10, 1)
x
```

```
[1] -5
```

```
square.pos(x)
```

```
The input was left unchanged
```

Multiple arguments, with and without defaults:

```
raise <- function(x, power = 2) {
  x^power
}

x <- sample(10, 1)
x
```

```
[1] 1
```

```
raise(x)
```

```
[1] 1
```

```
raise(x, power = 3)
```

```
[1] 1
```

```
raise(x, 3)
```

```
[1] 1
```

0.79 Arguments with prescribed list of allowed values

You can match specific values for an argument using `match.arg()`:

```
myfn <- function(type = c("alpha", "beta", "gamma")) {
  type <- match.arg(type)
  cat("You have selected type '", type, "'\n", sep = "")
```

```
}

myfn("a")
```

```
You have selected type 'alpha'
```

```
myfn("b")
```

```
You have selected type 'beta'
```

```
myfn("g")
```

```
You have selected type 'gamma'
```

```
myfn("d")
```

```
Error in match.arg(type): 'arg' should be one of "alpha", "beta", "gamma"
```

Above you see that partial matching using `match.arg()` was able to identify a valid option, and when there was no match, an informative error was printed.

Partial matching is also automatically done on the argument names themselves, but it's important to avoid depending on that.

```
adsr <- function(attack = 100,
                 decay = 250,
                 sustain = 40,
                 release = 1000) {
  cat("Attack time:", attack, "ms\n",
      "Decay time:", decay, "ms\n",
      "Sustain level:", sustain, "\n",
      "Release time:", release, "ms\n")
}

adsr(50, s = 100, r = 500)
```

```
Attack time: 50 ms
 Decay time: 250 ms
 Sustain level: 100
 Release time: 500 ms
```

0.80 Passing extra arguments to another function with the ... argument

Many functions include a ... argument at the end. Any arguments not otherwise matched are collected there. A common use for this is to pass them to another function:

```
cplot <- function(x, y,
                  cex = 1.5,
                  pch = 16,
                  col = "#18A3AC",
                  bty = "n", ... ) {
  plot(x, y, cex = cex, pch = pch, col = col, bty = bty, ... )
                  }
```

... is also used for variable number of iputs, often as the first argument of a function. For example, look at the documentation of `c`, `cat`, `cbind`, `rbind`, `paste`

Note: Any arguments after the ... , **must** be named fully, i.e. will not be partially matched.

0.81 Return multiple objects

R function can only return a single object. This is not much of a problem because you can simply put any collection of objects into a list and return it:

```
lfn <- function(x, fn = square) {
  xfn <- fn(x)

  list(x = x,
       xfn = xfn,
       fn = fn)
}

lfn(3)
```

```
$x
[1] 3

$xfn
[1] 9

$fn
```

```
function(x) {
  out <- x^2
  out
}
<bytecode: 0×7fb44a983720>
```

0.82 Warnings and errors

You can produce a warning at any point during a function evaluation using `warning("warning message here")`. This cause `warning message here` to be printed to the console as a warning, but does not stop function evaluation.

To stop function execution, e.g. if an error is encountered, use `stop()`. The following function calculates

$$e^{log_{10}(x)}$$

which is not defined for negative `x`. In this case we could let R give an error when it tries to compute `log10(x)`, or check x ourselves and write a custom error:

```
el10 <- function(x) {
  if (x < 0) stop("x must be positive")
  exp(log10(x))
}

el10(-3)
```

```
Error in el10(-3): x must be positive
```

```
el10(3)
```

```
[1] 1.611429
```

0.83 Scoping

Functions exist in their own environment, i.e. contain their own variable definitions.

```
x <- 3
y <- 4
fn <- function(x, y) {
  x <- 10*x
  y <- 20*y
```

```
  cat("Inside the function, x = ", x, " and y = ", y, "\n")
}

fn(x, y)
```

```
Inside the function, x =  30  and y =  80
```

```
cat("Outside the function, x = ", x, " and y = ", y, "\n")
```

```
Outside the function, x =  3  and y =  4
```

However, if a variable is referenced within a function but no local definition exists, the interpreter will look for the variable at the parent directory. It is best to not rely on this and instead make sure all variables are passed to the functions that need them.

In the following example, `x` is only defined outside the function definition, but referenced within it.

```
x <- 21

itfn <- function(y, lr = 1) {
  x + lr * y
}

itfn(3)
```

```
[1] 24
```

0.84 The pipe operator %>%

A pipe allows writing `f(x)` as `x %>% f`. It is often used to replace multiple temporary assignments in a multistep procedure, or as an alternative to nesting functions. Some packages and developers promote its use, other discourage it. As always, there is a big subjective component here and you should try and see if and when it suits you.

The following:

```
x <- f1(x)
x <- f2(x)
x <- f3(x)
```

is equivalent to:

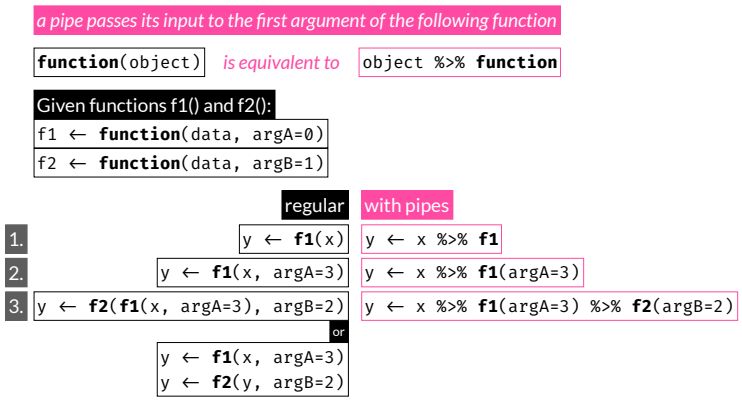


Figure 4: Illustration of pipes in R

```
x <- f3(f2(f1(x)))
```

is equivalent to:

```
x <- x %>% f1 %>% f2 %>% f3
```

The pipe operator was originally introduced in the **magrittr** package. Note that a number of other packages that allow or endorse the use of pipes export the pipe operator as well.

```
library(magrittr)
(iris[, -5] %>%
  split(iris$Species) %>%
  lapply(function(i) sapply(i, mean)) -> iris_mean_bySpecies)
```

```
$setosa
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.006        3.428        1.462        0.246

$versicolor
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.936        2.770        4.260        1.326

$virginica
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       6.588        2.974        5.552        2.026
```

Pipes are used extensively in the tidyverse[11] packages.
You can learn more about the pipe operator in the magrittr vignette[12]

[11] https://www.tidyverse.org
[12] https://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html

Working with data frames

0.85 Table Joins (i.e. Merging data.frames)

Scenario: You have received two or more tables with data. Each table consists of a unique identifier (ID), which is shared among the tables, plus a number of variables in columns, which may be unique to each table. You want to merge them into one big table so that for each ID you have all available information.

Let's make up some data:

```
a <- data.frame(PID = c(1:9),
                Hospital = c("UCSF", "HUP", "Stanford",
                             "Stanford", "UCSF", "HUP",
                             "HUP", "Stanford", "UCSF"),
                Age = c(22, 34, 41, 19, 53, 21, 63, 22, 19),
                Sex = c(1, 1, 0, 1, 0, 0, 1, 0, 0))

b  <- data.frame(PID = c(6:12),
                 V1 = c(153, 89, 112, 228,  91, 190, 101),
                 Department = c("Neurology", "Radiology",
                                "Emergency", "Cardiology",
                                "Surgery", "Neurology", "Psychiatry"))
a
```

```
  PID Hospital Age Sex
1   1     UCSF  22   1
2   2      HUP  34   1
3   3 Stanford  41   0
4   4 Stanford  19   1
5   5     UCSF  53   0
6   6      HUP  21   0
7   7      HUP  63   1
8   8 Stanford  22   0
9   9     UCSF  19   0
```

```
b
```

```
  PID  V1 Department
1   6 153  Neurology
2   7  89  Radiology
3   8 112  Emergency
4   9 228 Cardiology
5  10  91    Surgery
6  11 190  Neurology
7  12 101 Psychiatry
```

```
dim(a)
```

```
[1] 9 4
```

```
dim(b)
```

```
[1] 7 3
```

There are four main types of join operations:

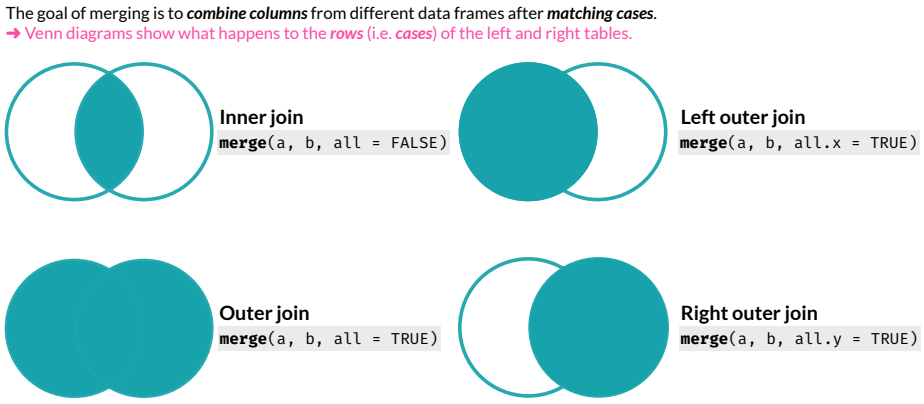


Figure 5: Common Join Operations

0.85.1 Inner join

The default arguments of `merge()` perform an **inner join**:

```
(ab.inner <- merge(a, b))
```

```
  PID Hospital Age Sex  V1 Department
1   6      HUP  21   0 153  Neurology
2   7      HUP  63   1  89  Radiology
3   8 Stanford  22   0 112  Emergency
4   9     UCSF  19   0 228 Cardiology
```

```
# same as
(ab.inner <- merge(a, b, by = "PID"))
```

```
  PID Hospital Age Sex  V1 Department
1   6      HUP  21   0 153  Neurology
2   7      HUP  63   1  89  Radiology
3   8 Stanford  22   0 112  Emergency
4   9     UCSF  19   0 228 Cardiology
```

```
# same as
(ab.inner <- merge(a, b, all = FALSE))
```

```
  PID Hospital Age Sex  V1 Department
1   6      HUP  21   0 153  Neurology
2   7      HUP  63   1  89  Radiology
3   8 Stanford  22   0 112  Emergency
4   9     UCSF  19   0 228 Cardiology
```

Note that the resulting table only contains cases found in both data frames (i.e. IDs 6 through 9)

0.85.2 Outer join

You can perform an **outer join** by specifying `all = TRUE`:

```
(ab.outer <- merge(a, b, all = TRUE))
```

```
   PID Hospital Age Sex  V1 Department
1    1     UCSF  22   1  NA       <NA>
2    2      HUP  34   1  NA       <NA>
3    3 Stanford  41   0  NA       <NA>
4    4 Stanford  19   1  NA       <NA>
5    5     UCSF  53   0  NA       <NA>
6    6      HUP  21   0 153  Neurology
7    7      HUP  63   1  89  Radiology
8    8 Stanford  22   0 112  Emergency
9    9     UCSF  19   0 228 Cardiology
```

```
10  10     <NA>  NA  NA  91    Surgery
11  11     <NA>  NA  NA 190  Neurology
12  12     <NA>  NA  NA 101 Psychiatry
```

```
(ab.outer <- merge(a, b, by = "PID", all = TRUE))
```

```
   PID Hospital Age Sex  V1 Department
1    1     UCSF  22   1  NA       <NA>
2    2      HUP  34   1  NA       <NA>
3    3 Stanford  41   0  NA       <NA>
4    4 Stanford  19   1  NA       <NA>
5    5     UCSF  53   0  NA       <NA>
6    6      HUP  21   0 153  Neurology
7    7      HUP  63   1  89  Radiology
8    8 Stanford  22   0 112  Emergency
9    9     UCSF  19   0 228 Cardiology
10  10     <NA>  NA  NA  91    Surgery
11  11     <NA>  NA  NA 190  Neurology
12  12     <NA>  NA  NA 101 Psychiatry
```

Note that the resulting data frame contains all IDs found in either input data frame and missing values are represented with NA

0.85.3 Left outer join

You can perform a **left outer join** by specifying `all.x = TRUE`:

```
(ab.leftOuter <- merge(a, b, all.x = TRUE))
```

```
  PID Hospital Age Sex  V1 Department
1   1     UCSF  22   1  NA       <NA>
2   2      HUP  34   1  NA       <NA>
3   3 Stanford  41   0  NA       <NA>
4   4 Stanford  19   1  NA       <NA>
5   5     UCSF  53   0  NA       <NA>
6   6      HUP  21   0 153  Neurology
7   7      HUP  63   1  89  Radiology
8   8 Stanford  22   0 112  Emergency
9   9     UCSF  19   0 228 Cardiology
```

Note how the resulting data frame contains all IDs present in the left input data frame only.

0.85.4 Right outer join

You can perform a **right outer join** by specifying `all.y = TRUE`:

```
(ab.rightOuter <- merge(a, b, all.y = TRUE))
```

```
  PID Hospital Age Sex  V1 Department
1   6      HUP  21   0 153  Neurology
2   7      HUP  63   1  89  Radiology
3   8 Stanford  22   0 112  Emergency
4   9     UCSF  19   0 228 Cardiology
5  10     <NA>  NA  NA  91    Surgery
6  11     <NA>  NA  NA 190  Neurology
7  12     <NA>  NA  NA 101 Psychiatry
```

Note how the resulting data frame contains all IDs present in the right input data frame only.

0.86 Wide to Long

Wide

ID	Variable A	Variable B	Variable C
1	1.1	1.2	1.3
2	2.1	2.2	2.3
3	3.1	3.2	3.3

Long

ID	Variable	Value
1	A	1.1
1	B	1.2
1	C	1.3
2	A	2.1
2	B	2.2
2	C	2.3
3	A	3.1
3	B	3.2
3	C	3.3

Figure 6: Wide and Long data format example. Take a moment to notice how the wide table on the left with 3 cases (3 IDs) and 3 variables gets converted from a 3 x 4 table to a 9 x 3 long table on the right. The values (outlined in magenta) are present once in each table: on the wide table they form an **ID x Variable** matrix, while on the long they are stacked on a **single column**. The IDs have to be repeated on the long table, once for each variable and there is a new 'Variable' column to provide the information present in the wide table's column names.

```
library(tidyr)
library(data.table)
```

Let's create an example data frame:

```
(dat_wide <- data.frame(ID = c(1, 2, 3),
                        mango = c(1.1, 2.1, 3.1),
                        banana = c(1.2, 2.2, 3.2),
                        tangerine = c(1.3, 2.3, 3.3)))
```

```
  ID mango banana tangerine
1  1   1.1    1.2       1.3
2  2   2.1    2.2       2.3
3  3   3.1    3.2       3.3
```

0.86.1 base

The `reshape()` function is probably one of the most complicated because the documentation is not clear, specifically with regards to which arguments refer to the input vs. output data frame. Use the following figure as a guide to understand `reshape()`'s syntax. You can use it as a reference when building your own `reshape()` command by following steps 1 through 5:

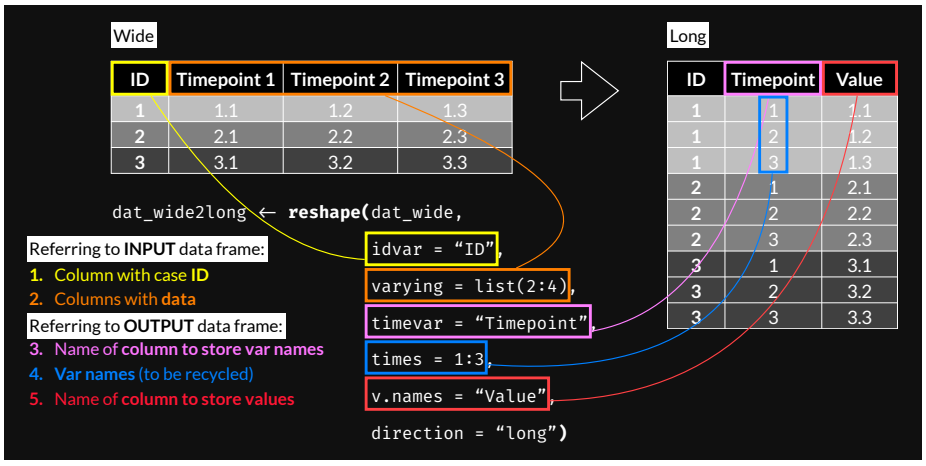


Figure 7: 'reshape()' syntax for Wide to Long transformation.

```
dat_wide2long <- reshape(# Data in wide format
                         data = dat_wide,
                         # The column name that defines case ID
                         idvar = "ID",
                         # The columns whose values we want to keep
                         varying = list(2:4),
                         # The name of the new column which will contain all
                         # the values from the columns above
                         v.names = "Score",
                         # The values/names, of length = (N columns in "varying"
                         #that will be recycled to indicate which column from th
                         #wide dataset each row corresponds to
                         times = c(colnames(dat_wide)[2:4]),
                         # The name of the new column created to hold the values
                         # defined by "times"
                         timevar = "Fruit",
                         direction = "long")
```

You can also define 'varying' with a character vector:
`varying = list(c("mango", "banana","tangerine")`

Explore the resulting data frame's attributes:

```
attributes(dat_wide2long)
```

```
$row.names
[1] "1.mango"    "2.mango"    "3.mango"    "1.banana"   "2.banana"
[6] "3.banana"   "1.tangerine" "2.tangerine" "3.tangerine"

$names
[1] "ID"    "Fruit" "Score"

$class
[1] "data.frame"

$reshapeLong
$reshapeLong$varying
$reshapeLong$varying[[1]]
[1] "mango"     "banana"    "tangerine"


$reshapeLong$v.names
[1] "Score"
```

```
$reshapeLong$idvar
[1] "ID"

$reshapeLong$timevar
[1] "Fruit"
```

These attributes and present if and only if a long data set was created from a wide as above. In that case, reshaping back to a wide data frame is as easy as:

```
reshape(dat_wide2long)
```

```
        ID mango banana tangerine
1.mango  1   1.1    1.2       1.3
2.mango  2   2.1    2.2       2.3
3.mango  3   3.1    3.2       3.3
```

0.86.2 tidyr

```
dat_wide2long_tv <- pivot_longer(dat_wide,
                                 cols = 2:4,
                                 names_to = "Fruit",
                                 values_to = "Score")
dat_wide2long_tv
```

```
# A tibble: 9 x 3
     ID Fruit     Score
  <dbl> <chr>     <dbl>
1     1 mango       1.1
2     1 banana      1.2
3     1 tangerine   1.3
4     2 mango       2.1
5     2 banana      2.2
6     2 tangerine   2.3
7     3 mango       3.1
8     3 banana      3.2
9     3 tangerine   3.3
```

0.86.3 data.table

```
dat_wide_dt <- as.data.table(dat_wide)
dat_wide2long_dt <- melt(dat_wide_dt,
                         id.vars = 1,
                         measure.vars = 2:4,
                         variable.name = "Fruit",
                         value.name = "Score")
setorder(dat_wide2long_dt, "ID")
dat_wide2long_dt
```

```
   ID     Fruit Score
1:  1     mango   1.1
2:  1    banana   1.2
3:  1 tangerine   1.3
4:  2     mango   2.1
5:  2    banana   2.2
6:  2 tangerine   2.3
7:  3     mango   3.1
8:  3    banana   3.2
9:  3 tangerine   3.3
```

0.87 Long to Wide

Let's create a long dataset:

```
(dat_long <- data.frame(ID = c(1, 2, 3, 1, 2, 3, 1, 2, 3),
                        Fruit = c("mango", "mango", "mango",
                                  "banana", "banana", "banana",
                                  "tangerine", "tangerine", "tangerine"),
                        Score = c(1.1, 2.1, 3.1, 1.2, 2.2, 3.2, 1.3, 2.3, 3.3)))
```

```
  ID     Fruit Score
1  1     mango   1.1
2  2     mango   2.1
3  3     mango   3.1
4  1    banana   1.2
5  2    banana   2.2
6  3    banana   3.2
7  1 tangerine   1.3
8  2 tangerine   2.3
9  3 tangerine   3.3
```

0.87.1 base

Using base `reshape()` for long-to-wide transformation is simpler than wide-to-long:

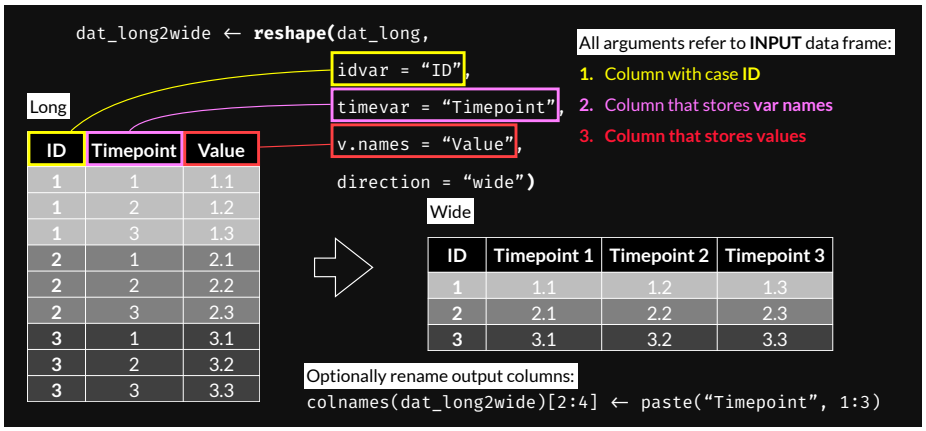


Figure 8: 'reshape()' syntax for Long to Wide transformation.

```
dat_long2wide <- reshape(dat_long,
                         idvar = "ID",
                         timevar = "Fruit",
                         v.names = "Score",
                         direction = "wide")
# Optionally rename columns
colnames(dat_long2wide) <- gsub("Score.", "", colnames(dat_long2wide))
dat_long2wide
```

```
  ID mango banana tangerine
1  1   1.1    1.2       1.3
2  2   2.1    2.2       2.3
3  3   3.1    3.2       3.3
```

0.87.2 tidyr

```
dat_long2wide_tv <- pivot_wider(dat_long,
                                id_cols = "ID",
                                names_from = "Fruit",
                                values_from = "Score")
dat_long2wide_tv
```

```
# A tibble: 3 x 4
     ID mango banana tangerine
  <dbl> <dbl>  <dbl>     <dbl>
1     1   1.1    1.2       1.3
2     2   2.1    2.2       2.3
3     3   3.1    3.2       3.3
```

0.87.3 data.table

`data.table`'s long to wide procedure is defined with a convenient formula notation:

```
dat_long_dt <- as.data.table(dat_long)
dat_long2wide_dt <- dcast(dat_long_dt,
                          ID ~ Fruit,
                          value.var = "Score")
dat_long2wide_dt
```

```
   ID banana mango tangerine
1:  1    1.2   1.1       1.3
2:  2    2.2   2.1       2.3
3:  3    3.2   3.1       3.3
```

0.88 Feature transformation with `transform()`

Make up some data:

```
dat <- data.frame(Sex = c(0, 0, 1, 1, 0),
                  Height = c(1.5, 1.6, 1.55, 1.73, 1.8),
                  Weight = c(55, 70, 69, 76, 91))
```

```
dat <- transform(dat, BMI = Weight/Height^2)
dat
```

```
  Sex Height Weight      BMI
1   0   1.50     55 24.44444
2   0   1.60     70 27.34375
3   1   1.55     69 28.72008
4   1   1.73     76 25.39343
5   0   1.80     91 28.08642
```

`transform()` is probably not used too often, because it is trivial to do the same with direct assignment:

```
dat$BMI <- dat$Weight/dat$Height^2
```

but can be useful when adding multiple variables and/or used in a pipe:

```
library(magrittr)
```

```
Attaching package: 'magrittr'
```

```
The following object is masked from 'package:tidyr':

    extract
```

```
dat %>%
  subset(Sex == 0) %>%
  transform(DeltaWeightFromMean = Weight - mean(Weight),
            BMI = Weight/Height^2,
            CI = Weight/Height^3)
```

```
  Sex Height Weight      BMI DeltaWeightFromMean       CI
1   0    1.5     55 24.44444                -17 16.29630
2   0    1.6     70 27.34375                 -2 17.08984
5   0    1.8     91 28.08642                 19 15.60357
```

Data Transformations

```
library(rtemis)
```

```
  .:rtemis 0.8.0: Welcome, egenn
 [x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
 Documentation & vignettes: https://rtemis.lambdamd.org
```

0.89 Continuous variables

0.89.1 Standardization / Scaling & Centering with `scale()`

Depending on your modeling needs / algorithms you plan to use, it is often important to scale and/or center your data. Note that many functions, but not all, will automatically scale and center data internally if it is required by the algorithm. Check the function documentation.
If you manually scale and/or center your data, you must:

- Perform scaling and centering on your training data
- Save the centering and scaling parameters for each feature
- Apply the training set-dervied centering and scaling parameters to the test set prior to prediction/inference

A common mistake is to either scale training and testing data together in the beginning, or scale them separately.

Standardizing, i.e. converting to Z-scores, involving subtracting the mean and dividing by the standard deviation. Scaling and centering in R is performed with the `scale` function. By default, both arguments `scale` and `center` are TRUE:

```
iris.scaled <- scale(iris[, -5])
```

First, let's check that it did what we were hoping:

```
colMeans(iris.scaled)
```

```
 Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
-4.480675e-16  2.035409e-16 -2.844947e-17 -3.714621e-17
```

```
apply(iris.scaled, 2, sd)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           1            1            1            1
```

Good - We got mean of 0 (effectively) and standard deviation of 1 for each column.

Now, let's get the scale and center attributes:

```
attributes(iris.scaled)
```

```
$dim
[1] 150   4

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"


$`scaled:center`
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333

$`scaled:scale`
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.8280661    0.4358663    1.7652982    0.7622377
```

Let's save the scale and center attributes and then check some values so that we are clear what is happening:

```
(.center <- attr(iris.scaled, "scaled:center"))
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

```
(.scale <- attr(iris.scaled, "scaled:scale"))
```

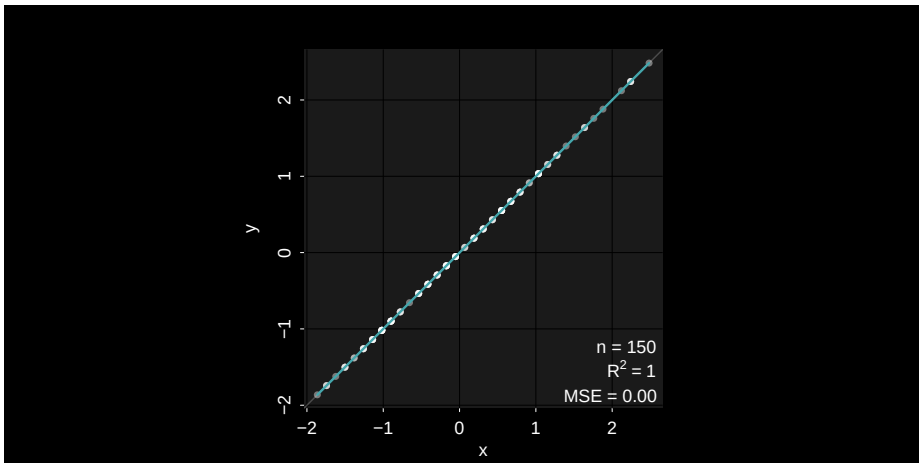
```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.8280661    0.4358663    1.7652982    0.7622377
```

```
Sepal.Length_scaled <- (iris$Sepal.Length - .center[1]) / .scale[1]
all(Sepal.Length_scaled == iris.scaled[, "Sepal.Length"])
```

```
[1] TRUE
```

Note: Due to limitation in numerical precision, checking sets of floats for equality after multiple operations is not recommended. Always a good idea to plot:

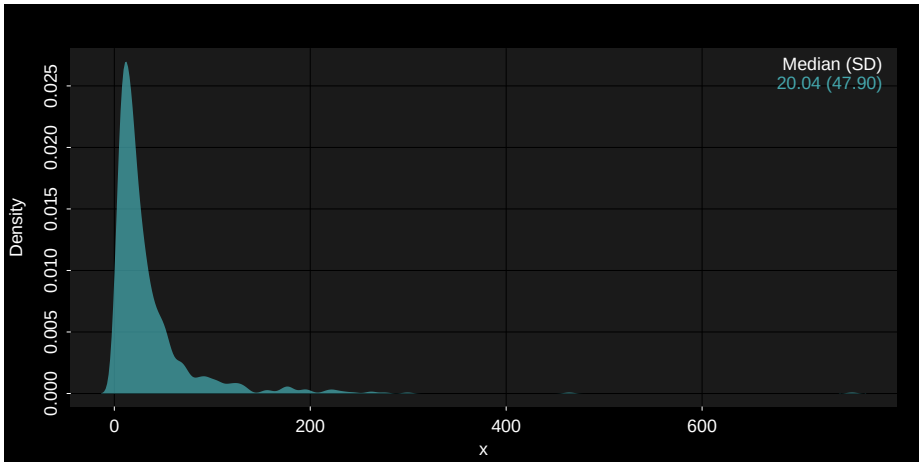
```
mplot3.fit(Sepal.Length_scaled, iris.scaled[, "Sepal.Length"])
```



0.89.2 Log-transform with `log()`

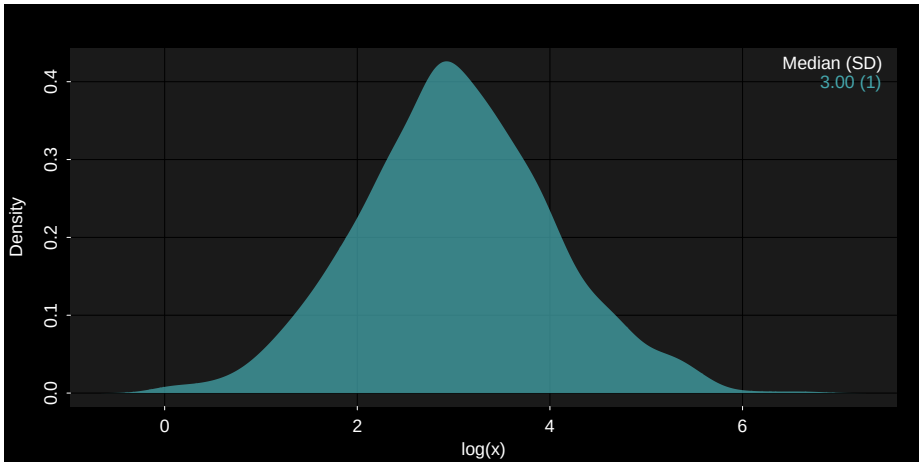
For the following example, `x` is an unknown feature in a new dataset we were just given. We start by plotting its distribution:

```
mplot3.x(x)
```



We can see it is highly skewed. A log transform may help here.
Let's check:

```
mplot3.x(log(x))
```



Looks like a good deal.

0.89.3 Data binning with `cut()`

Another approach for the above variable might be to bin it.
Let's look at a few different ways to bin continuous data.

0.89.3.1 Equal-interval cuts

By passing an integer to `cut()`'s `breaks` argument, we get that many equally-spaced intervals:

```
x_cut4 <- cut(x, breaks = 4)
table(x_cut4)
```

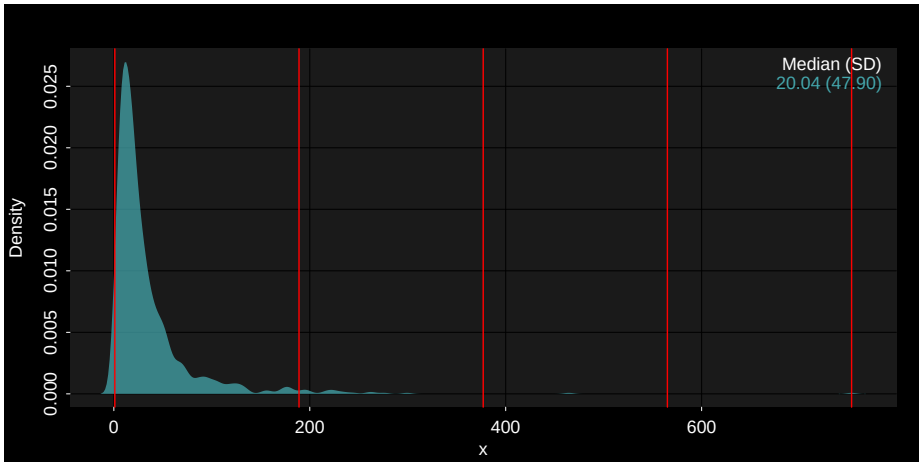
```
x_cut4
(0.248,189]   (189,377]   (377,565]   (565,754]
        981          17           1           1
```

Because the data is so skewed, equal intervals are not helpful in this case. The majority of the data gets grouped into a single bin.
Let's visualize the cuts.

```
(xcuts5 <- seq(min(x), max(x), length.out = 5))
```

```
[1]   1.0000 189.0175 377.0350 565.0525 753.0700
```

```
mplot3.x(x, par.reset = FALSE)
abline(v = xcuts5, col = "red", lwd = 1.5)
```



Note: We used `par.reset = FALSE` to stop `mplot3.x` from resetting its custom `par()` settings so that we can continue adding elements to the same plot, in this case with the `abline` command.

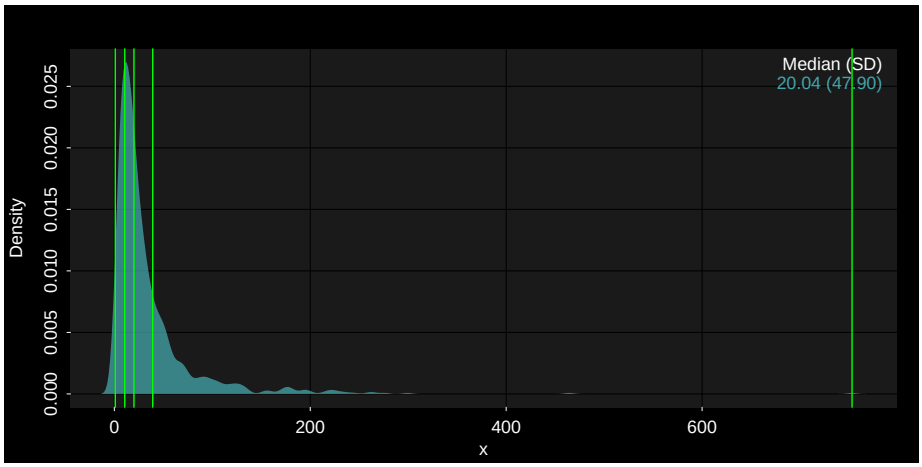
0.89.3.2 Quantile cuts

Instead, we can get quantiles. We ask for 5 quantiles which corresponds to 4 intervals:

```
(xquants5 <- quantile(x, seq(0, 1, length.out = 5)))
```

```
      0%       25%       50%       75%      100% 
 1.00000  10.65099  20.04102  39.21473 753.06995 
```

```
mplot3.x(x, par.reset = F)
abline(v = xquants5, col = "green", lwd = 1.5)
```



We can use the quantiles as breaks in `cut()`:

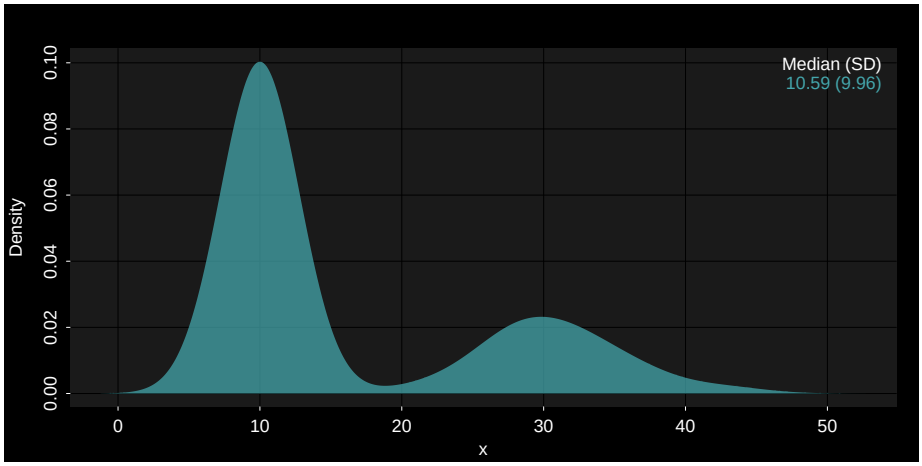
```
x_cutq4 <- cut(x, breaks = xquants5)
table(x_cutq4)
```

```
x_cutq4
  (1,10.7]  (10.7,20]  (20,39.2] (39.2,753] 
       249        250        250        250 
```

With quantile cuts, each bin contains the same number of observations (+/- 1).

We just got a new mystery `x`! Let's plot it:

```
mplot3.x(x)
```



It may be worth binning into 2. Let's look at equal-interval and quantile cuts:

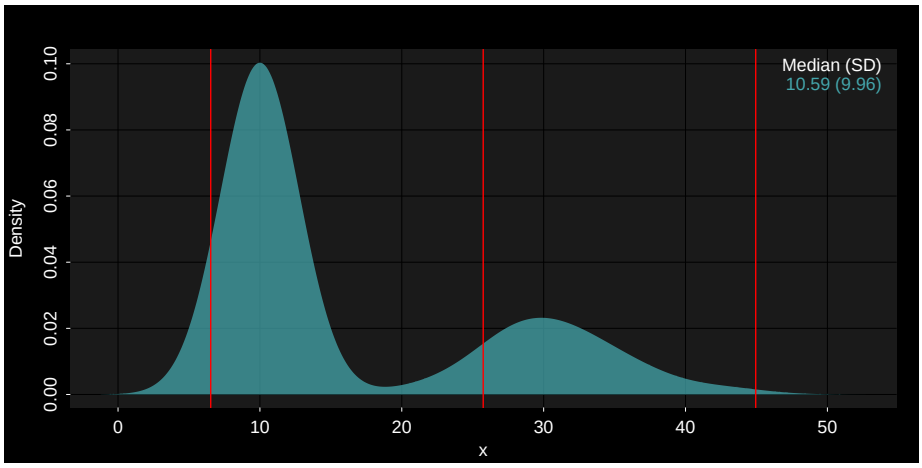
```
(xcuts3 <- seq(min(x), max(x), length.out = 3))
```

```
[1]  6.53193 25.73907 44.94622
```

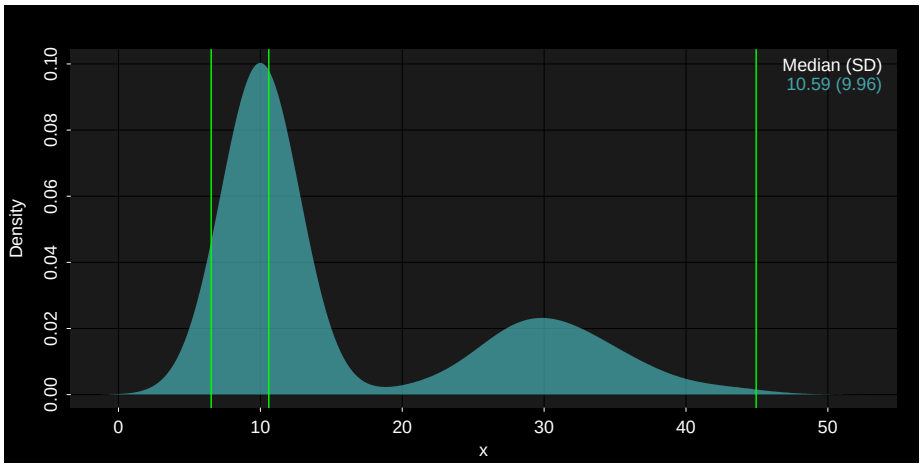
```
(xquants3 <- quantile(x, seq(0, 1, length.out = 3)))
```

```
      0%       50%      100%
 6.53193 10.59441 44.94622
```

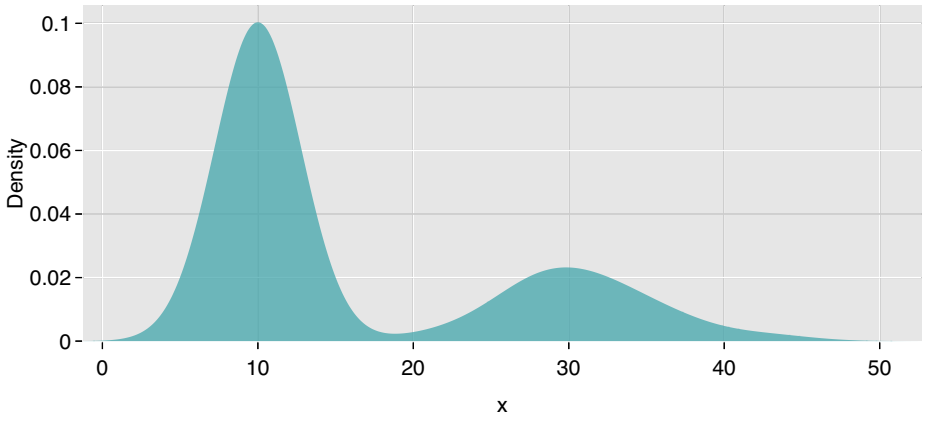
```
mplot3.x(x, par.reset = F)
abline(v = xcuts3, col = "red", lwd = 1.5)
```



```
mplot3.x(x, par.reset = F)
abline(v = xquants3, col = "green", lwd = 1.5)
```

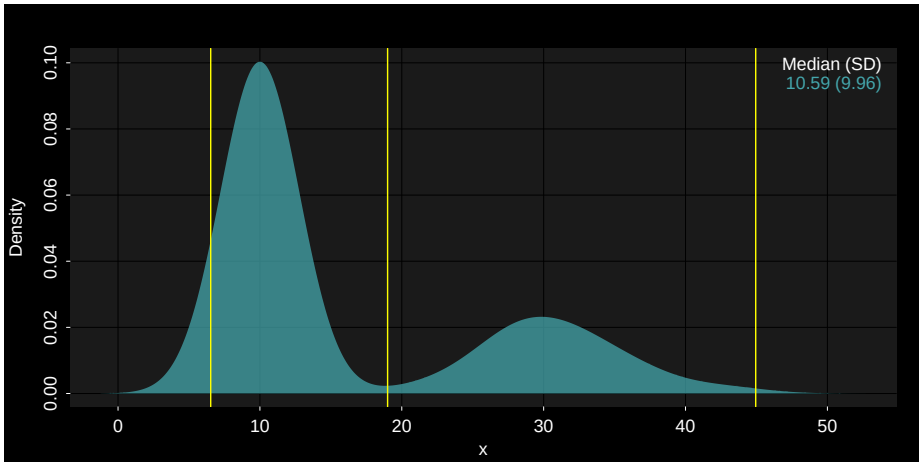



```
dplot3.x(x)
```




```
xcutm <- cut(x, breaks = c(min(x), 19, max(x)))
```

```
mplot3.x(x, par.reset = F)
abline(v = c(min(x), 19, max(x)), col = "yellow", lwd = 1.5)
```



0.90 Categorical variables

Many algorithms, or their implementations, do not support categorical variables directly and to use them, you must convert all categorical variables to some type of numeric encoding.

0.90.1 Integer encoding

If the categorical data is ordinal, you simply convert them to integers.
For example, the following ordered factor:

```
(brightness <- factor(c("bright", "brightest", "darkest",
                        "bright", "dark", "dim", "dark"),
                      levels = c("darkest", "dark", "dim", "bright", "brightest"),
                      ordered = TRUE))
```

```
[1] bright   brightest darkest  bright   dark     dim      dark
Levels: darkest < dark < dim < bright < brightest
```

...can be directly coerced to integer:

```
as.integer(brightness)
```

```
[1] 4 5 1 4 2 3 2
```

0.90.2 One-hot encoding

When categorical features are **not** ordinal, and your algorithm cannot handle them directly, you can one-hot encode them. (This is similar to creating dummy variables in statistics). In one-hot encoding, each categorical feature is converted to k binary features, where k = number of unique values in the input, such that only one feature is 1 per case.

```
admission_reasons <- c("plannedSurgery", "emergencySurgery", "medical")
(admission <- sample(admission_reasons, 10, T))
```

```
 [1] "plannedSurgery"   "emergencySurgery" "emergencySurgery" "plannedSurg
 [5] "plannedSurgery"   "plannedSurgery"   "medical"          "emergencySurge
 [9] "medical"           "plannedSurgery"
```

We can use the **rtemis** `oneHot()` function:

```
(admission_oneHot <- oneHot(admission))
```

```
    admission.emergencySurgery admission.medical admission.plannedSurger
 [1,]                          0                 0                       1
 [2,]                          1                 0                       0
 [3,]                          1                 0                       0
 [4,]                          0                 0                       1
 [5,]                          0                 0                       1
 [6,]                          0                 0                       1
 [7,]                          0                 1                       0
 [8,]                          1                 0                       0
 [9,]                          0                 1                       0
[10,]                          0                 0                       1
```

String Operations

0.91 Reminder: create - coerce - check

- `character()`: Initialize empty character vector
- `as.character()`: Coerce any vector to a character vector
- `is.character()`: Check object is character

```
x <- character(10)
```

```
v <- c(10, 20, 22, 43)
(x <- as.character(v))
```

```
[1] "10" "20" "22" "43"
```

```
x <- c("PID", "Age", "Sex", "Handedness")
is.character(x)
```

```
[1] TRUE
```

0.92 `nchar()`: Get number of characters in element

`nchar` counts the number of characters in each **element** of type character in a vector:

```
x <- c("a", "bb", "ccc")
nchar(x)
```

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

0.93 `substr()`: Get substring

```
x <- c("001Emergency", "010Cardiology", "018Neurology",
       "020Anesthesia", "021Surgery", "051Psychiatry")
substr(x, start = 1, stop = 3)
```

```
[1] "001" "010" "018" "020" "021" "051"
```

Neither `start` nor `stop` need to be valid character indices.
For example, if you want to get all characters from the fourth one to the last one, you can specify a very large `stop`

```
substr(x, 4, 99)
```

```
[1] "Emergency"  "Cardiology" "Neurology"  "Anesthesia" "Surgery"
[6] "Psychiatry"
```

If you start with too high an index, you end up with empty strings:

```
substr(x, 20, 24)
```

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

Note: `substring()` is also available, with similar syntax to `substr()`: (first, last) instead of (start, stop). It is available for compatibility with S (check its source code to see how it's an alias for `substr()`)

0.94 `strsplit()`: Split strings

```
x <- "This is one sentence"
strsplit(x, " ")
```

```
[[1]]
[1] "This"     "is"       "one"      "sentence"
```

```
x <- "In the beginning, there was the command line"
strsplit(x, ",")
```

```
[[1]]
[1] "In the beginning"         " there was the command line"
```

0.95 paste(): Concatenate strings

`paste()` and `paste0()` are particularly useful commands.
In it simplest form, it acts like `as.character()`:

```
v <- c(10, 20, 22, 43)
paste(v)
```

```
[1] "10" "20" "22" "43"
```

Combine strings from multiple vectors, elementwise:

```
id = c("001", "010", "018", "020", "021", "051")
dept = c("Emergency", "Cardiology", "Neurology",
         "Anesthesia", "Surgery", "Psychiatry")
paste(id, dept)
```

```
[1] "001 Emergency"  "010 Cardiology" "018 Neurology"  "020 Anesthesia"
[5] "021 Surgery"    "051 Psychiatry"
```

Use `sep` to define separator:

```
paste(id, dept, sep = "+++")
```

```
[1] "001+++Emergency"  "010+++Cardiology" "018+++Neurology"  "020+++Anesthesia"
[5] "021+++Surgery"    "051+++Psychiatry"
```

`paste0()` is an alias for the commonly used `paste( ... , sep = "")`:

```
paste0(id, dept)
```

```
[1] "001Emergency"  "010Cardiology" "018Neurology"  "020Anesthesia"
[5] "021Surgery"    "051Psychiatry"
```

As with other vectorized operations, value recycling can be very convenient:

```
paste0("Feature_", 1:10)
```

```
 [1] "Feature_1"  "Feature_2"  "Feature_3"  "Feature_4"  "Feature_5"
 [6] "Feature_6"  "Feature_7"  "Feature_8"  "Feature_9"  "Feature_10"
```

The argument `collapse` helps output a *single* character element after collapsing with some string:

```
paste0("Feature_", 1:10, collapse = ", ")
```

```
[1] "Feature_1, Feature_2, Feature_3, Feature_4, Feature_5, Feature_6, Fea
```

0.96 `cat()`: Concatenate and print

`cat()` concatenates strings in order to print to screen (console) or to file. It does not return any value. It is therefore useful to produce informative messages in your programs.

```
sbp <- 130
temp <- 98.4
cat("The blood pressure was", sbp, "and the temperature was", temp, "\n"
```

```
The blood pressure was 130 and the temperature was 98.4
```

0.97 String formatting

0.97.1 Change case with `toupper` and `tolower`

```
features <- c("id", "age", "sex", "sbp", "dbp", "hct", "urea", "creatini
(features <- toupper(features))
```

```
[1] "ID"       "AGE"      "SEX"      "SBP"      "DBP"
[6] "HCT"        "UREA"       "CREATININE"
```

```
(features <- tolower(features))
```

```
[1] "id"       "age"      "sex"      "sbp"      "dbp"
[6] "hct"        "urea"       "creatinine"
```

0.97.2 `abbreviate()`

```
x <- c("Emergency", "Cardiology", "Surgery", "Anesthesia", "Neurology", "Psychia
# x <- c("University of California San Francisco")
abbreviate(x)
```

```
       Emergency        Cardiology          Surgery       Anesthesia
          "Emrg"           "Crdl"           "Srgr"           "Anst"
        Neurology        Psychiatry Clinical Psychology
            "Nrlg"             "Psyc"              "ClnP"
```

```
abbreviate(x, minlength = 3)
```

```
       Emergency        Cardiology          Surgery       Anesthesia
           "Emr"            "Crd"            "Srg"            "Ans"
        Neurology        Psychiatry Clinical Psychology
             "Nrl"              "Psy"               "ClP"
```

0.98 Pattern matching

A very common task in programming is to find +/- replace string patterns in a vector of strings.
`grep` and `grepl` help find strings that contain a given pattern.
`sub` and `gsub` help find and replace strings.

0.98.1 `grep`: Get an integer index of elements that include a pattern

```
x <- c("001Age", "002Sex", "010Temp", "014SBP", "018Hct", "022PFratio", "030GCS"
grep(pattern = "SBP", x = x)
```

```
[1] 4 8
```

`grep()`'s `value` arguments which defaults to `FALSE`, allows returning the matched string itself (the value of the element) instead of its integer index:

```
grep("SBP", x, value = TRUE)
```

```
[1] "014SBP"     "112SBP-DBP"
```

0.98.2 `grepl`: Get a logical index of elements that include a pattern

`grepl` is similar to `grep`, but reuturns a logical index instead:

```
grepl("SBP", x)
```

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

0.98.3 `sub`: Find replace first match of a pattern

```
x <- c("The most important variable was PF ratio. Other significant vari
sub(pattern = "variable", replacement = "feature", x = x)
```

```
[1] "The most important feature was PF ratio. Other significant variables a
```

"First match" refers to each element of a character vector:

```
x <- c("var 1, var 2", "var 3, var 4")
sub("var", "feat", x)
```

```
[1] "feat 1, var 2" "feat 3, var 4"
```

0.98.4 `gsub`: Find and replace all matches of a pattern

```
x <- c("The most important variable was PF ratio. Other significant vari
gsub(pattern = "variable", replacement = "feature", x = x)
```

```
[1] "The most important feature was PF ratio. Other significant features ar
```

"All matches" means all matches across all elements:

```
x <- c("var 1, var 2", "var 3, var 4")
gsub("var", "feat", x)
```

```
[1] "feat 1, feat 2" "feat 3, feat 4"
```

0.99 Regular expressions

Regular expressions allow you to perform flexible pattern matching. For example, you can look for a pattern specifically at the beginning or the end of a word, or for a variable pattern with certain characteristics.
Regular expressions are very powerful and heavily used. They exist in multiple programming languages - with many similarities and a few differences.
There are many rules in defining regular expression. You can read the R manual by typing `?base::regex`.
Here are some of the most important rules:

0.99.1 Match a pattern at the beginning of a line/string with ^/\\<:

Use the caret sign ^ in the **beginning** of a pattern to only match strings that begin with this pattern.
pattern `012` matches both 2nd and 3rd elements:

```
(x <- c("001xyz993", "012qwe764", "029aqw012"))
```

```
[1] "001xyz993" "012qwe764" "029aqw012"
```

```
grep("012", x)
```

```
[1] 2 3
```

By adding ^ or \\<, only the 2nd element matches:

```
grep("^012", x)
```

```
[1] 2
```

```
grep("\\<012", x)
```

```
[1] 2
```

0.99.2 Match a pattern at the end of a line/string with $/\\>

The dollar sign $ is used at the **end** of a pattern to only match strings which end with this pattern:

```
x
```

```
[1] "001xyz993" "012qwe764" "029aqw012"
```

```
grep("012$", x)
```

```
[1] 3
```

```
grep("012\\>", x)
```

```
[1] 3
```

```
x <- c("1one", "2one", "3two", "3three")
grep("one$", x)
```

```
[1] 1 2
```

```
grep("one\\>", x)
```

```
[1] 1 2
```

0.99.3 . : Match any character

```
grep("e.X", c("eX", "enX", "ennX", "ennnX", "ennnnX"))
```

```
[1] 2
```

0.99.4 +: Match preceding character one or more times:

```
grep("en+X", c("eX", "enX", "ennX", "ennnX", "ennnnX"))
```

```
[1] 2 3 4 5
```

0.99.5 {n}: Match preceding character n times:

```
grep("en{2}X", c("eX", "enX", "ennX", "ennnX", "ennnnX"))
```

```
[1] 3
```

0.99.6 {n,}: Match preceding character n or more times:

```
grep("en{2,}X", c("eX", "enX", "ennX", "ennnX", "ennnnX"))
```

```
[1] 3 4 5
```

0.99.7 {n,m}: Match preceding character at least n times and no more than m times:

```
grep("en{2,3}X", c("eX", "enX", "ennX", "ennnX", "ennnnX"))
```

```
[1] 3 4
```

0.99.8 Escaping metacharacters

The following are defined as metacharacters, because they have special meaning within a regular expression: `. \ | ( ) [ { ^ $ * + ?`.
If you want to match one of these characters itself, you must "escape" it using a double backslash:

```
x <- c("dn3ONE", "d.3TWO", "dx3FIVE")
grep("d\\.3", x)
```

```
[1] 2
```

0.99.9 Match a character class

You can use brackets, `[ ]` to define sets of characters to match in any order, if present. Here we want to replace `$` and `@` with an underscore:

```
x <- c("Feat1$alpha", "Feat2$gamma", "Feat9@field2")
gsub("[$@]", "_", x)
```

```
[1] "Feat1_alpha"  "Feat2_gamma"  "Feat9_field2"
```

A number of character classes are predefined. They are themselves surrounded by brackets - to use them as a character class, you need a seconds set of brackets around them. Some of the most common ones include:

- `[:alnum:]`: alphanumeric, i.e. all letters and numbers
- `[:alpha:]`: all letters
- `[:digit:]`: all numbers
- `[:lower:]`: all lowercase letters
- `[:upper:]`: all uppercase letters
- `[:punct:]`: all punctuation characters (! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ ] ^ _ ` { | } ~.)
- `[:blank:]`: all spaces and tabs
- `[:space:]`: all spaces, tabs, newline characters, and some more

Let's look at some examples.
Here we us `[:digit:]` to remove all numbers:

```
x <- c("001Emergency", "010Cardiology", "018Neurology", "020Anesthesia",
       "021Surgery", "051Psychiatry")
gsub("[[:digit:]]", "", x)
```

```
[1] "Emergency"  "Cardiology" "Neurology"  "Anesthesia" "Surgery"
[6] "Psychiatry"
```

We can use `[:alpha:]` to remove all letters:

```
gsub("[[:alpha:]]", "", x)
```

```
[1] "001" "010" "018" "020" "021" "051"
```

We can use a caret ^ in the beginning of a character class to match any character *not* in the character set:

```
x <- c("001$Emergency", "010@Cardiology", "018*Neurology", "020!Anesthes
       "021!Surgery", "051*Psychiatry")
gsub("[^[:alnum:]]", "_", x)
```

```
[1] "001_Emergency"  "010_Cardiology" "018_Neurology"  "020_Anesthesia"
[5] "021_Surgery"    "051_Psychiatry"
```

0.99.10 Combine character classes

Use | to match from multiple character classes:

```
x <- c("123#$%alphaBeta")
gsub("[[:digit:]|[:punct:]]", "", x)
```

```
[1] "alphaBeta"
```

For more information on regular expressions, start by reading the builtin documentation: `?regex`

Dates

R includes builtin support for working with dates and/or time data and a number of packages exist that extend this support.
There are three builtin classes:

- `Date`: Represents **dates** (not time)
- `POSIXct`: Represents **dates and time** as the signed number of seconds since January 1, 1970
- `POSIXlt`: Represents **dates and time** as a named list of vectors (See `base::DateTimeClasses`)

Background info: Portable Operating System Interface (POSIX)[13] is a set of standards for maintaining compatibility among operating systems.

0.100 `as.Date()`: Character to Date

You can create a `Date` object from a string:

```
(x <- as.Date("1981-02-12"))
```

```
[1] "1981-02-12"
```

```
class(x)
```

```
[1] "Date"
```

The `tryFormats` argument defines which formats are recognized. The default is `tryFormats = c("%Y-%m-%d", "%Y/%m/%d")`.

[13] `https://en.wikipedia.org/wiki/POSIX`

0.101 `sys.Date()`: Get today's date

```
(today <- Sys.Date())
```

```
[1] "2020-10-19"
```

```
class(today)
```

```
[1] "Date"
```

0.102 Time intervals

The reason we care about Date objects in R is because we can do operations with them, i.e. we can substract date objects to get time intervals.
For example to get someone's age in days:

```
dob <- as.Date("1973-09-14")
Sys.Date() - dob
```

```
Time difference of 17202 days
```

Note: While you can use the subtraction operator `-`, it is advised you always use the `difftime()` function to perform subtraction on dates instead, because it allows you to specify units:

```
timepoint1 <- as.Date("2020-01-07")
timepoint2 <- as.Date("2020-02-03")
(interval_in_weeks <- difftime(timepoint2, timepoint1, units = "weeks"))
```

```
Time difference of 3.857143 weeks
```

```
(interval_in_days <- difftime(timepoint2, timepoint1, units = "days"))
```

```
Time difference of 27 days
```

```
(interval_in_hours <- difftime(timepoint2, timepoint1, units = "hours"))
```

```
Time difference of 648 hours
```

```
(interval_in_minutes <- difftime(timepoint2, timepoint1, units = "mins"))
```

```
Time difference of 38880 mins
```

```
(interval_in_seconds <- difftime(timepoint2, timepoint1, units = "secs"))
```

```
Time difference of 2332800 secs
```

0.103 as.POSIXct, as.POSIXlt, strptime: Character to Date-Time

As always, it can be very informative to look at the source code. For the common use case of converting a character to a Date-Time object, `as.POSIXct.default()` calls `as.POSIXlt.character()`, which calls `strptime()`.
See `?strptime` for conversion specifications. These define how characters are read as year - month - day - hour - minute - second information.
For example, the ISO 8601 internation standard is defined as:
`"%Y-%m-%d %H:%M:%S"`

- %Y: Year with century, (0-9999 accepted) e.g. `2020`
- %m: Month, 01-12, e.g. `03`
- %d: Day, 01-31, e.g. `04`
- %H: Hours, 00-23, e.g. `13`
- %M: Minutes, 00-59, e.g. `38`
- %S: Seconds, 00-61 (!) allowing for up to two leap seconds, e.g. `54`

```
(dt <- c("2020-03-04 13:38:54"))
```

```
[1] "2020-03-04 13:38:54"
```

```
dt_posix <- as.POSIXct(dt)
class(dt_posix)
```

```
[1] "POSIXct" "POSIXt"
```

You can compose a really large number of combination formats to match your data.

```
dt2 <- c("03.04.20 01:38.54 pm")
(dt2_posix <- as.POSIXct(dt2, format = "%m.%d.%y %I:%M.%S %p"))
```

```
[1] "2020-03-04 13:38:54 PST"
```

0.104 Bring the guessing in with `lubridate`

Instead of defining Date and/or time formats, we can use the `lubridate` package to do some guesswork for us, which works very well most of the time.

```
library(lubridate)
```

```
Attaching package: 'lubridate'

The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

```
dt <- c("2020-03-04 13:38:54")
(dt_posix <- as_datetime(dt))
```

```
[1] "2020-03-04 13:38:54 UTC"
```

Note that timezone defaults to UTC (Coordinated Universal Time[14]) and must be set manually. PST is defined with "America/Los_Angeles" or the (officially deprecated) "US/Pacific" (tz database[15])

```
dt_posix <- as_datetime(dt, tz = "America/Los_Angeles")
```

```
dt2_posix <- as_datetime(dt2)
```

`dt2` got misinterpreted as year-month-day.
For these cases, `lubridate` includes a number of convenient functions to narrow down the guessing. The functions are named using all permutations of `y`, `m`, and `d`. The letter order signifies the order the information appears in the character you are trying to import, i.e. `ymd`, `dmy`, `mdy`, `ydm`, `myd`

```
dt2 <- c("03.04.20 01:38.54 pm")
(dt2_posix <- mdy_hms(dt2, tz = "America/Los_Angeles"))
```

```
[1] "2020-03-04 13:38:54 PST"
```

[14] https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[15] https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

0.105 format() Dates

format() operates on Date and POSIX objects to convert between representations

```
(dt_us <- as.Date("07-04-2020", format = "%m-%d-%Y"))
```

```
[1] "2020-07-04"
```

```
(dt_eu <- format(dt_us, "%d.%m.%y"))
```

```
[1] "04.07.20"
```

Handling Missing data

```
library(rtemis)
```

```
  .:rtemis 0.8.0: Welcome, egenn
 [x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
 Documentation & vignettes: https://rtemis.lambdamd.org
```

Missing data is a very common problem in statistics and data science. Data may be missing for a variety of reasons. We often characterize the type of missingness using the following three types:

- **Missing completely at random (MCAR)** "The fact that the data are missing is independent of the observed and unobserved data"
- **Missing at random (MAR)** "The fact that the data are missing is systematically related to the observed but not the unobserved data"
- **Missing not at random (MNAR)** "The fact that the data are missing is systematically related to the unobserved data"

0.106 Check for missing data

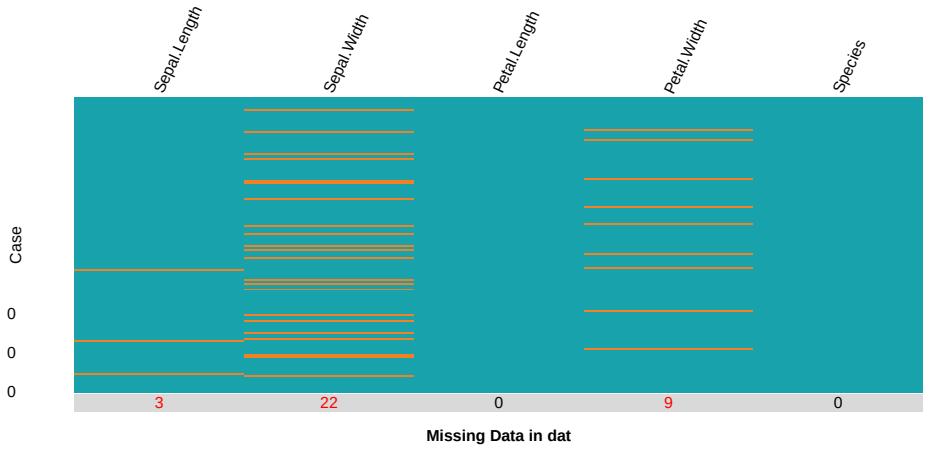
You can use your favorite base commands to check for missing data, by row, by column, total, etc.
Let's add some NA values to our favorite dataset:

```
dat <- iris
set.seed(2020)
dat[sample(1:150, 3), 1] <- dat[sample(1:150, 22), 2] <- dat[sample(1:150, 9), 4
```

0.106.1 Visualize

You can visualize missing data. A number of packages include functions to do this. I added a simple function in `rtemis`, `mplot.missing()`. In this examples, missing cases are represented in orange:

```
library(rtemis)
mplot.missing(dat)
```



0.106.2 Summarize

Get N of missing per column:

```
sapply(dat, function(i) sum(is.na(i)))
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
           3           22            0            9            0
```

The `checkData()` function in `rtemis` includes information on missing data:

```
checkData(dat)
```

0.107 Handle missing data

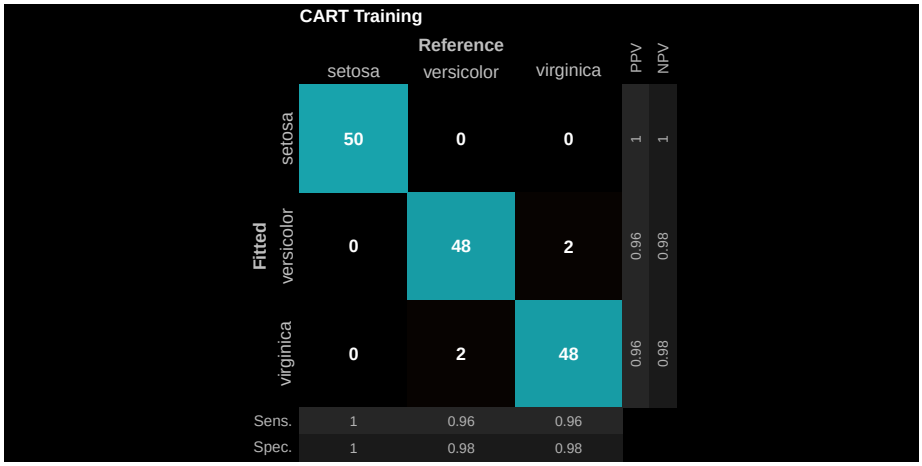
Different approaches can be used to handle missing data:

- Do nothing! - if your algorithm(s) can handle missing data (decision trees!)
- Exclude data: Use complete cases only
- Make up data: Replace or Impute
  - Replace with median/mean
  - Predict missing from present
    * Single imputation
    * Multiple imputation

0.107.1 Do nothing (decision trees!)

Decision trees and ensemble methods that use decision trees like random forest and gradient boosting.

```
dat.cart <- s.CART(dat)
```



0.107.2 Use complete cases only

R's builtin `complete.cases()` function returns a logical index of cases that have no missing values, i.e. are complete.

```
dim(dat)
```

```
[1] 150   5
```

```
(index_cc <- complete.cases(dat))
```

```
 [1] TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
[13] TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
[25] TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
[37] TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[49] TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
[61] TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
[73] TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
[85] TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
[97] TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[109] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[121]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE
[133]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
[145]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
dat_cc <- dat[index_cc, ]
dim(dat_cc)
```

```
[1] 116   5
```

We lost 34 cases in the above example. Maybe that's a lot.

0.107.3 Replace with a fixed value: mean, median vs. mode, "missing"

We can manually replace missing values with the mean or median for continuous variables, or with the mode for categorical features.
For example to replace the first feature's missing values with the mean

```
Sepal.Length_mean <- mean(dat$Sepal.Length, na.rm = TRUE)
dat_rm <- dat
dat_rm$Sepal.Length[is.na(dat_rm$Sepal.Length)] <- Sepal.Length_mean
```

The `preprocess()` function in `rtemis` can do this for you as well for all features:

```
dat_pre <- preprocess(dat, impute = TRUE, impute.type = "meanMode")
```

Verify there are no missing data by rerunning `checkData()`:

```
checkData(dat_pre)
```

You may want to include a "missingness" column that indicates which cases were imputed to include in your model. You can create this simply by running:

```
Sepal.Length_missing = factor(as.integer(is.na(dat$Sepal.Length)))
```

`preprocess()` includes the option `missingness` to add corresponding indicator columns after imputation:

```
dat_pre <- preprocess(dat, impute = TRUE, impute.type = "meanMode",
                      missingness = TRUE)
```

```
head(dat_pre)
```

```
  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
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
  Sepal.Length_missing Sepal.Width_missing Petal.Width_missing
1                    0                   0                   0
2                    0                   0                   0
3                    0                   0                   0
4                    0                   0                   0
5                    0                   0                   0
6                    0                   0                   0
```

With categorical variables, an alternative option would be to introduce a new level of "missing" to your data, instead of replacing with the mode, for example. If we bin a continuous variable to convert to categorical, the same can then also be applied.

(-> I will add a function to `preprocess()` to do this.)

0.107.4 Last observation carried forward

In longitudinal / timeseries data, we may want to replace missing values with the last observed value. This is called last observation carried forward (LOCF). As always, whether this procedure is appropriate depend the reasons for missingness. The `zoo` and `DescTool` packages provide commands to perform LOCF.

Some simulated data. We are missing blood pressure measurements on Saturdays and Sundays:

```
dat <- data.frame(Day = rep(c("Mon", "Tues", "Wed", "Thu", "Fri", "Sat",
                  SBP = sample(105:125, 21, TRUE))
dat$SBP[dat$Day == "Sat" | dat$Day == "Sun"] <- NA
dat
```

```
    Day SBP
1   Mon 117
2  Tues 106
3   Wed 120
4   Thu 117
5   Fri 105
6   Sat  NA
7   Sun  NA
8   Mon 117
9  Tues 115
10  Wed 109
11  Thu 115
12  Fri 110
13  Sat  NA
14  Sun  NA
15  Mon 122
16 Tues 105
17  Wed 111
18  Thu 112
19  Fri 125
20  Sat  NA
21  Sun  NA
```

The `zoo` package includes the `na.locf()`.

```
dat$SBPlocf <- zoo::na.locf(dat$SBP)
dat
```

```
    Day SBP SBPlocf
1   Mon 117     117
2  Tues 106     106
3   Wed 120     120
4   Thu 117     117
5   Fri 105     105
6   Sat  NA     105
7   Sun  NA     105
8   Mon 117     117
9  Tues 115     115
10  Wed 109     109
```

```
11  Thu 115     115
12  Fri 110     110
13  Sat  NA     110
14  Sun  NA     110
15  Mon 122     122
16 Tues 105     105
17  Wed 111     111
18  Thu 112     112
19  Fri 125     125
20  Sat  NA     125
21  Sun  NA     125
```

Similar functionality is included in `DescTools`' `LOCF()` function:

```
DescTools::LOCF(dat$SBP)
```

```
 [1] 117 106 120 117 105 105 105 117 115 109 115 110 110 110 122 105 111 112 125
[20] 125 125
```

0.107.5 Replace missing values with estimated values

0.107.5.1 Single imputation

You can use non-missing data to predict missing data in an iterative procedure (Buuren and Groothuis-Oudshoorn, 2010)(Stekhoven and Bühlmann, 2012). The `missRanger` package uses the optimized (and parallel-capable) package `ranger` (Wright and Ziegler, 2015) to iteratively train random forest models for imputation.

```
library(missRanger)
dat <- iris
set.seed(2020)
dat[sample(1:150, 5), 1] <- dat[sample(1:150, 22), 4] <- dat[sample(1:150, 18),
dat_rfimp <- missRanger(dat, num.trees = 100)
```

```
Missing value imputation by random forests

  Variables to impute:      Sepal.Length, Petal.Width
  Variables used to impute: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, S
iter 1: ..
iter 2: ..
iter 3: ..
iter 4: ..
```

```
head(dat_rfimp)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1     5.100000         3.5          1.4         0.2  setosa
2     4.900000         3.0          1.4         0.2  setosa
3     4.732533         3.2          1.3         0.2  setosa
4     4.600000         3.1          1.5         0.2  setosa
5     5.000000         3.6          1.4         0.2  setosa
6     5.400000         3.9          1.7         0.4  setosa
```

```
checkData(dat_rfimp)
```

Note: The default method for `preprocess(impute = TRUE)` is to use `missRanger`.

0.107.5.2 Multiple imputation

Multiple imputation creates multiple estimates of the missing data. It is more statistically valid for small datasets, but may not be practical for larger datasets. The package `mice` is a popular choice for multiple imputation in R.

```
library(mice)
dat_mice <- mice(dat)
```

Classes and Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm built around objects with associated data, known as attributes, and functions, known as methods.

There are 4 main class systems in R:

- S3: informally defined, minimal, lean, methods dispatch based on class; `base` and `stats` packages use S3 exclusively
- S4: formally defined, allows method dispatch on multiple arguments
- RC: Reference class: Reference semantics; similar to other programming languages; methods are part of the object
- R6: 3rd party class system similar to RC, more lightweight, faster

S3 and S4 methods are part of generic functions. RC and R6 methods are part of the object, but you can (and should) write generic functions for them as well.

This chapter will focus on the ubiquitous S3 system. For more advanced (and real OOP) applications, we recommend looking into the R6[16] system.

0.108 S3

Most R objects we have been using so far are S3 objects. Data frames are some of the most common S3 objects.

Generic functions are functions that act differently based on the class of the input object. We have already used many of them. For example, `summary()` works differently on a `data.frame`, on a `factor`, or a `glm` object, etc.

Generic functions in R are saved as `functionname.classname()` and called automatically, based on the class of the first argument. This allows the same function, e.g. `print()`, `summary()`, `c()`, to have a different effect on objects of different classes. For example, the `print()` function applied on a data frame,

[16] `https://r6.r-lib.org/index.html`

will actually call `print.data.frame()`, while applied on a factor, it will call `print.factor()`.
This means that when you type `print(iris)` this calls `print.data.frame(iris)`

Note how the R documentation lists usage information separately for each S3 method, e.g. `## S3 method for class 'factor'`

0.108.1 `methods()`

To get a list of all available methods defined for a specific class,
i.e. ***"What different functions can I use on this object?"***

```
methods(class = "data.frame")
```

```
 [1] [           [[          [[<-        [<-         $<-
 [6] aggregate   anyDuplicated anyNA      as.data.frame as.list
[11] as.matrix   by          cbind       coerce      dim
[16] dimnames    dimnames<-  droplevels   duplicated  edit
[21] format      formula     head        initialize  is.na
[26] Math        merge       na.exclude  na.omit     Ops
[31] plot        print       prompt      rbind       row.names
[36] row.names<- rowsum      show        slotsFromS3 split
[41] split<-     stack       str         subset      summary
[46] Summary     t           tail        transform   type.convert
[51] unique      unstack     within
see '?methods' for accessing help and source code
```

Conversely, to get a list of all available methods for a generic function (i.e. which classes have)
(i.e. "What objects can I use this function on?")

```
methods(generic.function = "plot")
```

```
 [1] plot.acf*        plot.data.frame*   plot.decomposed.ts*
 [4] plot.default     plot.dendrogram*   plot.density*
 [7] plot.ecdf        plot.factor*       plot.formula*
[10] plot.function    plot.hclust*       plot.histogram*
[13] plot.HoltWinters* plot.isoreg*      plot.lm*
[16] plot.medpolish*  plot.mlm*         plot.ppr*
[19] plot.prcomp*     plot.princomp*     plot.profile.nls*
[22] plot.raster*     plot.spec*         plot.stepfun
[25] plot.stl*        plot.table*        plot.ts
[28] plot.tskernel*   plot.TukeyHSD*
see '?methods' for accessing help and source code
```

0.108.2 Defining custom S3 classes

It very simple to assign an object to a new class.
There is no formal class definition, an object is directly assigned to a class by name. An object can belong to multiple classes:

```
x <- 1:10
class(x) <- c("specialvector", "numeric")
class(x)
```

```
[1] "specialvector" "numeric"
```

The hierarchy of classes goes left to right, meaning that generic methods are searched for classes in the order they appear in the output of `class()`.

If we print `x`, since there is no print method for class `specialvector` or for `numeric`, the default `print.default()` command is automatically called:

```
print(x)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"class")
[1] "specialvector" "numeric"
```

```
print.default(x)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"class")
[1] "specialvector" "numeric"
```

To create a custom `print()` function for out new class `specialvector`, we define a function named `print.[classname]`:

```
print.specialvector <- function(x, ...) {
  cat("This is a special vector of length", length(x), "\n")
  cat("Its mean value is", mean(x, na.rm = TRUE), "and its median is", median(x,
  cat("\nHere are the first few elements:\n", head(x), "\n")
}
```

Now, when you print an object of class `specialvector`, the custom `print()` command is invoked:

```
x
```

```
This is a special vector of length 10
Its mean value is 5.5 and its median is 5.5
Here are the first few elements:
 1 2 3 4 5 6
```

If needed, you can call the default or another appropriate method directly:

```
print.default(x)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"class")
[1] "specialvector" "numeric"
```

You can change the vector back to a regular numeric vector, or a different class, just as easily:

```
class(x) <- "numeric"
x
```

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

Efficient data analysis with data.table

The **data.table**[17] package provides a modern and highly optimized version of R's data.frame structure. It is highly memory efficient and automatically parallelizes internal operations to achieve substantial speed improvements over data.frames. The `data.table` package weighs in at just a few kilobytes, has zero dependencies, and maintains compatibility with R versions going as far back as possible.

There are two main ways in which `data.table` differs from `data.frame`:

- You can perform many operations ***"in-place"*** without creating a copy (i.e. make changes to a `data.table` without having to assign it back to itself).
- There is a lot more than indexing and slicing that you can do within a `data.table`'s "frame" i.e. the square brackets after a `data.table`, like applying any custom function to specific columns and/or cases.

```
library(rtemis)
```

```
  .:rtemis 0.8.0: Welcome, egenn
 [x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
 Documentation & vignettes: https://rtemis.lambdamd.org
```

```
library(data.table)
```

```
Attaching package: 'data.table'
```

```
The following object is masked from 'package:rtemis':

    cube
```

Let's look at `data.table` vs. `data.frame` operations:

[17] `https://github.com/Rdatatable/data.table`

0.109 Create a `data.table`

0.109.1 By assignment: `data.table()`

Same syntax with `data.frame()`:

```
(df <- data.frame(A = 1:5,
                  B = c(1.2, 4.3, 9.7, 5.6, 8.1),
                  C = c("a", "b", "b", "a", "a")))
```

```
  A   B C
1 1 1.2 a
2 2 4.3 b
3 3 9.7 b
4 4 5.6 a
5 5 8.1 a
```

```
class(df)
```

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

```
(dt <- data.table(A = 1:5,
                  B = c(1.2, 4.3, 9.7, 5.6, 8.1),
                  C = c("a", "b", "b", "a", "a")))
```

```
   A   B C
1: 1 1.2 a
2: 2 4.3 b
3: 3 9.7 b
4: 4 5.6 a
5: 5 8.1 a
```

```
class(dt)
```

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

Notice how `data.table` inherits from `data.frame`. This means that if a method does not exist for `data.table`, the method for `data.frame` will be used.
One difference from `data.frame()` is that, as you can see above, is that `stringsAsFactors` defaults to FALSE in `data.table()`. Also, as part of efficiency improvements, data.tables do away with row names, which are rarely

used. Instead of using rownames, you can always add an extra column with the same information - this is advisable when working with data.frame as well.

```
(dt <- data.table(A = 1:5,
                  B = c(1.2, 4.3, 9.7, 5.6, 8.1),
                  C = c("a", "b", "b", "a", "a"),
                  stringsAsFactors = TRUE))
```

```
   A   B C
1: 1 1.2 a
2: 2 4.3 b
3: 3 9.7 b
4: 4 5.6 a
5: 5 8.1 a
```

0.109.2 By coercion: `as.data.table()`

```
dat <- data.frame(A = 1:5,
                  B = c(1.2, 4.3, 9.7, 5.6, 8.1),
                  C = c("a", "b", "b", "a", "a"))
(dat2 <- as.data.table(dat))
```

```
   A   B C
1: 1 1.2 a
2: 2 4.3 b
3: 3 9.7 b
4: 4 5.6 a
5: 5 8.1 a
```

0.109.3 By coercion *in-place*: `setDT()`

`setDT` converts a list or data.frame into a `data.table` in-place. Note: the original object itself is changed, you do not need to assign the output of `setDT` to a new name.

```
dat <- data.frame(A = 1:5,
                  B = c(1.2, 4.3, 9.7, 5.6, 8.1),
                  C = c("a", "b", "b", "a", "a"))
class(dat)
```

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

```
setDT(dat)
class(dat)
```

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

You can similarly convert a `data.table` to a `data.frame`, in-place:

```
setDF(dat)
class(dat)
```

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

0.109.4 Read into `data.table` from file

data.table includes the `fread()` function to read data from files, in a similar way as the base functions `read.csv()` and `read.table()`. It is orders of magnitude faster for very large data (e.g. thousands to millions of rows) and it can read directly from URLs, and zipped files. The `sep` arguments defines the separator (same as in `read.csv()` and `read.table()`), but when set to `"auto"` (the default) it does a great job of figuring it out by itself.

```
dat <- fread("path/to/file.csv")
dat <- fread("https://url/to/file.csv.gz")
```

For its speed and convenience, `fread()` is recommended over `read.csv()`/`read.table()` even if you intend to work with a `data.frame` exclusively, in which case you can pass the argument `data.table = FALSE` to `fread()`

0.109.5 Write a `data.table` to file: `fwrite()`

```
fwrite(dt, "/path/to/file.csv")
```

0.110 Combine `data.table`s

`cbind()` and `rbind()` work on `data.table`s the same as on `data.frame`s:

```
dt1 <- data.table(a = 1:5)
dt2 <- data.table(b = 11:15)
cbind(dt1, dt2)
```

```
   a  b
1: 1 11
2: 2 12
3: 3 13
4: 4 14
5: 5 15
```

```
rbind(dt1, dt1)
```

```
    a
 1: 1
 2: 2
 3: 3
 4: 4
 5: 5
 6: 1
 7: 2
 8: 3
 9: 4
10: 5
```

0.111 `str` works the same (and you should keep using it!)

```
str(df)
```

```
'data.frame':   5 obs. of  3 variables:
 $ A: int  1 2 3 4 5
 $ B: num  1.2 4.3 9.7 5.6 8.1
 $ C: chr  "a" "b" "b" "a" ...
```

```
str(dt)
```

```
Classes 'data.table' and 'data.frame':  5 obs. of  3 variables:
 $ A: int  1 2 3 4 5
 $ B: num  1.2 4.3 9.7 5.6 8.1
 $ C: Factor w/ 2 levels "a","b": 1 2 2 1 1
```

```
 - attr(*, ".internal.selfref")=<externalptr>
```

0.112 Indexing a `data.table`

Indexing is largely unchanged, with a few notable exceptions.
Integer indexing is mostly the same:

```
df[1, ]
```

```
  A   B C
1 1 1.2 a
```

```
dt[1, ]
```

```
   A   B C
1: 1 1.2 a
```

Selecting a single column with integer indexing in `data.table` does not drop to a vector (i.e. similar to `drop = FALSE` in a data.frame):

```
df[, 1]
```

```
[1] 1 2 3 4 5
```

```
df[, 1, drop = FALSE]
```

```
  A
1 1
2 2
3 3
4 4
5 5
```

```
dt[, 1]
```

```
   A
1: 1
2: 2
3: 3
4: 4
5: 5
```

In `data.table`, you can access column names directly without quoting or using `$`:

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

```
[1] 1.2 4.3 9.7 5.6 8.1
```

```
df$B
```

```
[1] 1.2 4.3 9.7 5.6 8.1
```

```
dt[, B]
```

```
[1] 1.2 4.3 9.7 5.6 8.1
```

```
df[df$B > 5, ]
```

```
  A   B C
3 3 9.7 b
4 4 5.6 a
5 5 8.1 a
```

```
with(df, df[B > 5, ])
```

```
  A   B C
3 3 9.7 b
4 4 5.6 a
5 5 8.1 a
```

```
dt[B > 5, ]
```

```
   A   B C
1: 3 9.7 b
2: 4 5.6 a
3: 5 8.1 a
```

Think of working inside the `data.table` frame (i.e. the "[...]") like an environment. You have direct access to the variables within it.
If you want to refer to variables outside the `data.table`, prefix the variable name with `..`. This is similar to how you access contents of a directory above your current directory in the terminal:

```
varname = "C"
df[, varname]
```

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

```
dt[, ..varname]
```

```
   C
1: a
2: b
3: b
4: a
5: a
```

This tells the `data.table` "don't look for 'varname' in the data.table, go outside to find it"

0.112.1 Conditionally select cases:

It is easy to select cases by combining conditions by using column names directly. Note that `data.table` does not require you to add "," to select all columns after you have specified rows - works just the same if you so include it:

There are a few way to conditionally select in a data.frame:

```
df[df$A > mean(df$A) & df$B > mean(df$B), ]
```

```
  A   B C
5 5 8.1 a
```

```
subset(df, A > mean(A) & B > mean(B))
```

```
  A   B C
5 5 8.1 a
```

```
with(df, df[A > mean(A) & B > mean(B), ])
```

```
  A   B C
5 5 8.1 a
```

The data.table equivalent is probably simplest:

```
dt[A > mean(A) & B > mean(B)]
```

```
   A   B C
1: 5 8.1 a
```

```
(a <- rnormmat(10, 5, seed = 2020, return.df = TRUE))
```

```
          V1         V2         V3         V4         V5
1  0.3769721 -0.85312282  2.17436525 -0.81250466  0.90850113
2  0.3015484  0.90925918  1.09818265 -0.74370217 -0.50505960
3 -1.0980232  1.19637296  0.31822032  1.09534507 -0.30100401
4 -1.1304059 -0.37158390 -0.07314756  2.43537371 -0.72603598
5 -2.7965343 -0.12326023  0.83426874  0.38811847 -1.18007703
6  0.7205735  1.80004312  0.19875064  0.29062767  0.25307471
7  0.9391210  1.70399588  1.29784138 -0.28559829 -0.37071130
8 -0.2293777 -3.03876461  0.93671831  0.07601472  0.02217956
9  1.7591313 -2.28897495 -0.14743319 -0.56029860  0.66004412
10 0.1173668  0.05830349  0.11043199  0.44718837  0.48879364
```

```
a[1, 3] <- a[3, 4] <- a[5, 3] <- a[7, 3] <- NA
adt <- as.data.table(a)
```

```
a[!is.na(a$V3), ]
```

```
          V1         V2         V3         V4         V5
2  0.3015484  0.90925918  1.09818265 -0.74370217 -0.50505960
3 -1.0980232  1.19637296  0.31822032         NA -0.30100401
4 -1.1304059 -0.37158390 -0.07314756  2.43537371 -0.72603598
6  0.7205735  1.80004312  0.19875064  0.29062767  0.25307471
8 -0.2293777 -3.03876461  0.93671831  0.07601472  0.02217956
9  1.7591313 -2.28897495 -0.14743319 -0.56029860  0.66004412
10 0.1173668  0.05830349  0.11043199  0.44718837  0.48879364
```

```
adt[!is.na(V3)]
```

```
          V1         V2         V3         V4         V5
1:  0.3015484  0.90925918  1.09818265 -0.74370217 -0.50505960
2: -1.0980232  1.19637296  0.31822032         NA -0.30100401
3: -1.1304059 -0.37158390 -0.07314756  2.43537371 -0.72603598
4:  0.7205735  1.80004312  0.19875064  0.29062767  0.25307471
5: -0.2293777 -3.03876461  0.93671831  0.07601472  0.02217956
```

```
6:  1.7591313 -2.28897495 -0.14743319 -0.56029860  0.66004412
7:  0.1173668  0.05830349  0.11043199  0.44718837  0.48879364
```

0.112.2 Select columns

by integer index, same as with a data.frame

```
dt[, 2]
```

```
     B
1: 1.2
2: 4.3
3: 9.7
4: 5.6
5: 8.1
```

```
dt[, 2:3]
```

```
     B C
1: 1.2 a
2: 4.3 b
3: 9.7 b
4: 5.6 a
5: 8.1 a
```

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

```
   A C
1: 1 a
2: 2 b
3: 3 b
4: 4 a
5: 5 a
```

by name: selecting a single column by name returns a vector:

```
dt[, A]
```

```
[1] 1 2 3 4 5
```

by name: selecting one or more columns by name enclosed in `.()` which, in this case, is short for `list()`, return a `data.table`:

```
dt[, .(A)]
```

```
   A
1: 1
2: 2
3: 3
4: 4
5: 5
```

```
dt[, .(A, B)]
```

```
   A   B
1: 1 1.2
2: 2 4.3
3: 3 9.7
4: 4 5.6
5: 5 8.1
```

0.113 Add new columns *in-place*

Use `:=` assignment to add a new column in the existing `data.table`. Once again, in-place means you do not have to assign the result to a variable, the existing `data.table` will be changed.

```
dt[, AplusC := A + C]
```

```
Warning in Ops.factor(A, C): '+' not meaningful for factors
```

```
dt
```

```
   A   B C AplusC
1: 1 1.2 a     NA
2: 2 4.3 b     NA
3: 3 9.7 b     NA
4: 4 5.6 a     NA
5: 5 8.1 a     NA
```

0.114 Add multiple columns *in-place*

To add multiple columns, use `:=` in a little more awkward notation:

```
dt[, `:=`(AminusC = A - C, AoverC = A / C)]
```

```
Warning in Ops.factor(A, C): '-' not meaningful for factors
Warning in Ops.factor(A, C): '/' not meaningful for factors
```

```
dt
```

```
   A   B C AplusC AminusC AoverC
1: 1 1.2 a     NA       NA     NA
2: 2 4.3 b     NA       NA     NA
3: 3 9.7 b     NA       NA     NA
4: 4 5.6 a     NA       NA     NA
5: 5 8.1 a     NA       NA     NA
```

0.115 Convert column type

Use any base R coercion function (`as.*`) to convert a column in-place using the `:=` notation

```
dt[, A := as.numeric(A)]
dt
```

```
   A   B C AplusC AminusC AoverC
1: 1 1.2 a     NA       NA     NA
2: 2 4.3 b     NA       NA     NA
3: 3 9.7 b     NA       NA     NA
4: 4 5.6 a     NA       NA     NA
5: 5 8.1 a     NA       NA     NA
```

0.116 Delete column in-place

To delete a column, use `:=` to set it to NULL:

```
dt[, AoverC := NULL]
dt
```

```
   A   B C AplusC AminusC
1: 1 1.2 a     NA       NA
2: 2 4.3 b     NA       NA
```

```
3: 3 9.7 b     NA      NA
4: 4 5.6 a     NA      NA
5: 5 8.1 a     NA      NA
```

Same awkward notation as earlier to delete multiple columns:

```
dt[, `:=`(AplusC = NULL, AminusC = NULL)]
dt
```

```
   A   B C
1: 1 1.2 a
2: 2 4.3 b
3: 3 9.7 b
4: 4 5.6 a
5: 5 8.1 a
```

0.117 Summarize

Create a *new* data.table using any summary function:

```
Asummary <- dt[, .(Amax = max(A), Amin = min(A), Asd = sd(A))]
Asummary
```

```
   Amax Amin      Asd
1:    5    1 1.581139
```

0.117.1 `address`: Object location in memory

When you add a new column to an existing data.frame, the data.frame is copied behind the scenes - you can tell becasue its memory address (where it's physically stored in your computer) changes:

```
df1 <- data.frame(alpha = 1:5, beta = 11:15)
address(df1)
```

```
[1] "0×7fa3652795c8"
```

```
df1$gamma <- df1$alpha + df1$beta
address(df1)
```

```
[1] "0×7fa3657152e8"
```

When you add a new column in a data.table ***in-place*** its address remains unchanged:

```
dt1 <- data.table(alpha = 1:5, beta = 11:15)
address(dt1)
```

```
[1] "0×7fa365c70e00"
```

```
dt1[, gamma := alpha + beta]
address(dt1)
```

```
[1] "0×7fa365c70e00"
```

0.117.2 Reference semantics at work

Up to now, you are likely used to working with regular R objects that behave like this:

```
(df1 <- data.frame(a = rep(1, 5)))
```

```
  a
1 1
2 1
3 1
4 1
5 1
```

```
(df2 <- df1)
```

```
  a
1 1
2 1
3 1
4 1
5 1
```

```
df2$a <- df2$a*2
df2
```

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

```
4 2
5 2
```

```
df1
```

```
  a
1 1
2 1
3 1
4 1
5 1
```

```
address(df1)
```

```
[1] "0×7fa3669faa68"
```

```
address(df2)
```

```
[1] "0×7fa366a34448"
```

`data.table` uses "reference semantics" or "pass-by-reference". Be very careful or you might be mightily confused:

```
(dt1 <- data.table(a = rep(1, 5)))
```

```
   a
1: 1
2: 1
3: 1
4: 1
5: 1
```

```
(dt2 <- dt1)
```

```
   a
1: 1
2: 1
3: 1
4: 1
5: 1
```

```
dt2[, a := a * 2]
dt2
```

```
   a
1: 2
2: 2
3: 2
4: 2
5: 2
```

```
dt1
```

```
   a
1: 2
2: 2
3: 2
4: 2
5: 2
```

```
address(dt1)
```

```
[1] "0×7fa364294e00"
```

```
address(dt2)
```

```
[1] "0×7fa364294e00"
```

If you want to create a copy of a data.table, use `copy()`:

```
(dt3 <- copy(dt1))
```

```
   a
1: 2
2: 2
3: 2
4: 2
5: 2
```

```
address(dt3)
```

```
[1] "0×7fa3616f6400"
```

```
dt3[, a := a * 2]
dt3
```

```
   a
1: 4
2: 4
3: 4
4: 4
5: 4
```

```
dt1
```

```
   a
1: 2
2: 2
3: 2
4: 2
5: 2
```

0.118 `set*()`: Set attributes *in-place*

`data.table` includes a number of function that begin with `set*`, all of which change their input *by reference* and as such require no assignment.
You may be surprised to find out that even an inocuous operation like changing the column names of a data.frame, requires a copy:

```
address(df)
```

```
[1] "0×7fa3617992e8"
```

```
colnames(df) <- c("A", "B", "Group")
address(df)
```

```
[1] "0×7fa3652707a8"
```

Use `setnames()` to edit a `data.table`'s column names ***in-place***:

```
address(dt)
```

```
[1] "0×7fa363242a00"
```

```
setnames(dt, old = 1:3, new = c("A", "B", "Group"))
address(dt)
```

```
[1] "0×7fa363242a00"
```

0.119 `setorder()`: Set order of `data.table`

Since this is a `set*` function, it changes a `data.table` in-place. You can order by any number of columns, ascending or descending:
Order by Group and then by A:

```
setorder(dt, Group, A)
dt
```

```
   A   B Group
1: 1 1.2     a
2: 4 5.6     a
3: 5 8.1     a
4: 2 4.3     b
5: 3 9.7     b
```

Order by Group and then by decreasing B:

```
setorder(dt, Group, -B)
dt
```

```
   A   B Group
1: 5 8.1     a
2: 4 5.6     a
3: 1 1.2     a
4: 3 9.7     b
5: 2 4.3     b
```

0.120 Group according to `by`

Up to now, we have learned how to use the `data.table` frame `dat[i, j]` to filter cases in `i` or add/remove/transform columns in-place in `j`. There is a whole other dimension in the `data.table` frame: `by`.

The complete `data.table` syntax is:

`dt[i, j, by]`

- Take data.table `dt`
- Subset rows using `i`
- Manipulate columns with `j`
- Grouped according to `by`

Again, using `.()` or `list()` in `j`, returns a new `data.table`:

```
dt[, .(meanAbyGroup = mean(A)), by = Group]
```

```
   Group meanAbyGroup
1:     a     3.333333
2:     b     2.500000
```

```
dt[, list(medianBbyGroup = median(B)), by = Group]
```

```
   Group medianBbyGroup
1:     a            5.6
2:     b            7.0
```

Making an assignment with `:=` in `j`, adds a column in-place. Since here we are grouping, the same value will be assigned to all cases of the group:

```
dt[, meanAbyGroup := mean(A), by = Group]
dt
```

```
   A   B Group meanAbyGroup
1: 5 8.1     a     3.333333
2: 4 5.6     a     3.333333
3: 1 1.2     a     3.333333
4: 3 9.7     b     2.500000
5: 2 4.3     b     2.500000
```

For more complex operations, you may need to refer to the slice of the `data.table` defined by `by` within `j`. There is a special notation for this: `.SD` (think sub-`data.table`):

```
dt[, A_DiffFromGroupMin := .SD[, 1] - min(.SD[, 1]), by = Group]
dt
```

```
   A   B Group meanAbyGroup A_DiffFromGroupMin
```

```
1: 5 8.1     a     3.333333                   4
2: 4 5.6     a     3.333333                   3
3: 1 1.2     a     3.333333                   0
4: 3 9.7     b     2.500000                   1
5: 2 4.3     b     2.500000                   0
```

By now, it should be clearer that the `data.table` frame provides a very flexible way to perform a very wide range of operations with minimal new notation.

0.121 Apply functions to columns

Any function that returns a list can be used in `j` to return a new data.table - therefore lapply is perfect for getting summary on multiple columns:

```
(dt1 <- as.data.table(rnormmat(10, 3, seed = 2020)))
```

```
            V1          V2          V3
 1:  0.3769721 -0.85312282  2.17436525
 2:  0.3015484  0.90925918  1.09818265
 3: -1.0980232  1.19637296  0.31822032
 4: -1.1304059 -0.37158390 -0.07314756
 5: -2.7965343 -0.12326023  0.83426874
 6:  0.7205735  1.80004312  0.19875064
 7:  0.9391210  1.70399588  1.29784138
 8: -0.2293777 -3.03876461  0.93671831
 9:  1.7591313 -2.28897495 -0.14743319
10:  0.1173668  0.05830349  0.11043199
```

```
setnames(dt1, c("Alpha", "Beta", "Gamma"))
dt1[, lapply(.SD, mean)]
```

```
        Alpha       Beta     Gamma
1: -0.1039628 -0.1007732 0.6748199
```

You can specify which columns to operate on by adding the `.SDcols` argument:

```
dt2 <- data.table(A = 1:5,
                  B = c(1.2, 4.3, 9.7, 5.6, 8.1),
                  C = rnorm(5),
                  Group = c("a", "b", "b", "a", "a"))
dt2
```

```
   A   B          C Group
1: 1 1.2 -0.8125047     a
2: 2 4.3 -0.7437022     b
3: 3 9.7  1.0953451     b
4: 4 5.6  2.4353737     a
5: 5 8.1  0.3881185     a
```

```
dt2[, lapply(.SD, mean), .SDcols = 1:2]
```

```
   A    B
1: 3 5.78
```

```
# same as
dt2[, lapply(.SD, mean), .SDcols = c("A", "B")]
```

```
   A    B
1: 3 5.78
```

```
cols <- c("A", "B")
dt2[, lapply(.SD, mean), .SDcols = cols]
```

```
   A    B
1: 3 5.78
```

You can combine `.SDcols` and `by`:

```
dt2[, lapply(.SD, median), .SDcols = c("B", "C"), by = Group]
```

```
   Group   B         C
1:     a 5.6 0.3881185
2:     b 7.0 0.1758215
```

Create multiple new columns from transformation of existing and store with custom prefix:

```
dt1
```

```
         Alpha        Beta       Gamma
 1:  0.3769721 -0.85312282  2.17436525
 2:  0.3015484  0.90925918  1.09818265
 3: -1.0980232  1.19637296  0.31822032
 4: -1.1304059 -0.37158390 -0.07314756
 5: -2.7965343 -0.12326023  0.83426874
```

```
 6:  0.7205735  1.80004312  0.19875064
 7:  0.9391210  1.70399588  1.29784138
 8: -0.2293777 -3.03876461  0.93671831
 9:  1.7591313 -2.28897495 -0.14743319
10:  0.1173668  0.05830349  0.11043199
```

```
dt1[, paste0(names(dt1), "_abs") := lapply(.SD, abs)]
dt1
```

```
      Alpha      Beta     Gamma Alpha_abs  Beta_abs  Gamma_abs
 1:  0.3769721 -0.85312282  2.17436525 0.3769721 0.85312282 2.17436525
 2:  0.3015484  0.90925918  1.09818265 0.3015484 0.90925918 1.09818265
 3: -1.0980232  1.19637296  0.31822032 1.0980232 1.19637296 0.31822032
 4: -1.1304059 -0.37158390 -0.07314756 1.1304059 0.37158390 0.07314756
 5: -2.7965343 -0.12326023  0.83426874 2.7965343 0.12326023 0.83426874
 6:  0.7205735  1.80004312  0.19875064 0.7205735 1.80004312 0.19875064
 7:  0.9391210  1.70399588  1.29784138 0.9391210 1.70399588 1.29784138
 8: -0.2293777 -3.03876461  0.93671831 0.2293777 3.03876461 0.93671831
 9:  1.7591313 -2.28897495 -0.14743319 1.7591313 2.28897495 0.14743319
10:  0.1173668  0.05830349  0.11043199 0.1173668 0.05830349 0.11043199
```

```
dt2
```

```
   A   B          C Group
1: 1 1.2 -0.8125047     a
2: 2 4.3 -0.7437022     b
3: 3 9.7  1.0953451     b
4: 4 5.6  2.4353737     a
5: 5 8.1  0.3881185     a
```

```
cols <- c("A", "C")
dt2[, paste0(cols, "_groupMean") := lapply(.SD, mean), .SDcols = cols, b
dt2
```

```
   A   B          C Group A_groupMean C_groupMean
1: 1 1.2 -0.8125047     a    3.333333   0.6703292
2: 2 4.3 -0.7437022     b    2.500000   0.1758215
3: 3 9.7  1.0953451     b    2.500000   0.1758215
4: 4 5.6  2.4353737     a    3.333333   0.6703292
5: 5 8.1  0.3881185     a    3.333333   0.6703292
```

0.122 Reshape a `data.table`

0.122.1 `melt()`: Wide to long

```
dt_wide <- data.table(ID = 1:4, Timepoint_A = 11:14,
                      Timepoint_B = 21:24, Timepoint_C = 51:54)
dt_wide
```

```
   ID Timepoint_A Timepoint_B Timepoint_C
1:  1          11          21          51
2:  2          12          22          52
3:  3          13          23          53
4:  4          14          24          54
```

```
dt_long <- melt(dt_wide, id.vars = "ID",
                measure.vars = 2:4, # defaults to all non-id columns
                variable.name = "Timepoint",
                value.name = c("Score"))
dt_long
```

```
    ID   Timepoint Score
 1:  1 Timepoint_A    11
 2:  2 Timepoint_A    12
 3:  3 Timepoint_A    13
 4:  4 Timepoint_A    14
 5:  1 Timepoint_B    21
 6:  2 Timepoint_B    22
 7:  3 Timepoint_B    23
 8:  4 Timepoint_B    24
 9:  1 Timepoint_C    51
10:  2 Timepoint_C    52
11:  3 Timepoint_C    53
12:  4 Timepoint_C    54
```

0.122.2 `dcast()`: Long to wide

```
dt_long
```

```
    ID   Timepoint Score
 1:  1 Timepoint_A    11
```

```
 2:  2 Timepoint_A    12
 3:  3 Timepoint_A    13
 4:  4 Timepoint_A    14
 5:  1 Timepoint_B    21
 6:  2 Timepoint_B    22
 7:  3 Timepoint_B    23
 8:  4 Timepoint_B    24
 9:  1 Timepoint_C    51
10:  2 Timepoint_C    52
11:  3 Timepoint_C    53
12:  4 Timepoint_C    54
```

```
dcast(dt_long, ID ~ Timepoint,
      value.var = "Score")
```

```
   ID Timepoint_A Timepoint_B Timepoint_C
1:  1          11          21          51
2:  2          12          22          52
3:  3          13          23          53
4:  4          14          24          54
```

0.123 Table Joins

`data.table` allow you to perform table joins either with the base R `merge()` or with its own bracket notation:

```
(a <- data.table(PID = c(1:9),
                 Hospital = c("UCSF", "HUP", "Stanford",
                              "Stanford", "UCSF", "HUP",
                              "HUP", "Stanford", "UCSF"),
                 Age = c(22, 34, 41, 19, 53, 21, 63, 22, 19),
                 Sex = c(1, 1, 0, 1, 0, 0, 1, 0, 0)))
```

```
   PID Hospital Age Sex
1:   1     UCSF  22   1
2:   2      HUP  34   1
3:   3 Stanford  41   0
4:   4 Stanford  19   1
5:   5     UCSF  53   0
6:   6      HUP  21   0
7:   7      HUP  63   1
8:   8 Stanford  22   0
9:   9     UCSF  19   0
```

```
(b  <- data.table(PID = c(6:12),
                  V1 = c(153, 89, 112, 228,  91, 190, 101),
                  Department = c("Neurology", "Radiology", "Emergency",
                                 "Cardiology", "Surgery", "Neurology",
                                 "Psychiatry")))
```

```
   PID  V1 Department
1:   6 153  Neurology
2:   7  89  Radiology
3:   8 112  Emergency
4:   9 228 Cardiology
5:  10  91    Surgery
6:  11 190  Neurology
7:  12 101 Psychiatry
```

0.123.1 Inner

```
merge(a, b)
```

```
   PID Hospital Age Sex  V1 Department
1:   6      HUP  21   0 153  Neurology
2:   7      HUP  63   1  89  Radiology
3:   8 Stanford  22   0 112  Emergency
4:   9     UCSF  19   0 228 Cardiology
```

0.123.2 Outer

```
merge(a, b, all = TRUE)
```

```
    PID Hospital Age Sex  V1 Department
 1:   1     UCSF  22   1  NA       <NA>
 2:   2      HUP  34   1  NA       <NA>
 3:   3 Stanford  41   0  NA       <NA>
 4:   4 Stanford  19   1  NA       <NA>
 5:   5     UCSF  53   0  NA       <NA>
 6:   6      HUP  21   0 153  Neurology
 7:   7      HUP  63   1  89  Radiology
 8:   8 Stanford  22   0 112  Emergency
 9:   9     UCSF  19   0 228 Cardiology
```

```
10:  10     <NA>  NA  NA  91    Surgery
11:  11     <NA>  NA  NA 190  Neurology
12:  12     <NA>  NA  NA 101 Psychiatry
```

0.123.3 Left outer

```
merge(a, b, all.x = TRUE)
```

```
   PID Hospital Age Sex  V1 Department
1:   1     UCSF  22   1  NA       <NA>
2:   2      HUP  34   1  NA       <NA>
3:   3 Stanford  41   0  NA       <NA>
4:   4 Stanford  19   1  NA       <NA>
5:   5     UCSF  53   0  NA       <NA>
6:   6      HUP  21   0 153  Neurology
7:   7      HUP  63   1  89  Radiology
8:   8 Stanford  22   0 112  Emergency
9:   9     UCSF  19   0 228 Cardiology
```

One way to allow fast joins with bracket notation is to set keys:

```
setkey(a, "PID")
setkey(b, "PID")
```

```
b[a, ]
```

```
   PID  V1 Department Hospital Age Sex
1:   1  NA       <NA>     UCSF  22   1
2:   2  NA       <NA>      HUP  34   1
3:   3  NA       <NA> Stanford  41   0
4:   4  NA       <NA> Stanford  19   1
5:   5  NA       <NA>     UCSF  53   0
6:   6 153  Neurology      HUP  21   0
7:   7  89  Radiology      HUP  63   1
8:   8 112  Emergency Stanford  22   0
9:   9 228 Cardiology     UCSF  19   0
```

0.123.4 Right outer

```
merge(a, b, all.y = TRUE)
```

```
   PID Hospital Age Sex  V1 Department
1:   6      HUP  21   0 153  Neurology
2:   7      HUP  63   1  89  Radiology
3:   8 Stanford  22   0 112  Emergency
4:   9     UCSF  19   0 228 Cardiology
5:  10     <NA>  NA  NA  91    Surgery
6:  11     <NA>  NA  NA 190  Neurology
7:  12     <NA>  NA  NA 101 Psychiatry
```

```
a[b, ]
```

```
   PID Hospital Age Sex  V1 Department
1:   6      HUP  21   0 153  Neurology
2:   7      HUP  63   1  89  Radiology
3:   8 Stanford  22   0 112  Emergency
4:   9     UCSF  19   0 228 Cardiology
5:  10     <NA>  NA  NA  91    Surgery
6:  11     <NA>  NA  NA 190  Neurology
7:  12     <NA>  NA  NA 101 Psychiatry
```

Base Graphics

R has powerful graphical capabilities built in to the core language. This chapter is an introduction to what is known as base graphics which is provided by the **graphics** builtin package. Their defaults produce minimalist plots, but they can be customized extensively. In this chapter we shall begin with default plots and demonstrate some of the more common/useful ways to customize them.

Plot type	Command
Scatterplot	`plot(x, y)`
Line plot	`plot(x, y, type = 'l')`
Histogram	`hist(x)`
Density plot	`plot(density(x))`
Barplot	`barplot(x)`
Boxplot	`boxplot(x)`
Heatmap	`heatmap(x)`

R documentation for each of the above commands provides extensive coverage of graphical parameters. `?par` gives the main documentation file for a long list of graphical parameters. These can be set either with the `par()` command ahed before using any plotting command

Let's create some synthetic data:

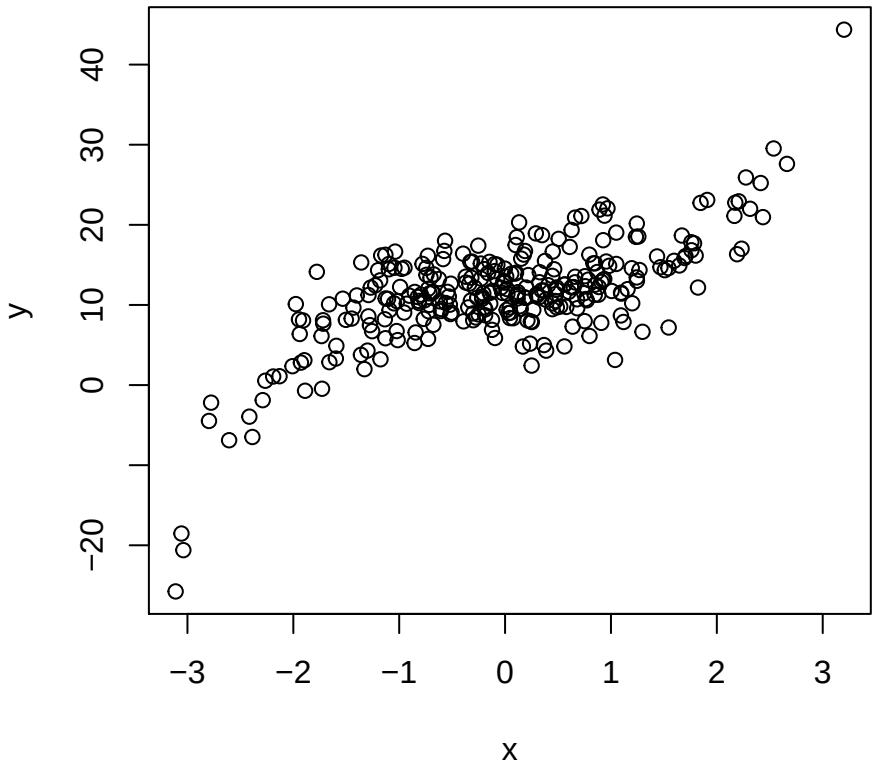
```
set.seed(2020)
x <- rnorm(300)
y_true <- 12 + x^3
y <- 12 + x^3 + 2.5 * rnorm(300)*1.5
```

0.124 Scatter plot

Input: 2 **numeric vectors**

A 2D scatterplot displays of two numeric vectors as X and Y coordinates.

```
plot(x, y)
```



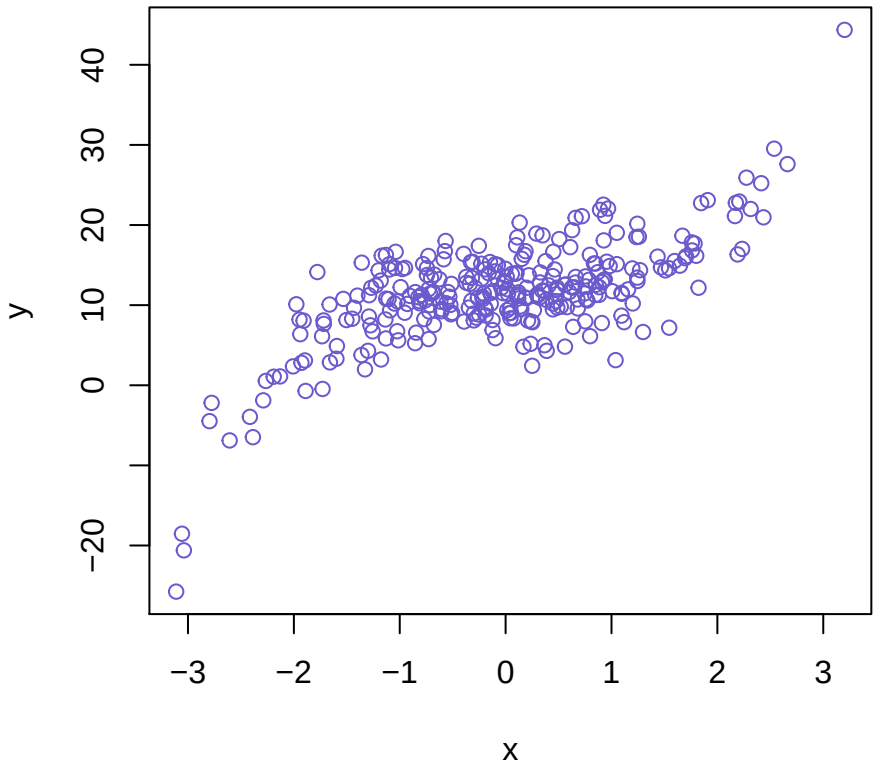
0.124.1 `col`: point color

See Colors in R to learn about the different ways to define colors in R.

Some common ways include:

- By name using one of 657 names given by `colors()`, e.g. “red”, “magenta”, “blue”, “navy”, “cyan”
- By RGB code

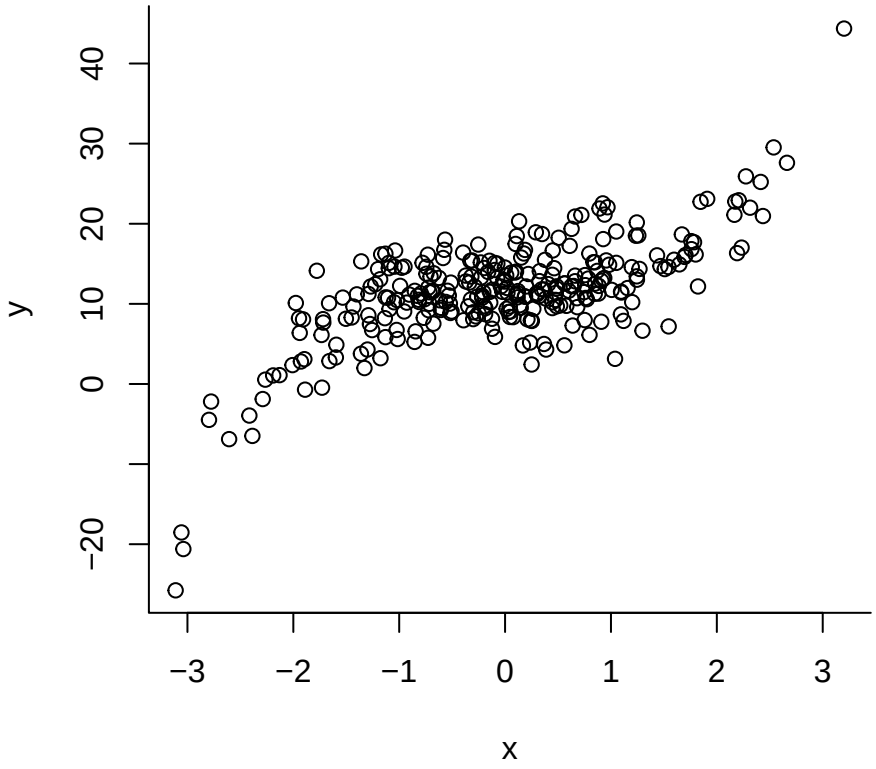
```
plot(x, y, col = "slateblue")
```



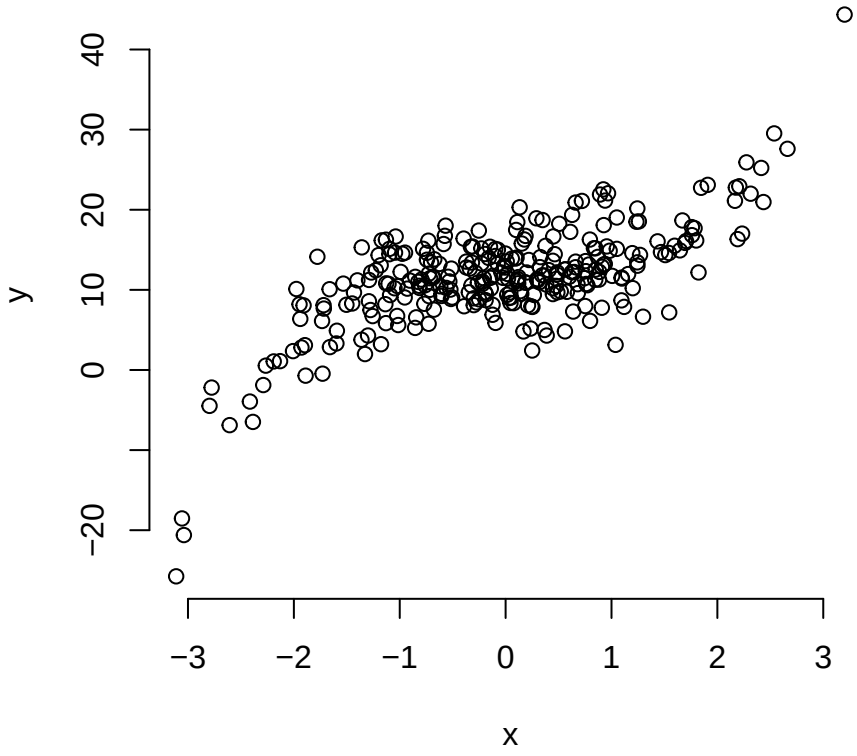
0.124.2 `bty`: box type

There are 7 bty options: “o” “l”, “7”, “c”, “u”, or “]” and “none”. They produce a box that resembles the corresponding symbol. “none” draws no box but allows the axes to show:

```
plot(x, y, bty = "l")
```




```
plot(x, y, bty = "none")
```



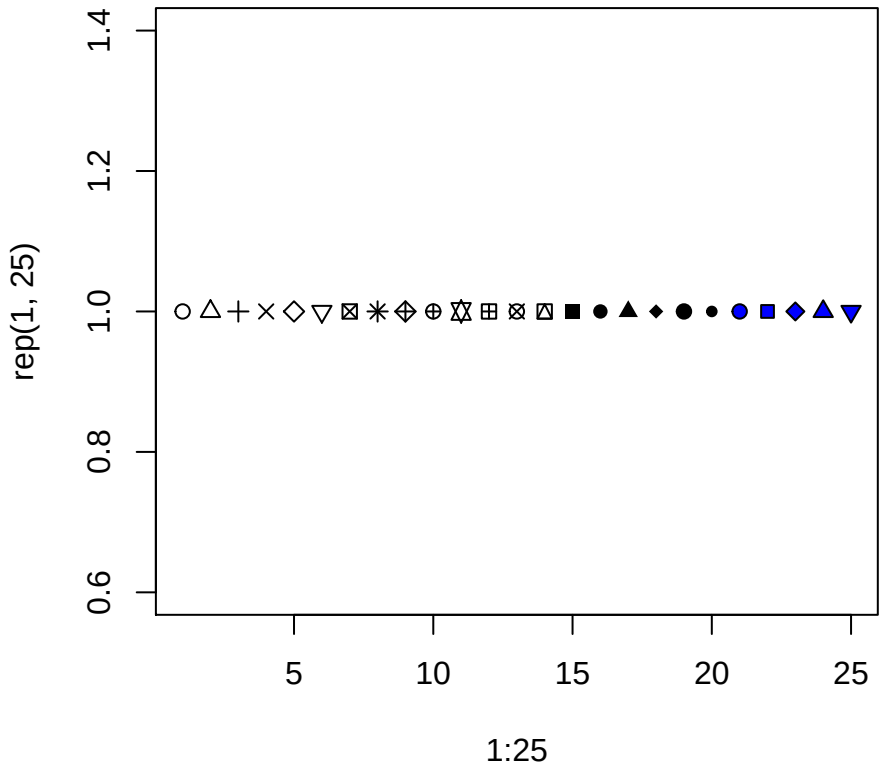
0.124.3 `pch`: point character

The default point character is a circle as seen above. This helps visualize overlapping points (especially for devices that do not support transparency).

There are 25 point characters, designated by integers 1 through 25.

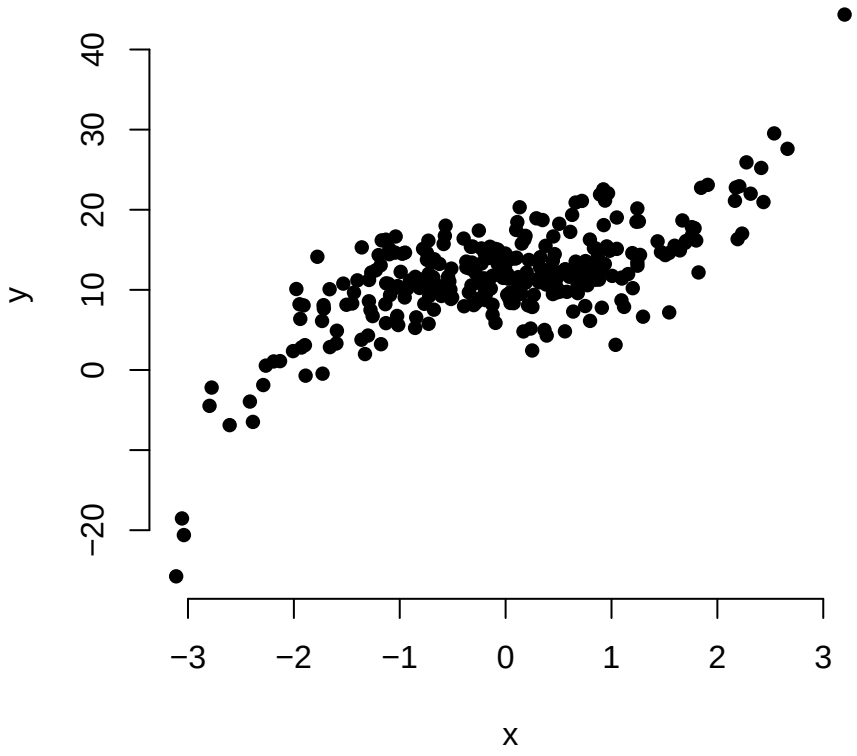
Here's a preview of all 25 `pch` options. `pch` types 21 through 25 can be filled by a color specified by `bg`.

```
plot(1:25, rep(1, 25), pch = 1:25, bg = "blue")
```



Let's use a solid disc:

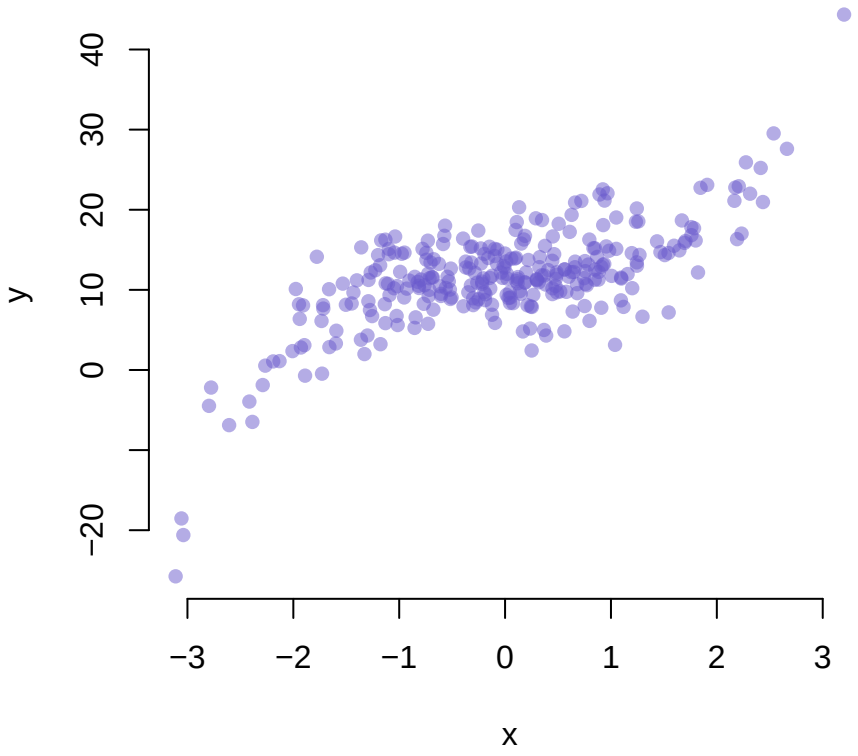
```
plot(x, y, bty = "n", pch = 16)
```



We cannot tell how many points are overlapping in the middle and therefore it's a good idea to make the points a little transparent.

There are different ways to add transparency (see Colors). The easiest way is probably to use `adjustcolor()`. In the context of colors, `alpha` refers to transparency: `a = 1` is opaque and `a = 0` is completely transparent (therefore use a value greater than 0).

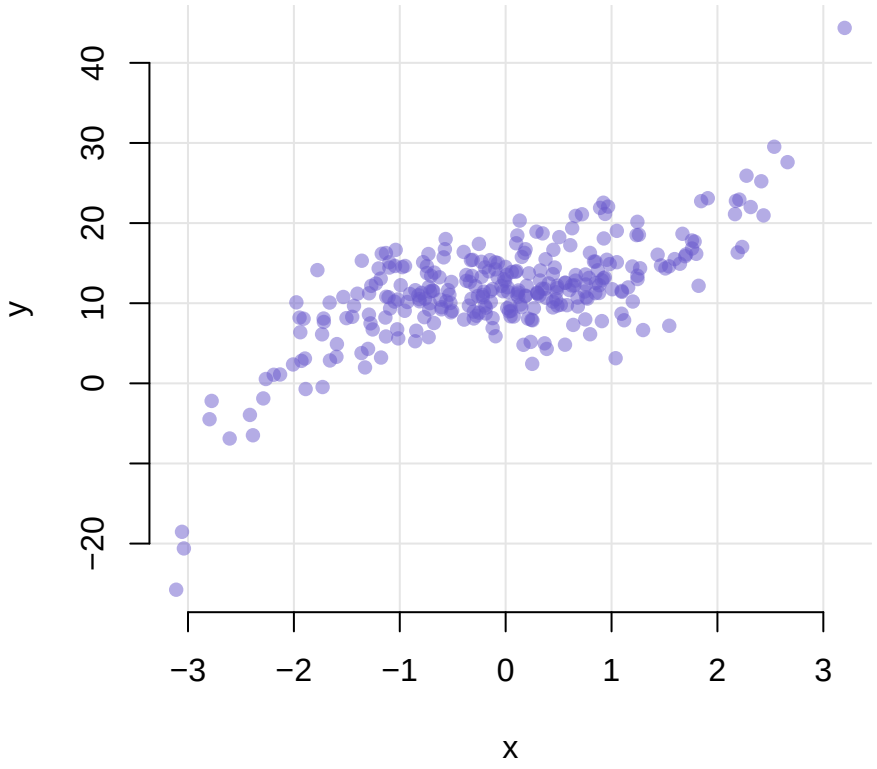
```
plot(x, y,
     bty = "n", pch = 16,
     col = adjustcolor("slateblue", alpha.f = .5))
```



0.124.4 grid

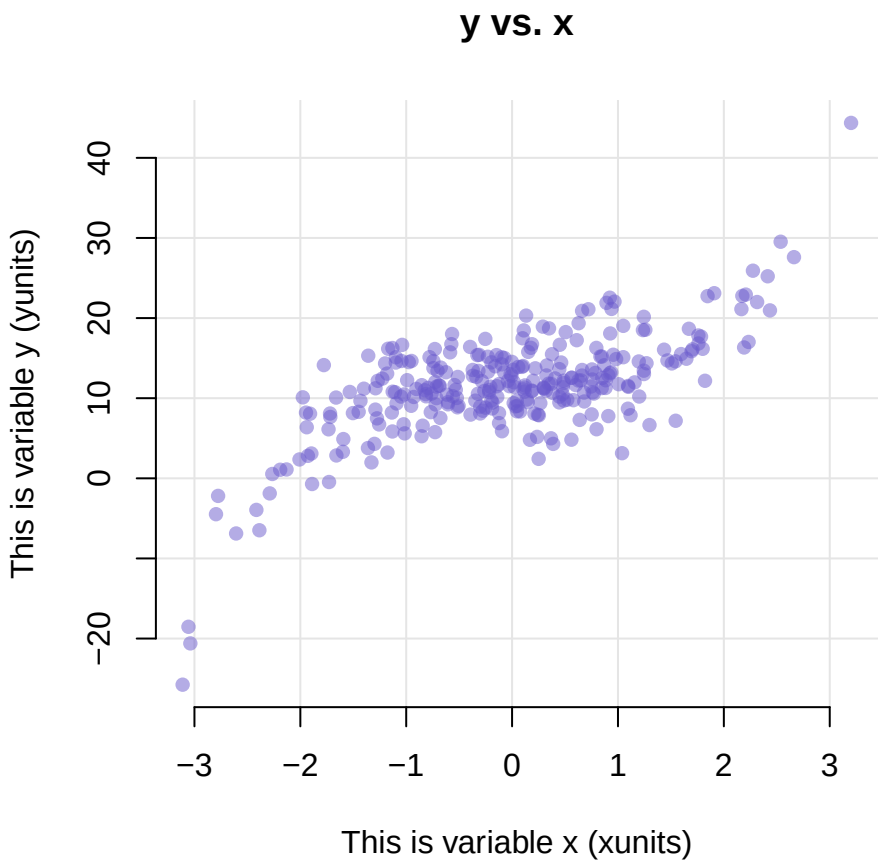
We can add a grid behind the plot area using the `panel.first`, which accepts a graphical expression (a function that draws something), which is evaluated before plotting the points on the graph (therefore appears behind the points as required).

```
plot(x, y,
     bty = "n", pch = 16,
     col = adjustcolor("slateblue", alpha.f = .5),
     panel.first = grid(lty = 1, col = 'gray90'))
```



0.124.5 `main`, `xlab`, `ylab`: Title and axes labels

```
plot(x, y,
     bty = "n", pch = 16,
     col = adjustcolor("slateblue", alpha.f = .5),
     panel.first = grid(lty = 1, col = 'gray90'),
     main = "y vs. x",
     xlab = "This is variable x (xunits)",
     ylab = "This is variable y (yunits)")
```



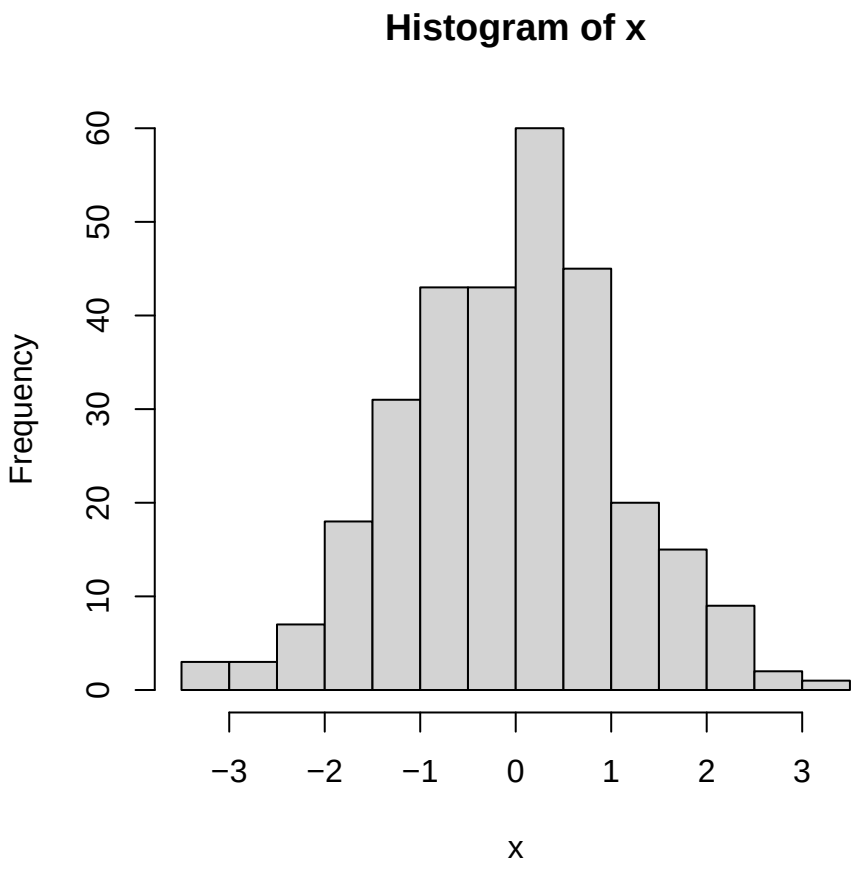
Note that depending on where you intend to display the plot, you may leave the title blank and instead place it in the figure caption along with an explanation of the data (e.g. in a journal article)

0.125 Histogram

Input: **numeric vector**

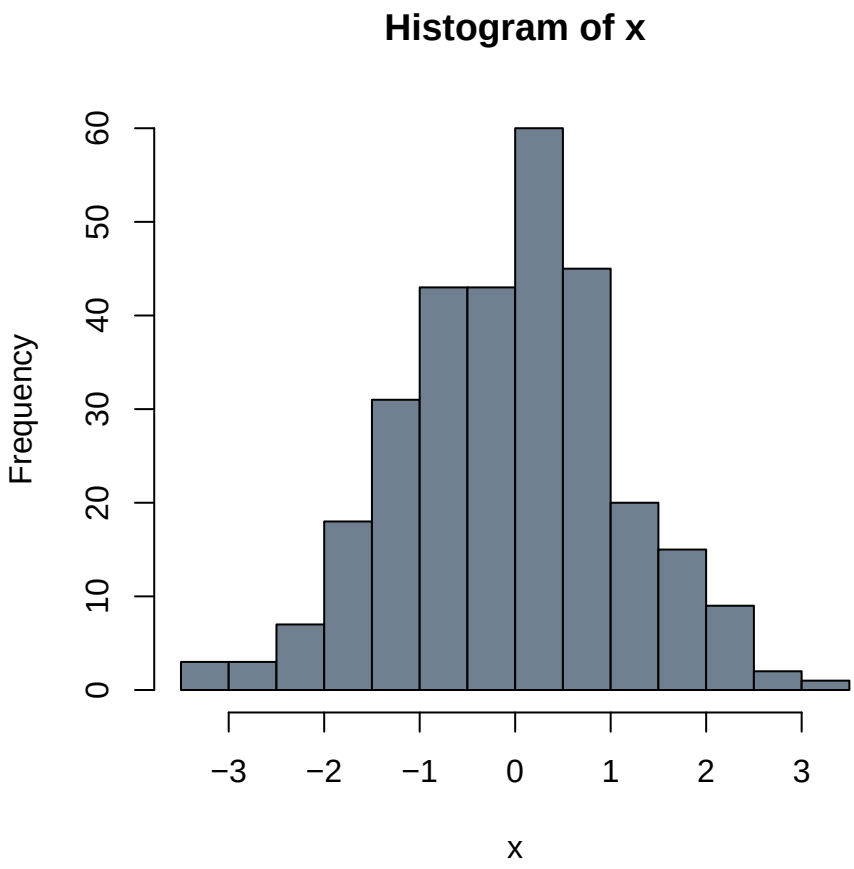
A histogram displays an approximation of the distribution of a numeric vector. First the data is binned and then the number of elements that falls in each bin is counted. The histogram plot draws bars ofr each bin whose heights corresponds to the count of elements in the corresponding interval.

```
hist(x)
```



0.125.1 `col`: bar color

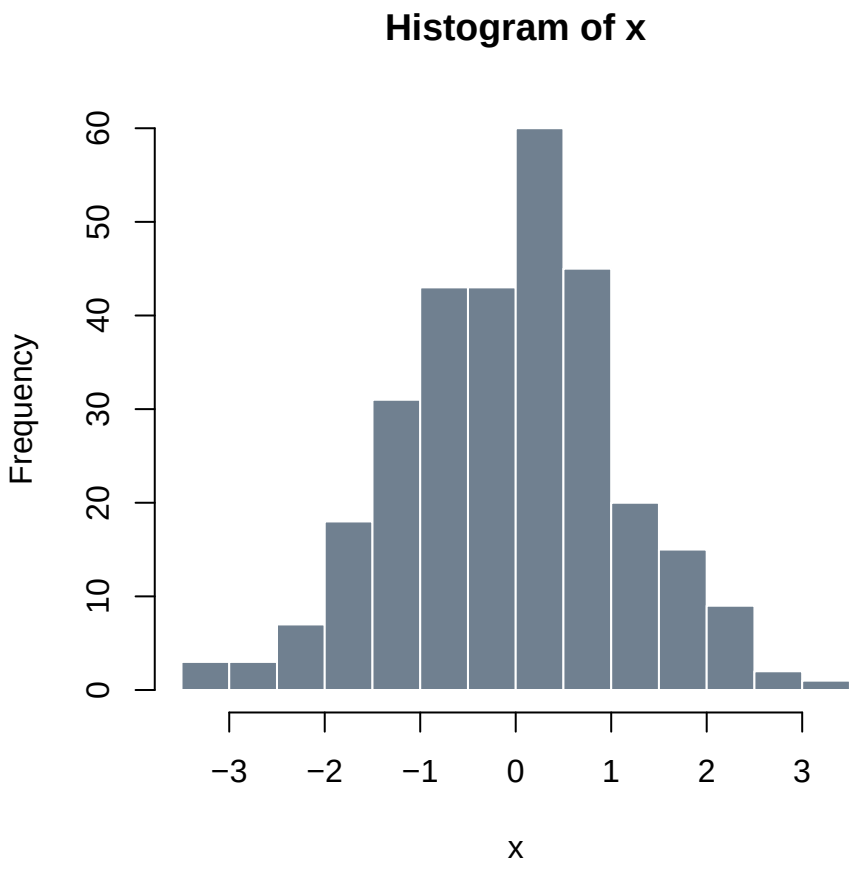
```
hist(x, col = "slategrey")
```



0.125.2 `border`: border color

Setting border color to the same as the background gives a clean look:

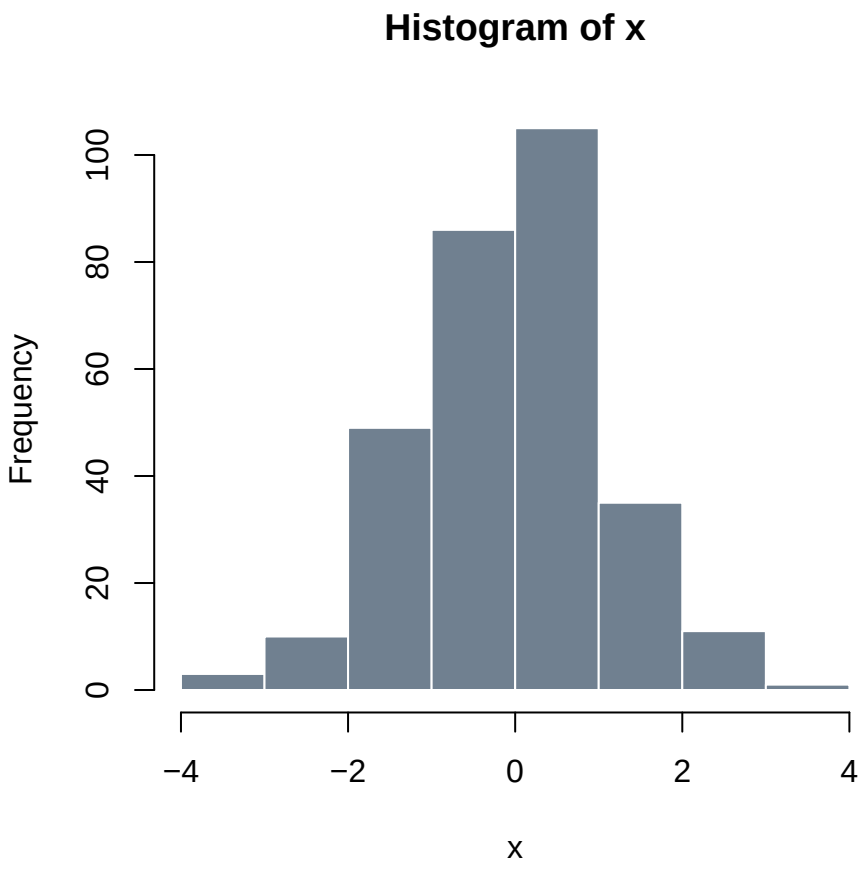
```
hist(x, col = "slategrey", border = "white")
```



0.125.3 `breaks`: number and/or value of breakpoints

The `breaks` argument can be used to define the breakpoints to use for the binning of the values of the input to `hist()`. See the documentation in `?hist` for the full range of options. An easy way to control the number of bins is to pass an integer to the `breaks` argument. Depending on the length of x and its distribution, it may or may not be possible to use the exact number requested, but the closest possible number will be automatically chosen.

```
hist(x, col = "slategrey", border = "white",
     breaks = 8)
```

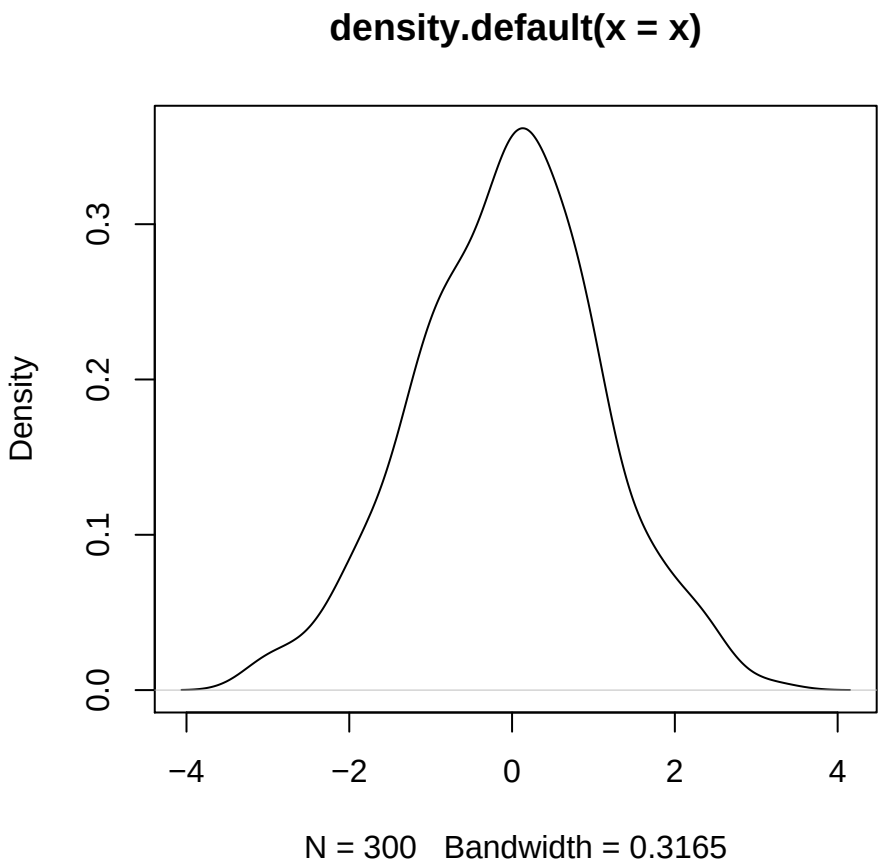


0.126 Density plot

Input: numeric vector

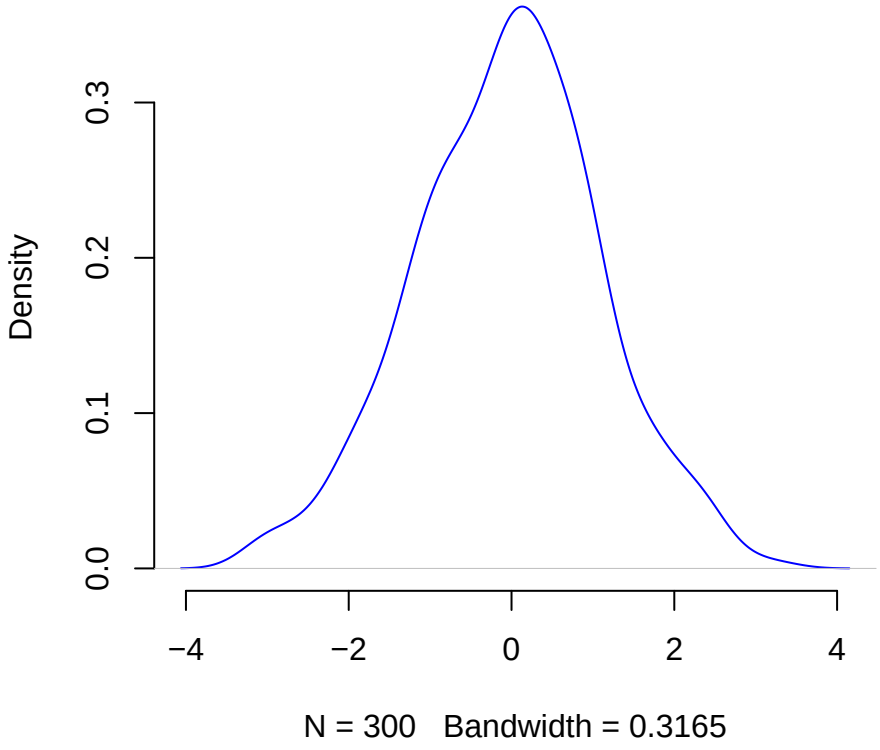
A density plot is a different way to display an approximation of the distribution of a numeric vector. The `density()` function estimates the density of x and can be passed to `plot()` directly:

```
plot(density(x))
```



You can use `main = NA` or `main = ""` to suppress printing a title.

```
plot(density(x), col = "blue",
     bty = "n",
     main = NA)
```



0.127 Barplot

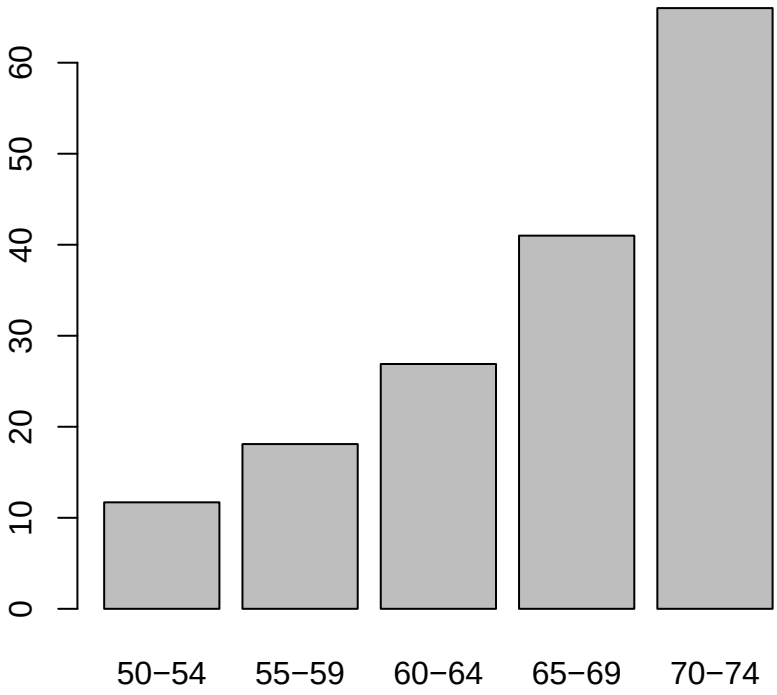
Input: **vector** or **matrix**

Let's look at the VADeaths built-in dataset which describes death rater per 1000 population per year broken down by age range and population group.

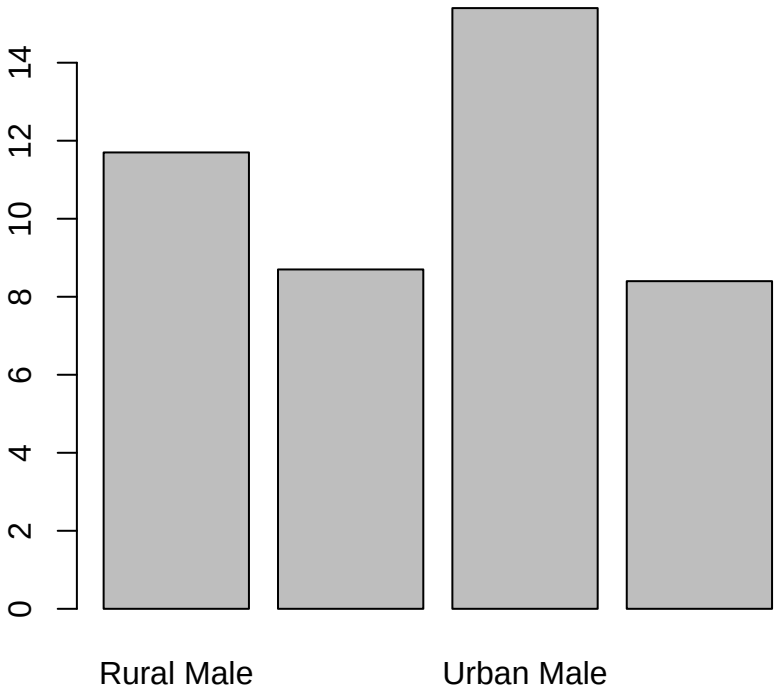
0.127.1 Single vector

We can plot a single column or row. Note how R automatically gets the corresponding dimension names:

```
barplot(VADeaths[, 1])
```



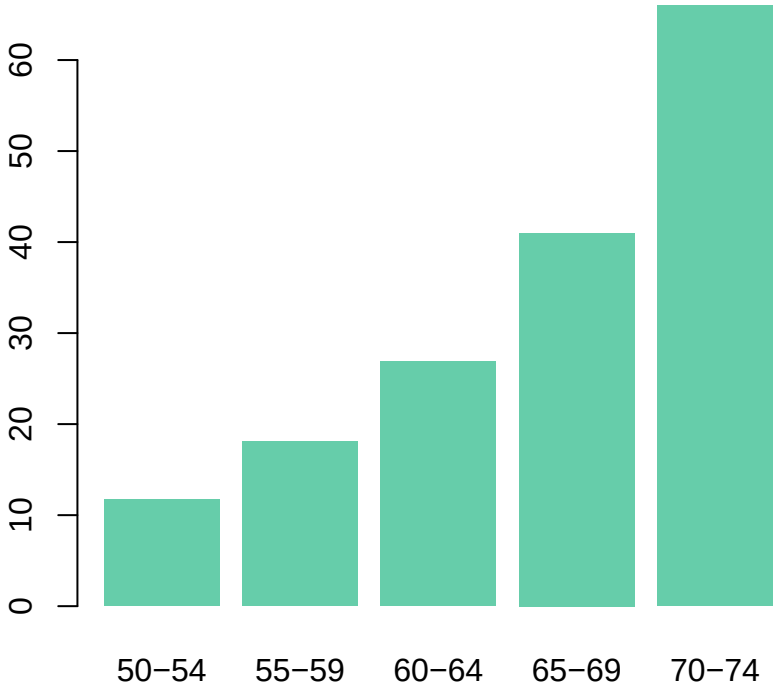

```
barplot(VADeaths[1, ])
```



0.127.1.1 `col` and `border`: bar fill and border color

As in most plotting functions, color is controlled by the **`col`** argument. **`border`** can be set to any color separately, or to `NA` to omit, which gives a clean look:

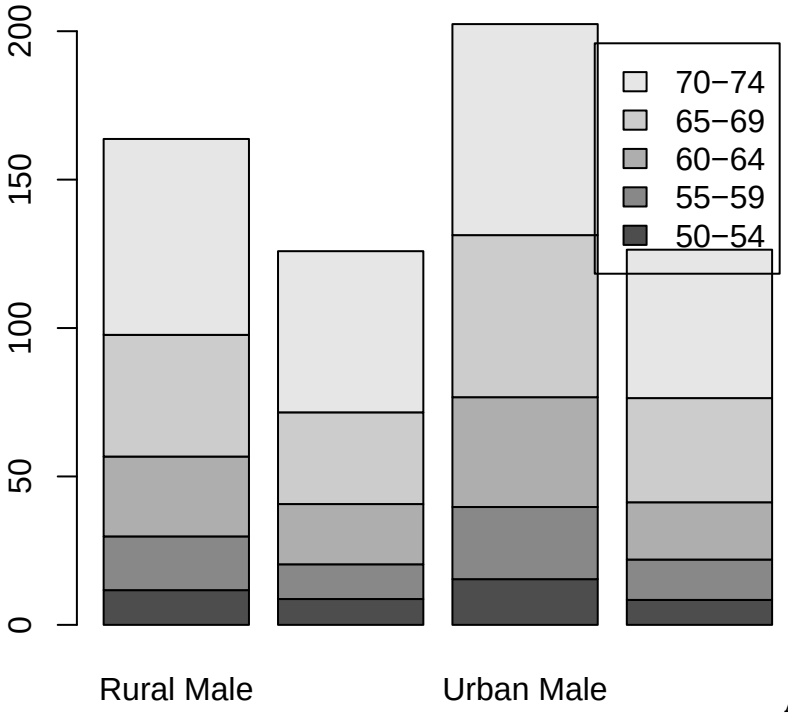
```
barplot(VADeaths[, 1],
        col = "aquamarine3", border = NA)
```



0.127.2 Matrix

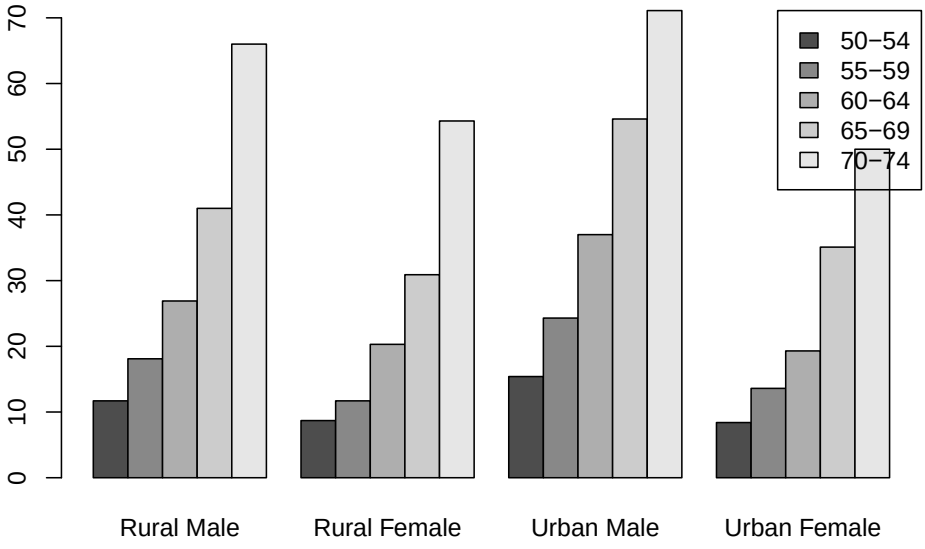
We can draw barplots of multiple columns at the same time by passing a matrix input. The grouping on the x-axis is based on the columns. By default, data from different rows is stacked. The argument `legend.text` can be used to add a legend with the row labels:

```
barplot(VADeaths, legend.text = TRUE)
```



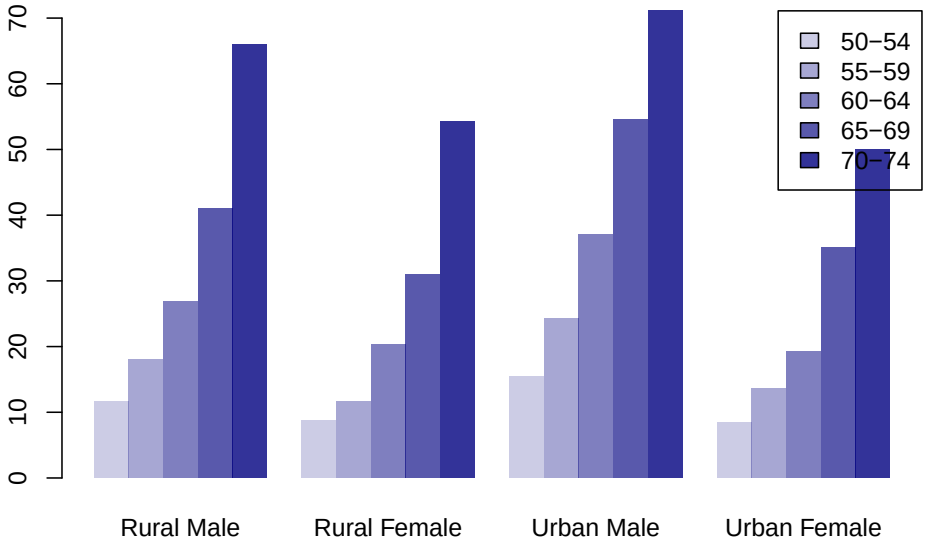
Alternatively, we can draw groups of bars beside each other with the argument `beside = TRUE`:

```
barplot(VADeaths, beside = TRUE,
        legend.text = TRUE, args.legend = list(x = "topright"))
```



To use custom colors, we pass a vector of length equal to the number of bars within each group. These will get recycled across groups, giving a consistent color coding. Here, we use the `adjustcolor()` function again to produce 5 shades of navy.

```
col <- sapply(seq(.2, .8, length.out = 5), function(i) adjustcolor("navy", i))
barplot(VADeaths,
        col = col,
        border = NA,
        beside = TRUE,
        legend.text = TRUE, args.legend = list(x = "topright"))
```



0.128 Boxplot

Input: One or more **vectors** of any length

A boxplot is another way to visualize the distribution of one or more vectors. Each vector does not need to be of the same length. For example if you are plotting lab results of a patient and control group, they do not have to contain the same number of individuals.

There are two ways to use the `boxplot()` function. Either pass two separate vectors of data (whet)

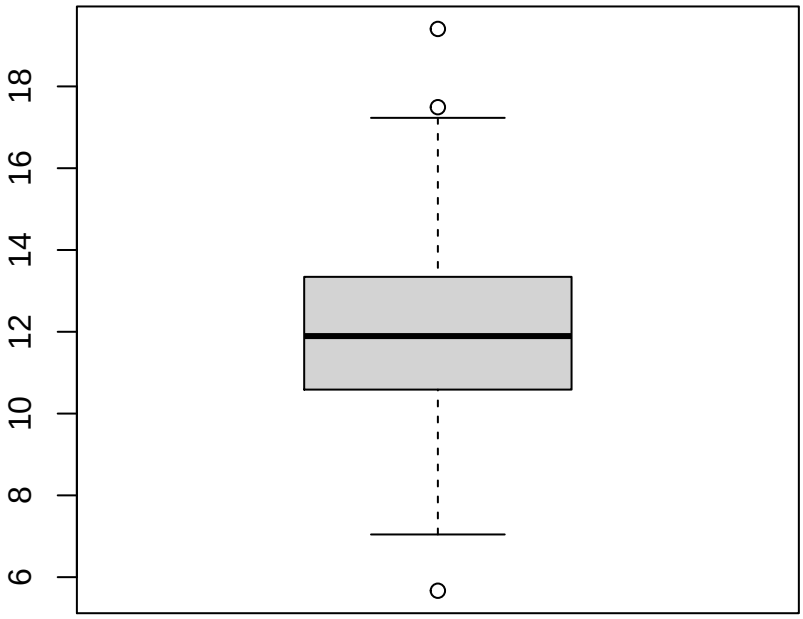
`boxplot()` makes it easy to plot your data from different objects. It can accept:

- individual vectors
- columns of a matrix, columns/elements of a data.frame, elements of a list

- formula interface of the form `variable ~ factor`

0.128.1 Single vector

```
a <- rnorm(500, mean = 12, sd = 2)
boxplot(a)
```



0.128.2 Anatomy of a boxplot

A boxplot shows:

- the median
- first and third quartiles
- outliers (defines as `x < Q1 - 1.5 * IQR | x > Q3 + 1.5 * IQR`)
- range after excluding outliers

Some synthetic data:

```
alpha <- rnorm(10)
beta <- rnorm(100)
gamma <- rnorm(200, 1, 2)
dl <- list(alpha = alpha, beta = beta, gamma = gamma)
```

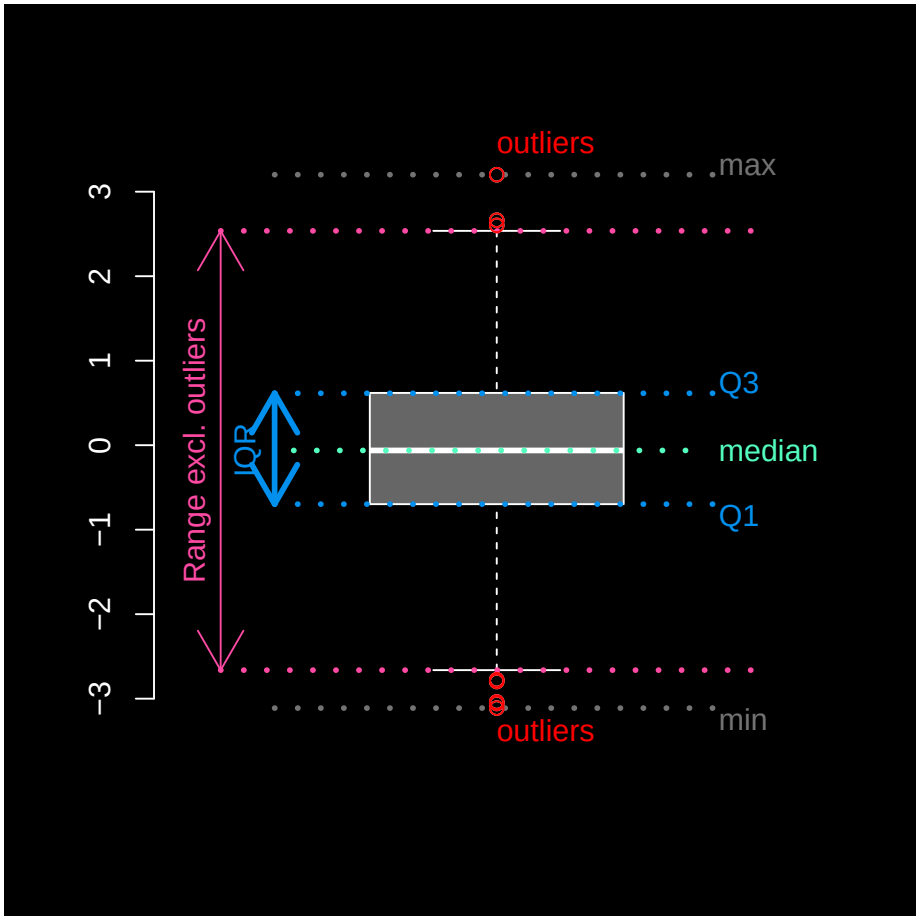
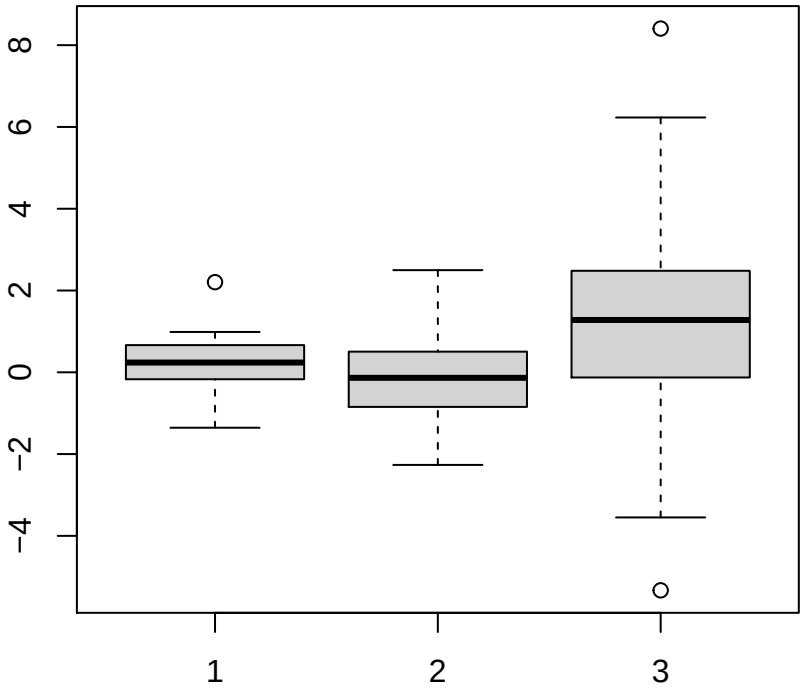


Figure 9: Boxplot abatomy

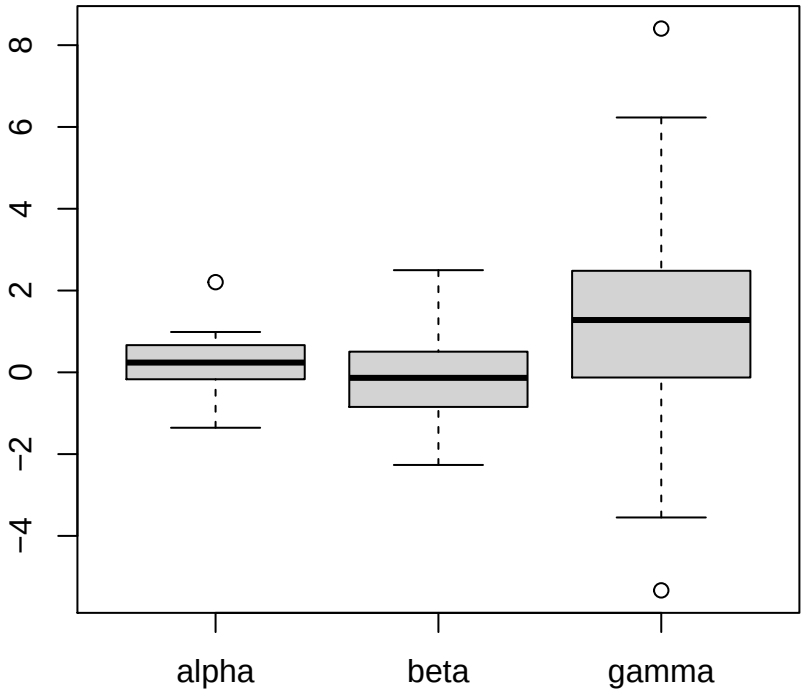
0.128.3 Multiple vectors

```
boxplot(alpha, beta, gamma)
```



0.128.4 List

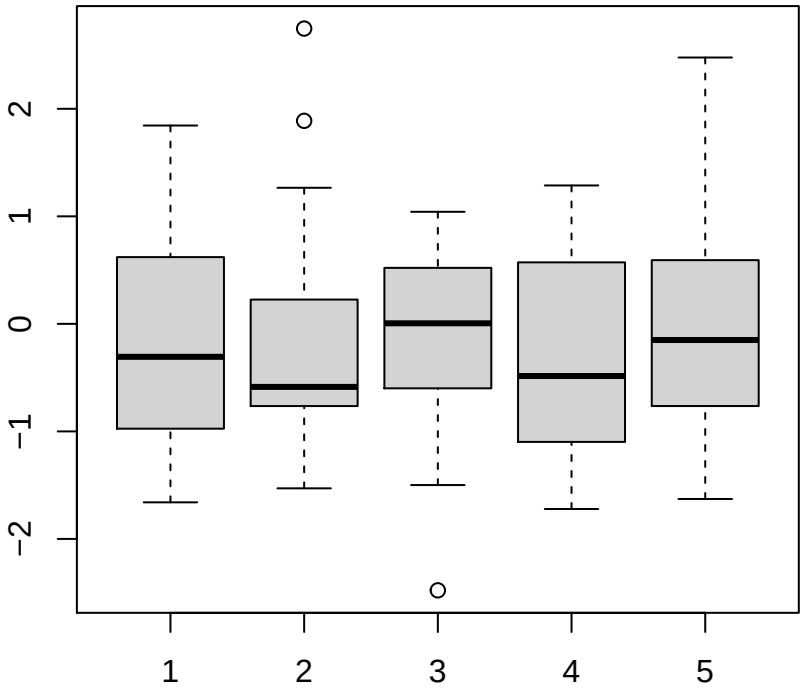
```
boxplot(dl)
```



0.128.5 Matrix

Passing a matrix to `boxplot()` draws one boxplot per column:

```
mat <- sapply(seq(5), function(i) rnorm(20))
boxplot(mat)
```

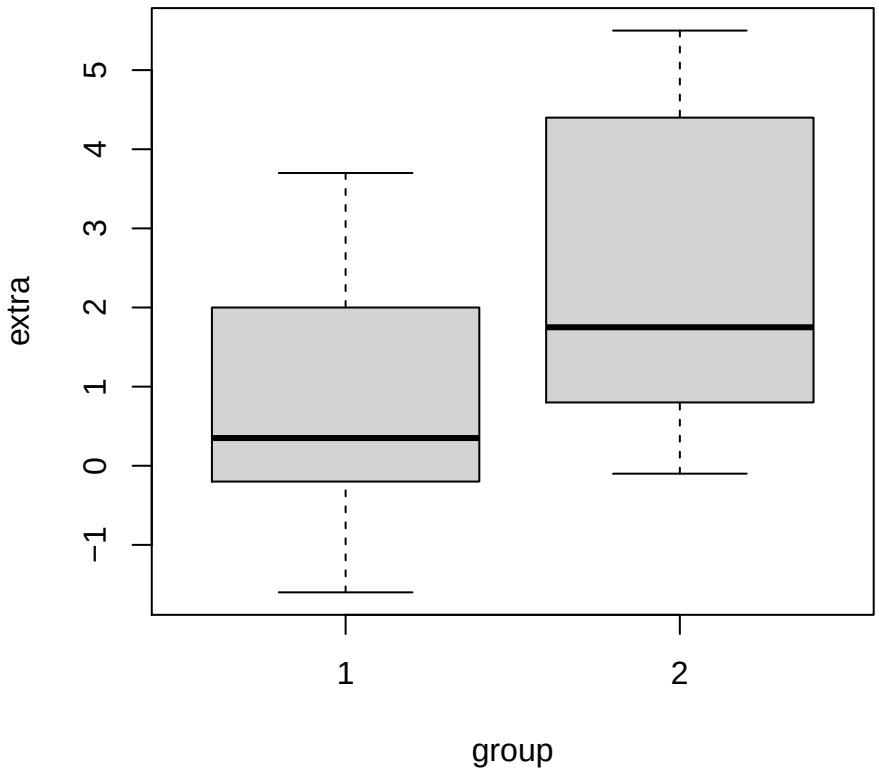


0.128.6 formula interface

The formula interface can be used to group any vector by a factor of the same length.

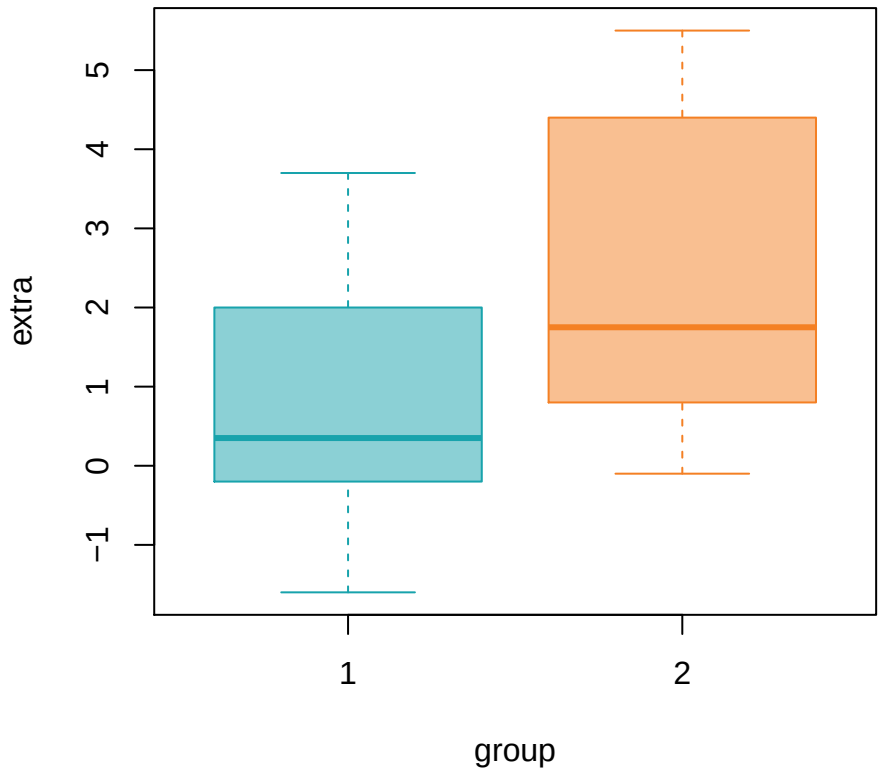
Let's use the built-in `sleep` dataset which shows the effect of two different drugs in increasing hours of sleep compared to a control group.

```
boxplot(extra ~ group, sleep)
```



The **`col`** and **`border`** arguments work as expected. Here we define two custom colors using their hexadecimal RGB code and use the solid version for the border and a 50% transparent version for the fill. Note that we do not need two separate colors to produce an unambiguous plot since they are clearly labeled in the y-axis. It is often considered desirable/prefered to use the minimum number of different colors that is necessary. (Color coding like the following could be useful if for example data from the two groups were used on a different plot, like a scatterplot, in a multi-panel figure).

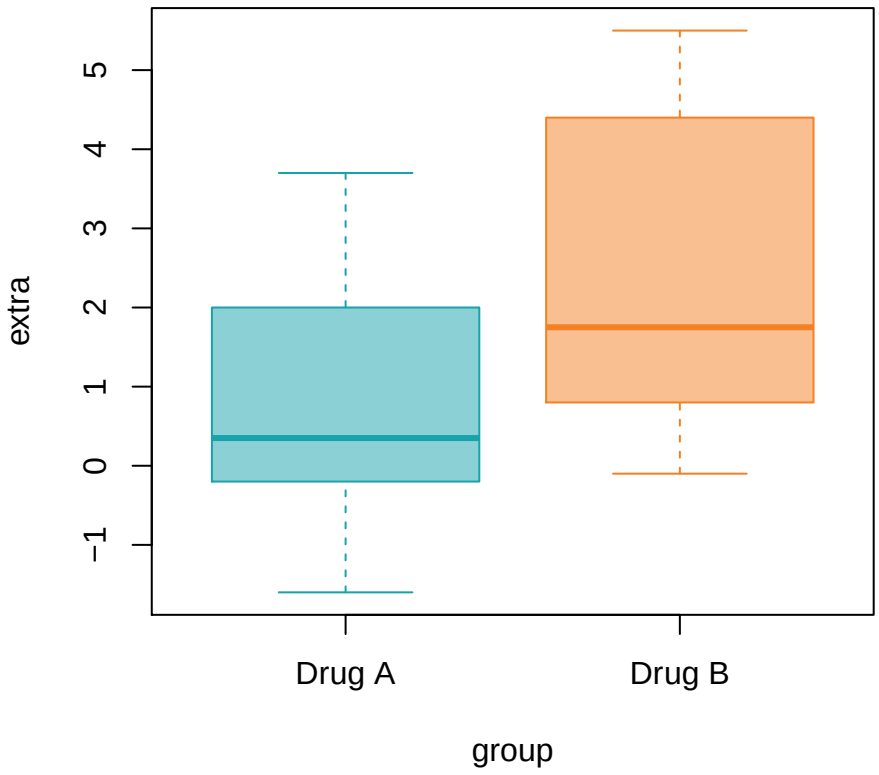
```
border <- c("#18A3AC", "#F48024")
col <- c(adjustcolor("#18A3AC", .5), adjustcolor("#F48024", .5))
boxplot(extra ~ group, sleep,
        col = col, border = border)
```



0.128.7 `names`: group labels

The x-axis group names can be defined with the `names` argument:

```
boxplot(extra ~ group, sleep,
        col = col, border = border,
        names = c("Drug A", "Drug B"))
```



0.129 Heatmap

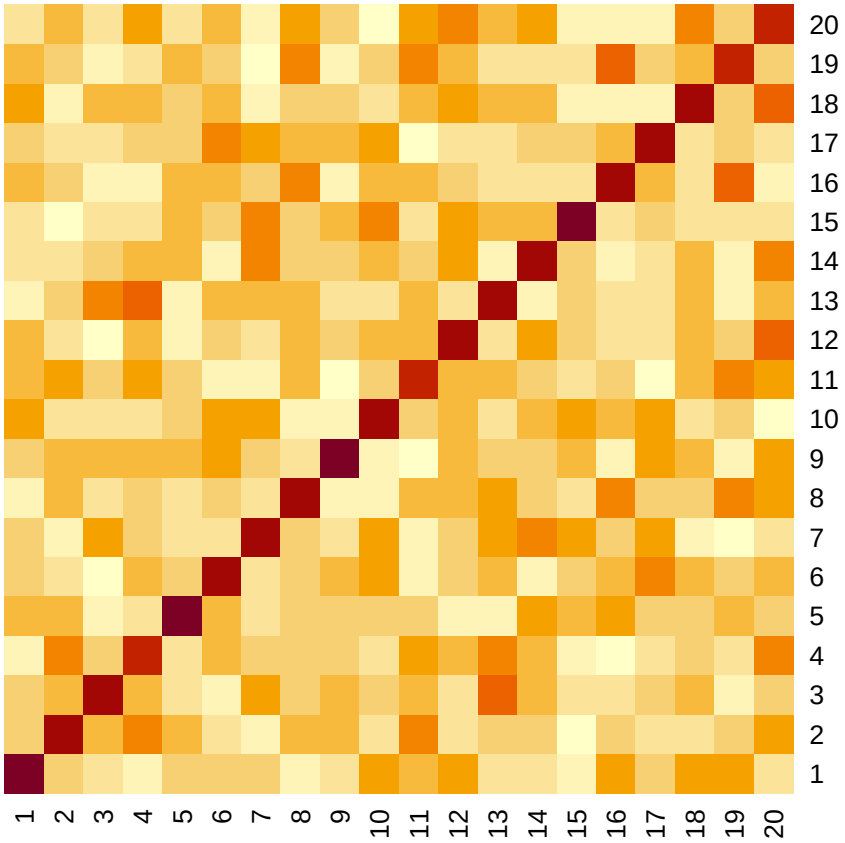
Input: **matrix**

A heatmap is a 2D matrix-like plot with x- and y-axis labels and a value in each cell. It can be used to display many different types of data. A common usage in data science is to plot the correlation matrix of a set of numerical features. In many cases, the rows and/or columns of a heatmap can be reordered based on hierarchical clustering.

```
x <- sapply(1:20, function(i) rnorm(20))
x_cor <- cor(x)
```

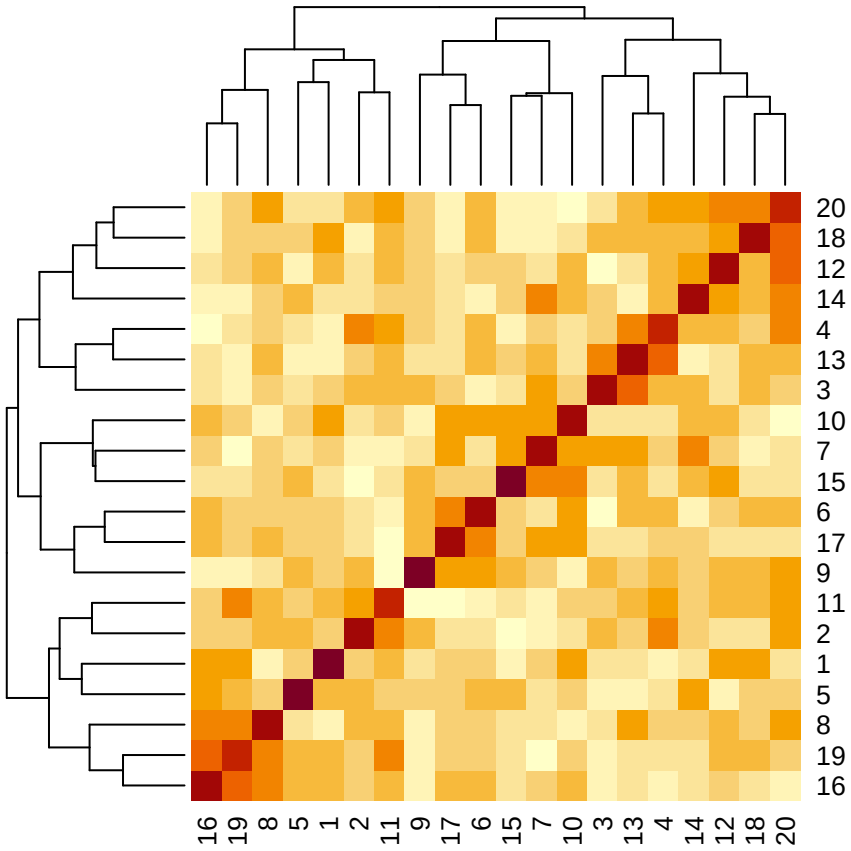
By default, the `heatmap()` function draws marginal dendrograms and rearranges rows and columns. We can prevent that by setting `Rowv` and `Colv` to `NA`:

```
heatmap(x_cor, Rowv = NA, Colv = NA)
```



To allow clustering and row and column reordering, use the defaults:

```
heatmap(x_cor)
```



0.130 Graphical parameters

The `par` function allows setting or querying graphical parameters of the base graphics system. Have a look at its documentation (`?par`).
Some graphical parameters can only be set with a call to `par` prior to using a base plotting function. However, many parameters can also be passed using the `...` construct of each base plotting function.

Some common base graphical parameters:

- *pch*: Point character
- *col*: Color
- *cex*: Character expansion, i.e. relative size
- *bty*: Box type
- *xlab*: x-axis label
- *ylab*: y-axis label
- *main*: Main title

Always make sure that your plotting characters, axis labels and titles are legible. You must avoid, at all costs, ever using a huge graph with tiny letters spread over an entire slide in a presentation.

- *cex*: Character expansion for the plotting characters
- *cex.axis*: cex for axis annotation
- *cex.lab*: cex for x and y labels
- *cex.main*: cex for main title

Note: All of these can be set either with a call to `par()` prior to plotting or passed as arguments in a plotting command, like `plot()`.

There is one important distinction: `cex` set with `par()` (which defaults to 1), sets the baseline and all other `cex` parameters multiply it. However, `cex` set within `plot()` still multiplies `cex` set with `par()`, but only affectts the plotting character size.

3x Graphics

```
library(rtemis)
```

```
  .:rtemis 0.8.1: Welcome, egenn
 [x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
 Documentation & vignettes: https://rtemis.lambdamd.org
```

```
library(ggplot2)
library(plotly)
```

```
Attaching package: 'plotly'
```

```
The following object is masked from 'package:ggplot2':

    last_plot
```

```
The following object is masked from 'package:stats':

    filter
```

```
The following object is masked from 'package:graphics':

    layout
```

```
library(mgcv)
```

```
Loading required package: nlme
```

```
This is mgcv 1.8-33. For overview type 'help("mgcv-package")'.
```

Visualization is central to statistics and data science. It is used to check data, explore data, and communicate results.

R has powerful graphical capabilities built in to the core language. It contains two largely separate graphics systems: 'base' graphics in the `graphics` package, inher-

ited from the S language, and 'grid' graphics in the `grid` package: a "rewrite of the graphics layout capabilities". There is limited support for interaction between the two. In practice, for a given application, choose one or the other. There are no high level functions for the grid graphics system built into the base R distribution, but a few very popular packages have been built on top of it. Both graphics systems can produce beautiful, layered, high quality graphics. It is possible to build functions using either system to produce most, if not all, types of plots.

0.131 Base graphics

Common R plotting functions like `plot`, `barplot`, `boxplot`, `heatmap`, etc. are built ontop of base graphics (Murrell, 2018). Their default arguments provide a minimalist output, but can be tweaked extensively. An advantage of base graphics is they are very fast and relatively easy to extend.

The `par` function allows setting or querying graphical parameters of the base graphics system. Have a look at its documentation (`?par`).
Some graphical parameters can only be set with a call to `par` prior to using a base plotting function. However, many parameters can also be passed using the `...` construct of each base plotting function.

Some common base graphical parameters:

- *pch*: Point character
- *col*: Color
- *cex*: Character expansion, i.e. relative size
- *bty*: Box type
- *xlab*: x-axis label
- *ylab*: y-axis label
- *main*: Main title

Always make sure that your plotting characters, axis labels and titles are legible. You must avoid, at all costs, ever using a huge graph with tiny letters spread over a whole slide in a presentation.

- *cex*: Character expansion for the plotting characters
- *cex.axis*: cex for axis annotation
- *cex.lab*: cex for x and y labels
- *cex.main*: cex for main title

Note: All of these can be set either with a call to `pa()r` prior to plotting or passed as arguments in a plotting command, like `plot()`.
However, there is one important distinction: `cex` set with `par()` (which defaults to 1), sets the baseline and all other `cex` parameters multiply it. However, `cex` set within `plot()` stil multiplies `cex` set with `par()`, but only affectts the plotting character size.

0.132 Grid graphics

The two most popular packages built on top of the `grid` package are:

- lattice[18]
- ggplot2[19] (Wickham, 2011)

0.132.1 `ggplot2`

`ggplot2`, created by Hadley Wickham (Wickham, 2011), follows the Grammar of Graphics[20] approach of Leland Wilkinson (Wilkinson, 2012) and has a very different syntax than base functions.
The general idea is to start by defining the data and then add and/or modify graphical elements in a stepwise manner, which allows one to build complex and layered visualizations. A simplified interface to `ggplot` graphics is provided in the `qplot` function of `ggplot2` (but you should avoid it and use learn to use the `ggplot` command which is fun and much more flexible and useful to know)

0.133 3rd party APIs

There are also third party libraries with R APIs that provide even more modern graphic capabilities to the R user:

- plotly[21] (Sievert et al., 2017)
- rbokeh[22]

Both build interactive plots, which can be viewed in a web browser or exported to bitmap graphics, and both also follow the grammar of graphics paradigm, and therefore follow similar syntax to `ggplot2`.

The **rtemis**[23] package (Gennatas, 2017) provides visualization functions built on top of base graphics (for speed and extendability) and **plotly** (for interactivity):

- mplot3[24] static graphics (base)
- dplot3[25] interactive graphics (plotly)

Let's go over the most common plot types using base graphics, mplot3, dplot3, and ggplot.

[18] https://cran.r-project.org/web/packages/lattice/lattice.pdf
[19] https://ggplot2.tidyverse.org
[20] https://www.springer.com/statistics/computational/book/978-0-387-24544-7
[21] https://plot.ly
[22] https://hafen.github.io/rbokeh/index.html
[23] https://rtemis.lambdamd.org
[24] https://rtemis.lambdamd.org/staticgraphics.html
[25] https://rtemis.lambdamd.org/interactivegraphics.html

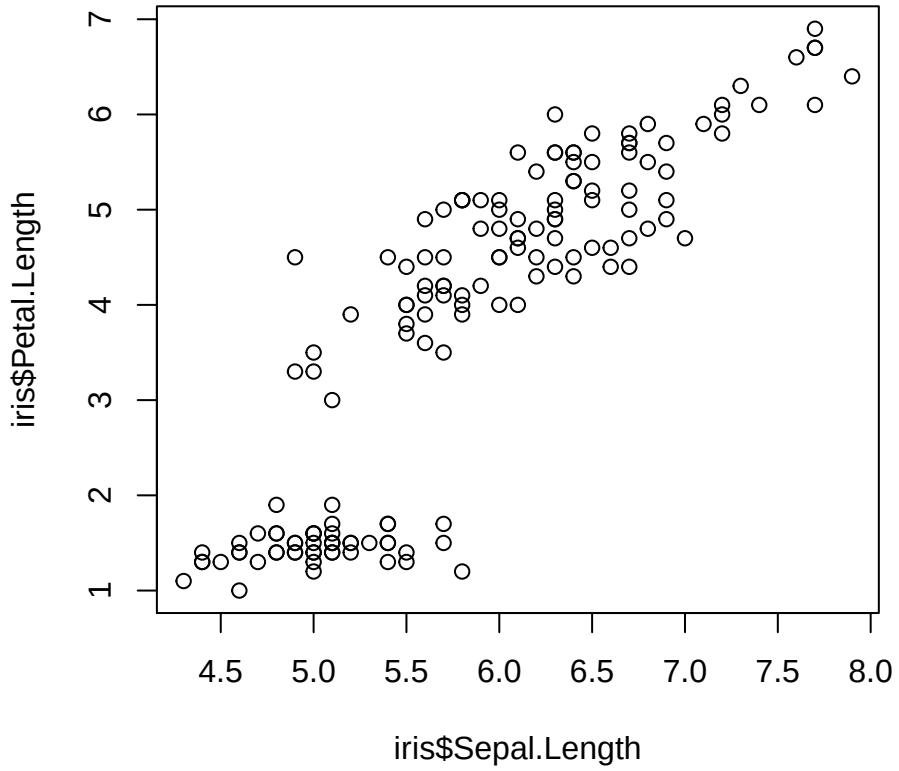
You should be familiar with the basic functionality of both base graphics and ggplot as they are extensively used.

0.134 Scatterplot

0.134.1 base

A default base graphics plot is rather minimalist:

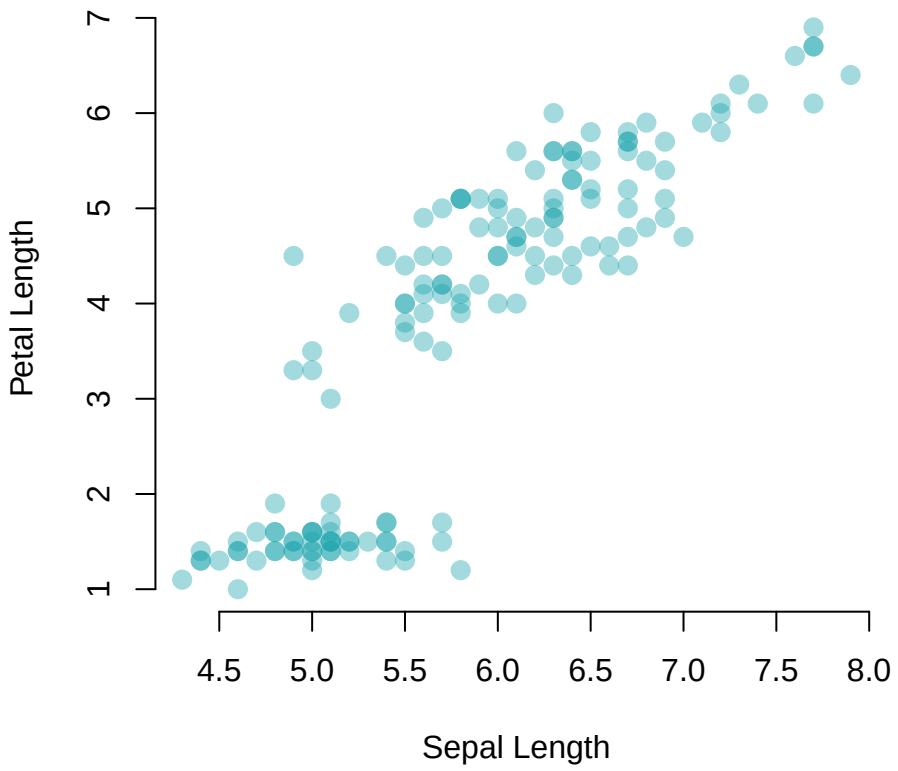
```
plot(iris$Sepal.Length, iris$Petal.Length)
```



By tweaking a few parameters, we get a perhaps prettier result:

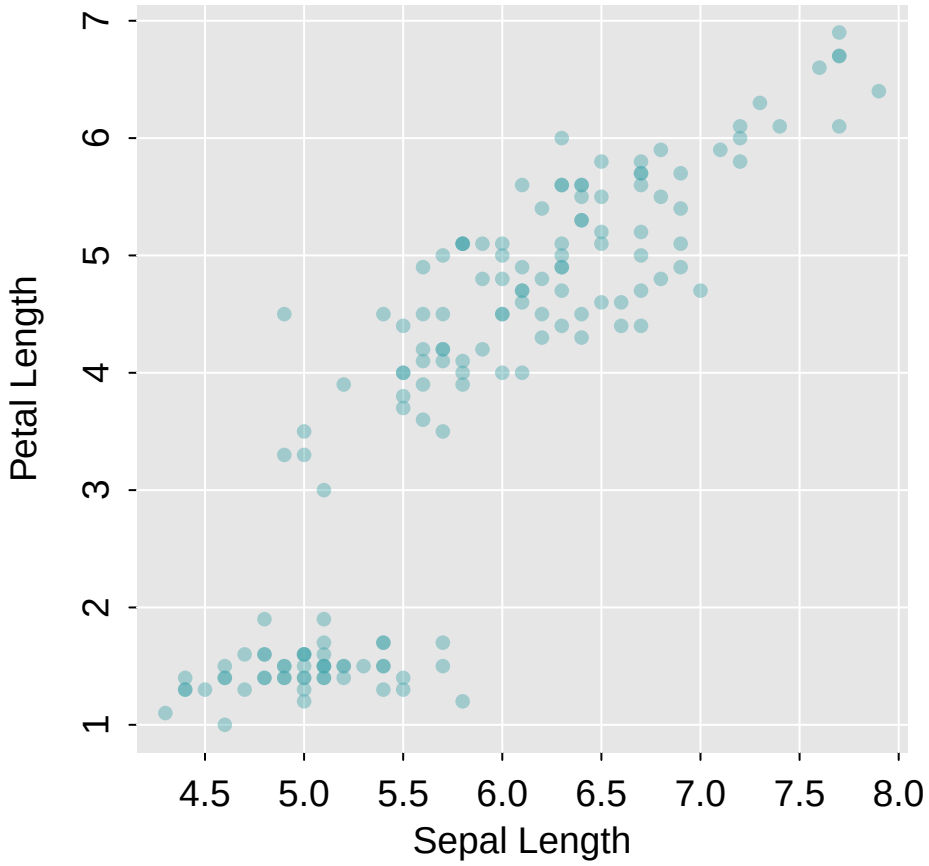
```
plot(iris$Sepal.Length, iris$Petal.Length,
     pch = 16,
     col = "#18A3AC66",
     cex = 1.4,
```

```
    bty = "n",
    xlab = "Sepal Length", ylab = "Petal Length")
```



0.134.2 mplot3

```
mplot3.xy(iris$Sepal.Length, iris$Petal.Length)
```

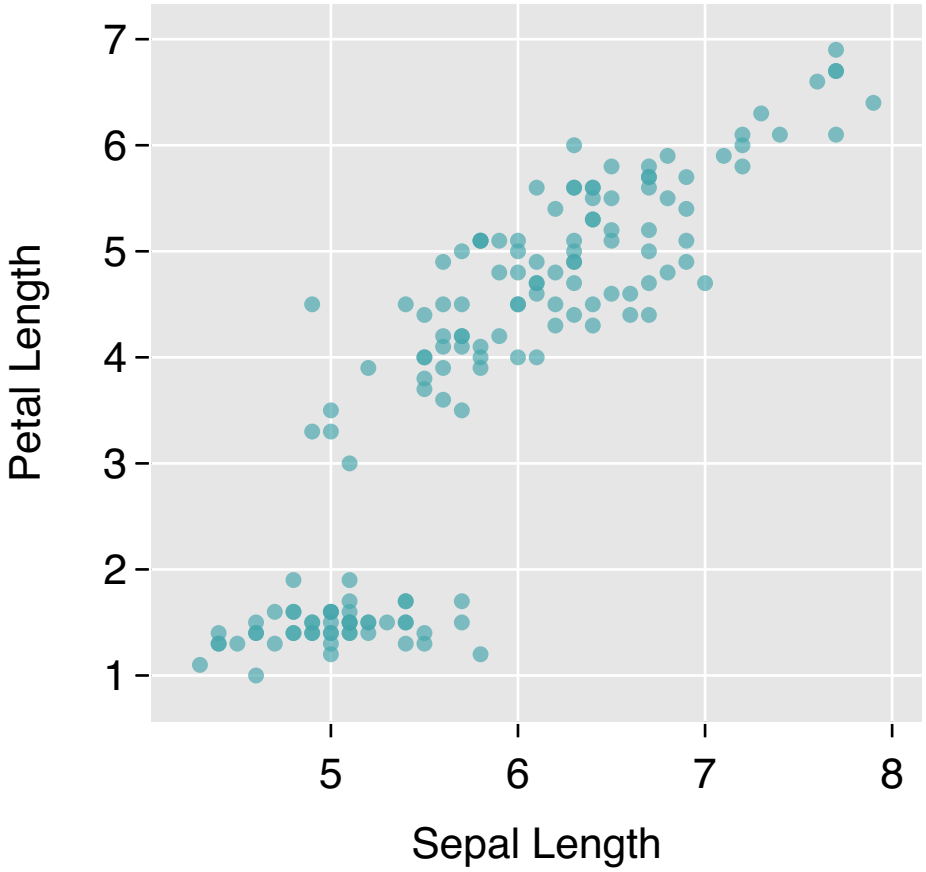


`dplot3()`[26] provides similar functionality to `mplot3()`, built on top of **plotly**. Notice how you can interact with the plot using the mouse:

0.134.3 dplot3

```
dplot3.xy(iris$Sepal.Length, iris$Petal.Length)
```

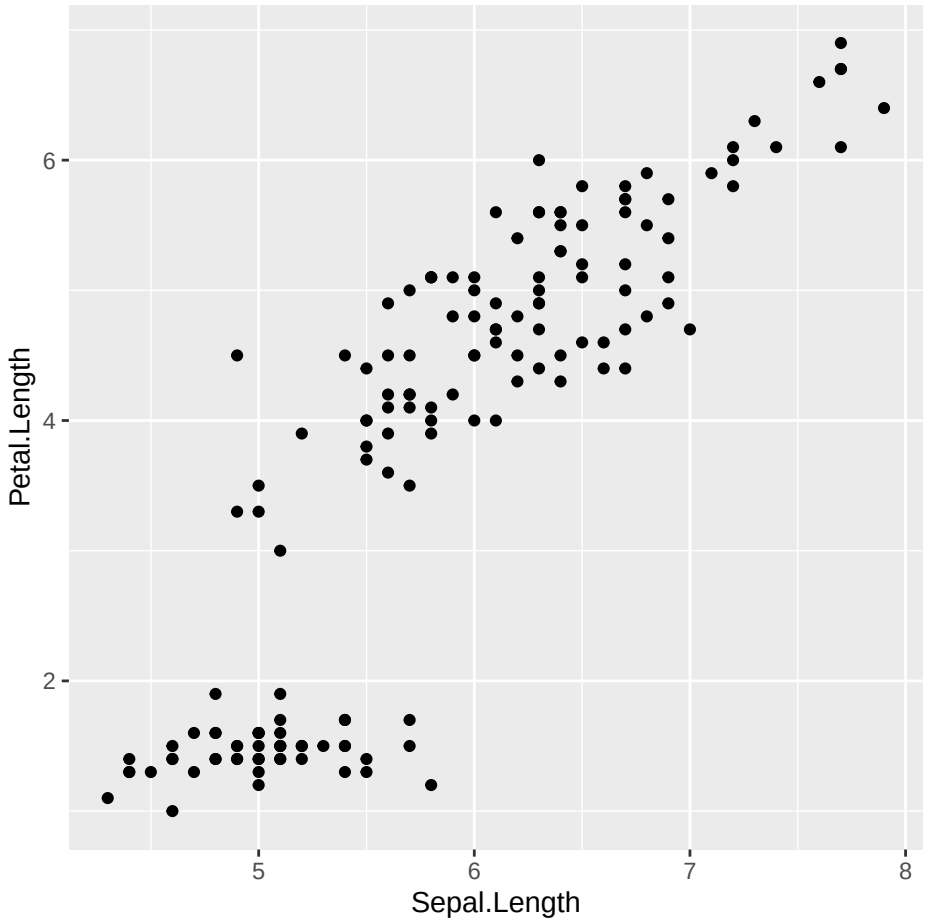
[26] `https://rtemis.lambdamd.org/interactivegraphics.html`



0.134.4 ggplot2

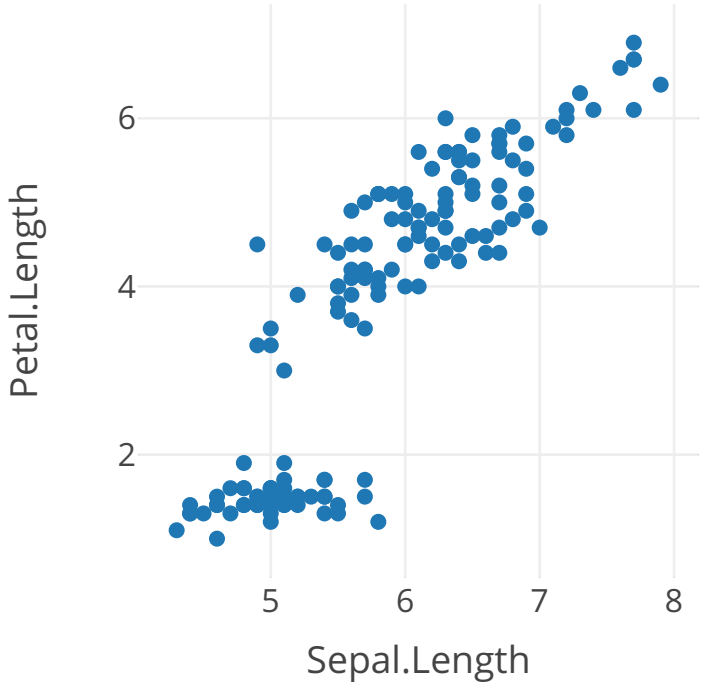
Note: The name of the package is `ggplot2`, the name of the function is `ggplot`.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) + geom_point()
```



0.134.5 plotly

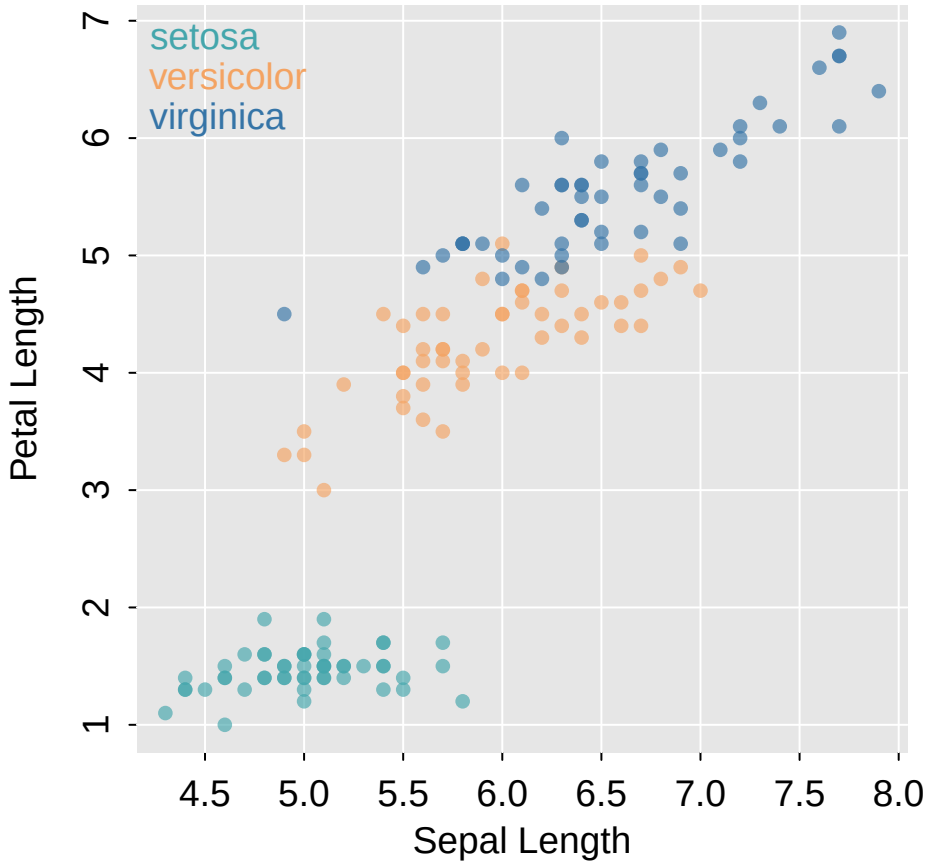
```
p <- plot_ly(iris, x = ~Sepal.Length, y = ~Petal.Length) %>%
  add_trace(type = "scatter", mode = "markers")
p
```



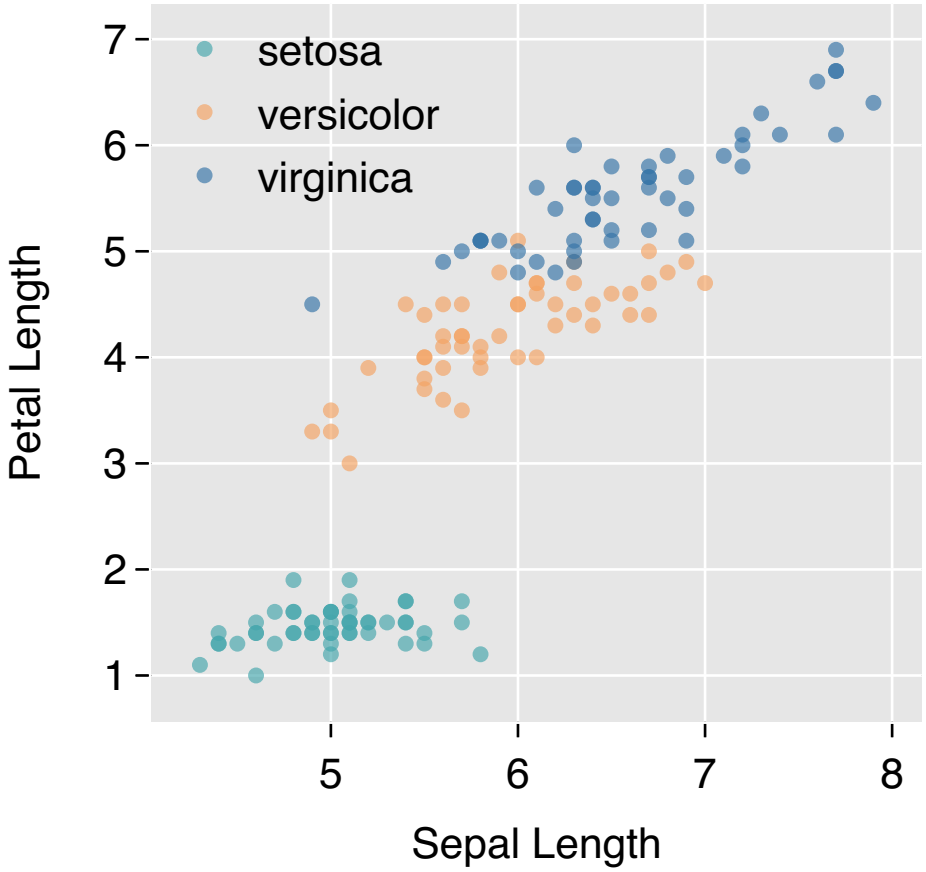
0.134.6 Grouped

In `mplot3()` and `dplot3()`, add a `group` argument:

```
mplot3.xy(iris$Sepal.Length, iris$Petal.Length,
          group = iris$Species)
```

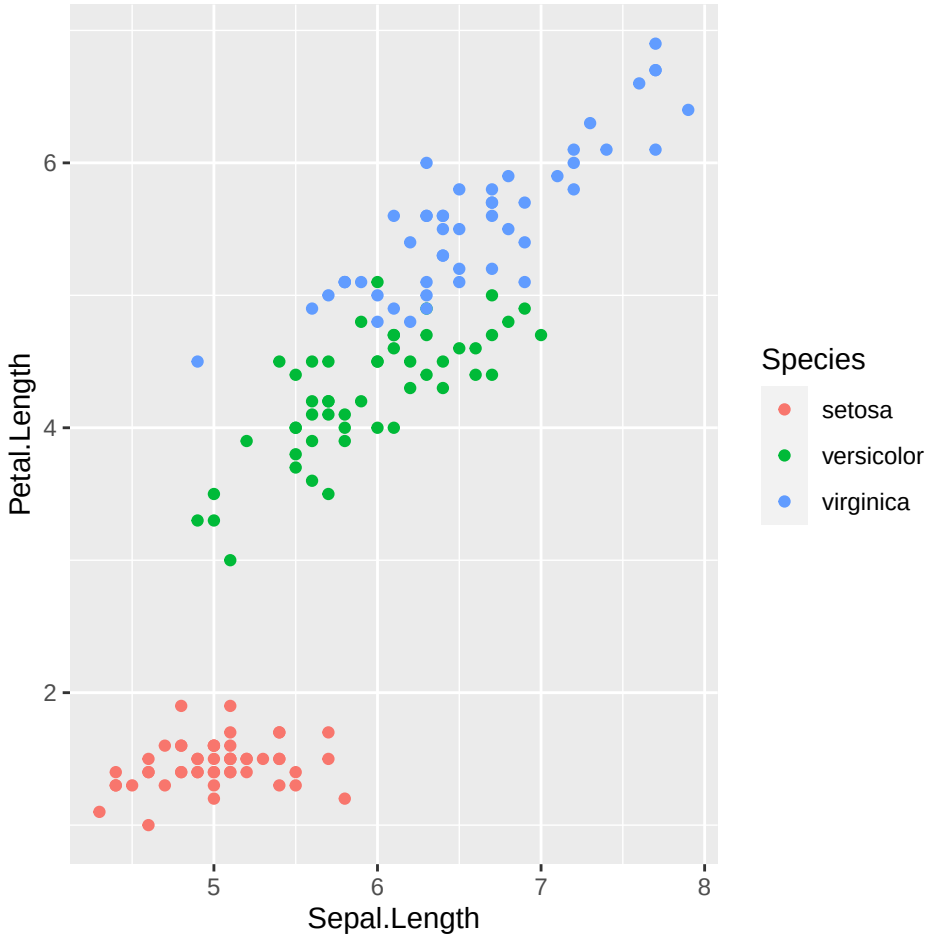



```
dplot3.xy(iris$Sepal.Length, iris$Petal.Length,
          group = iris$Species)
```



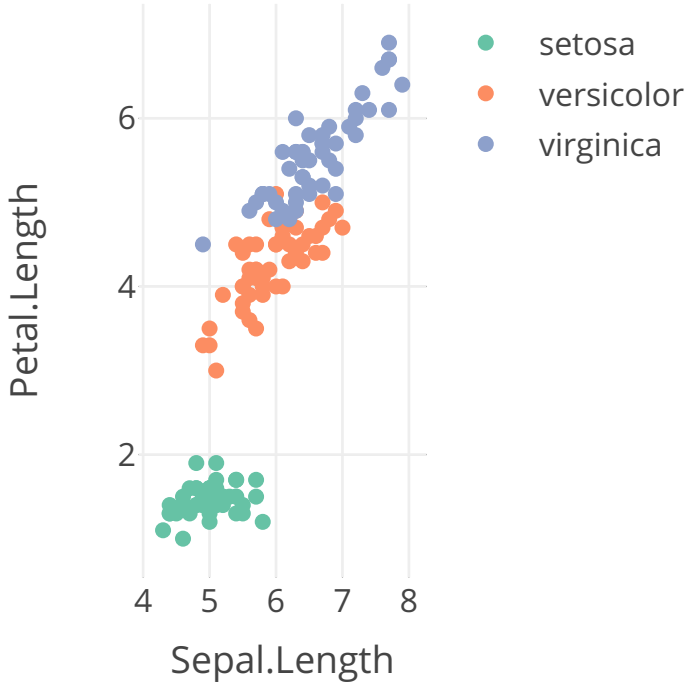
In **ggplot2**, specify `color` within `aes`.
`ggplot` plots can be assigned to an object. Print the object to view it.

```
p <- ggplot(iris, aes(Sepal.Length, Petal.Length, color = Species)) +
  geom_point()
p
```



In **plotly** define the `color` argument:

```
p <- plot_ly(iris, x = ~Sepal.Length, y = ~Petal.Length, color = ~Specie
  add_trace(type = "scatter", mode = "markers")
p
```

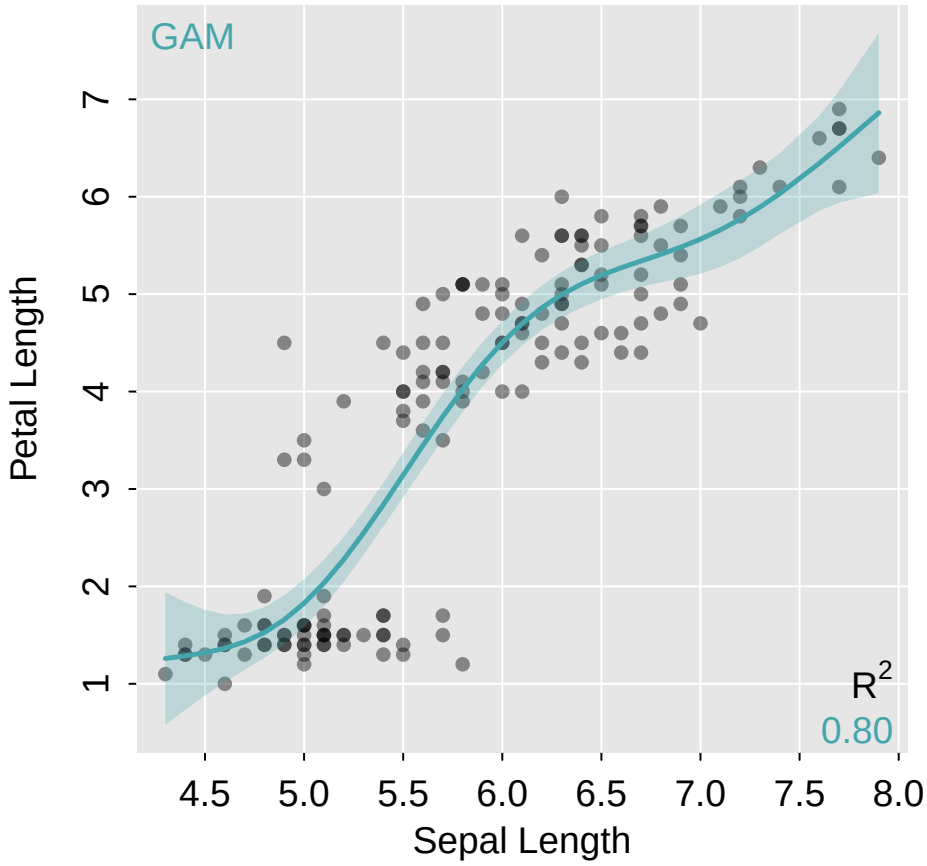


0.135 Scatterplot with fit

0.135.1 mplot3

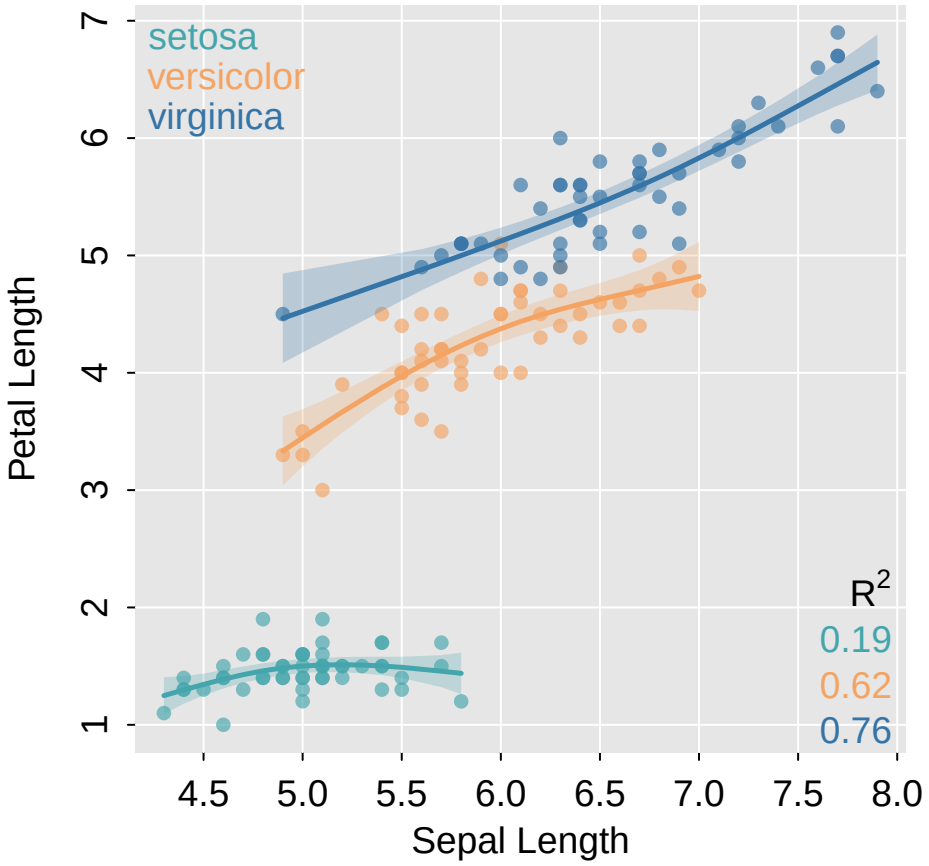
In `mplot3.xy()`, define the algorithm to use to fit a curve, with `fit`. `se.fit` allows plotting the standard error bar (if it can be provided by the algorithm in `fit`)

```
mplot3.xy(iris$Sepal.Length, iris$Petal.Length,
          fit = "gam", se.fit = T)
```



Passing a `group` argument, automatically fits separate models:

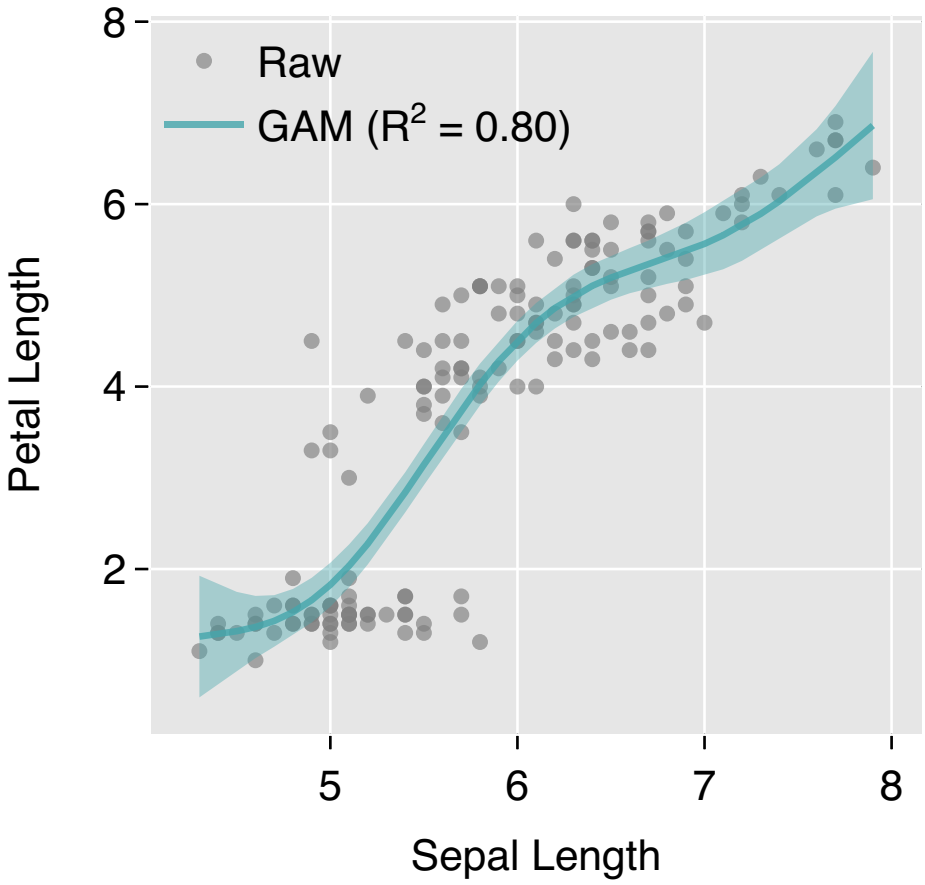
```
mplot3.xy(iris$Sepal.Length, iris$Petal.Length,
          fit = "gam", se.fit = T,
          group = iris$Species)
```



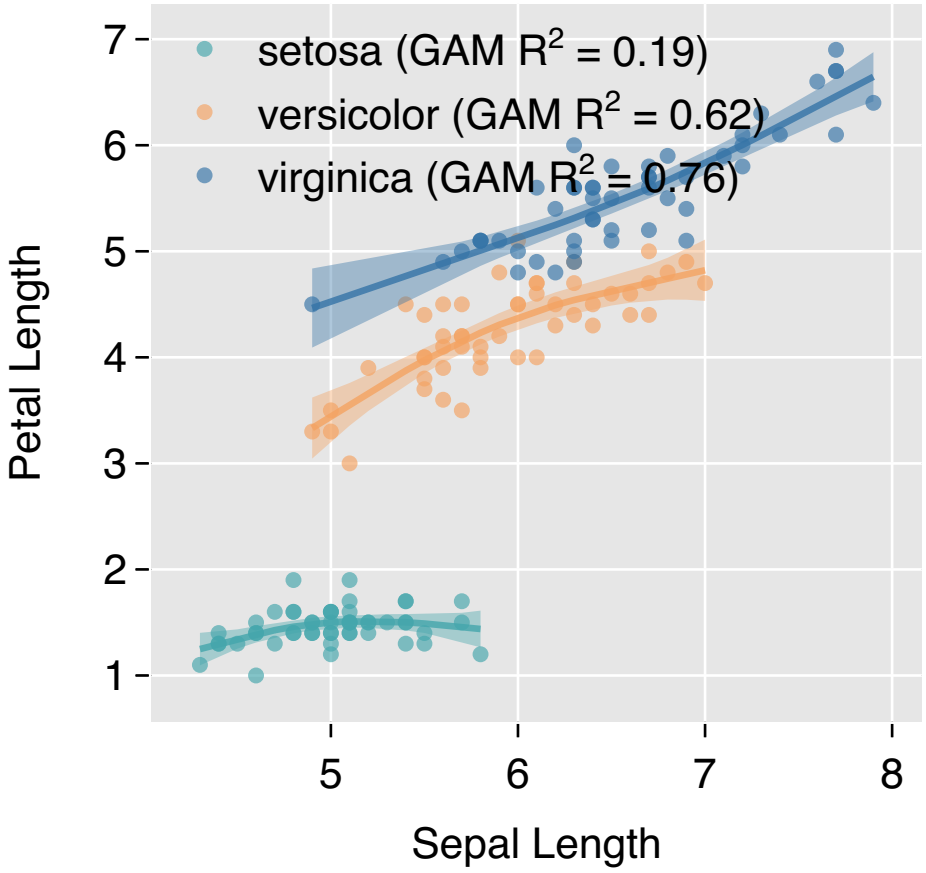
0.135.2 dplot3

Same syntax as `mplot3.xy()` above:

```
dplot3.xy(iris$Sepal.Length, iris$Petal.Length,
          fit = "gam", se.fit = T)
```




```
dplot3.xy(iris$Sepal.Length, iris$Petal.Length,
          fit = "gam", se.fit = T,
          group = iris$Species)
```

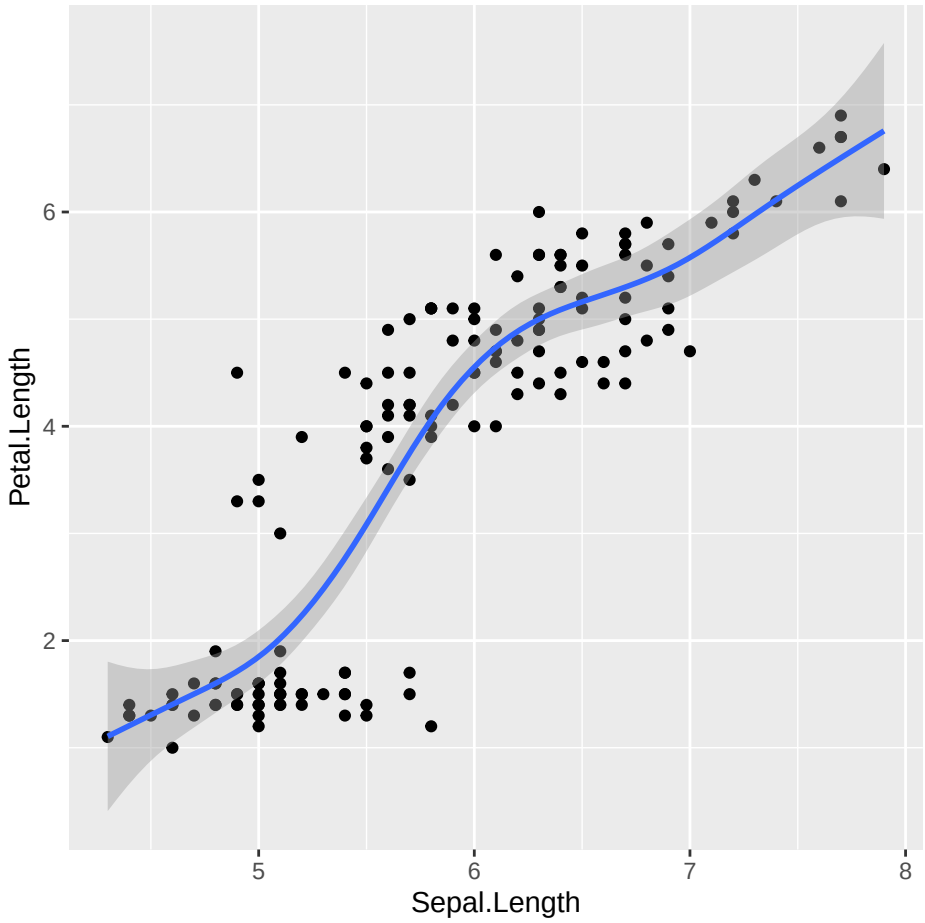


0.135.3 ggplot2

In `ggplot()`, add a `geom_smooth`:

```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length)) +
  geom_point() +
  geom_smooth(method = 'gam')
```

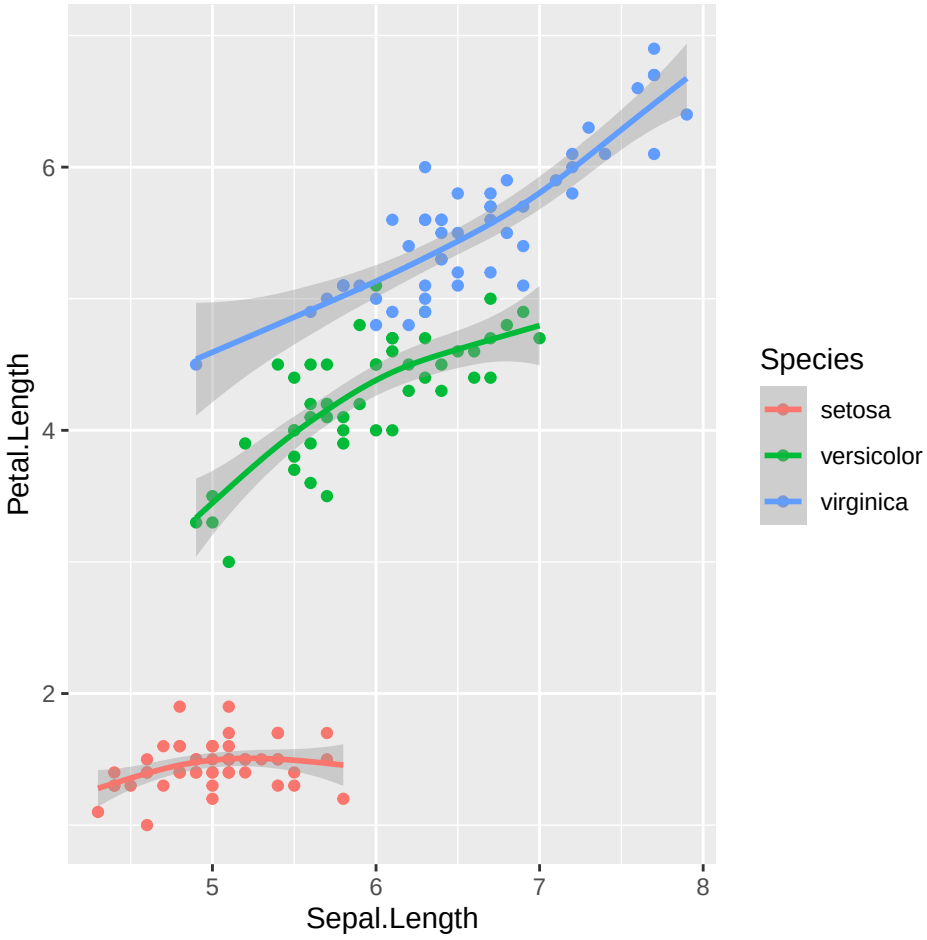
```
`geom_smooth()` using formula 'y ~ s(x, bs = "cs")'
```



To group, again, use `color`:

```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length, color = Species)) +
  geom_point() +
  geom_smooth(method = 'gam')
```

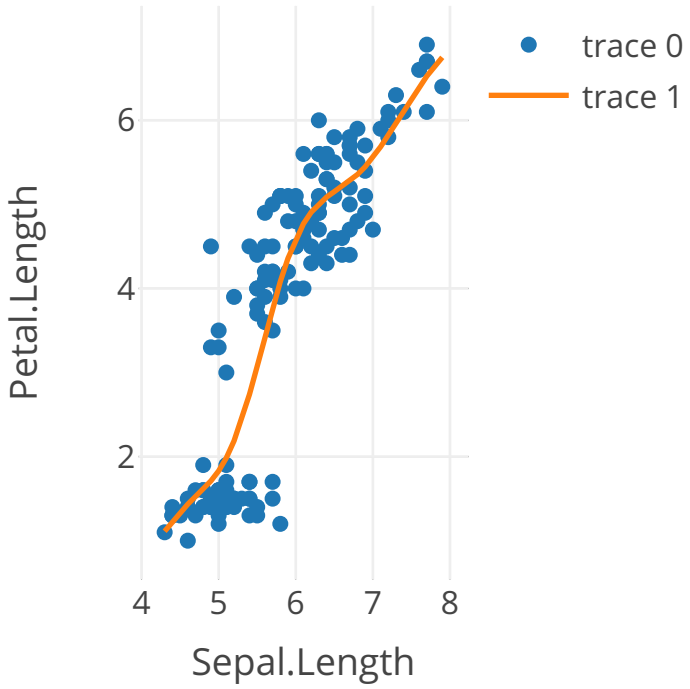
```
`geom_smooth()` using formula 'y ~ s(x, bs = "cs")'
```



0.135.4 plotly

In `plot_ly()`, `add_lines()`:

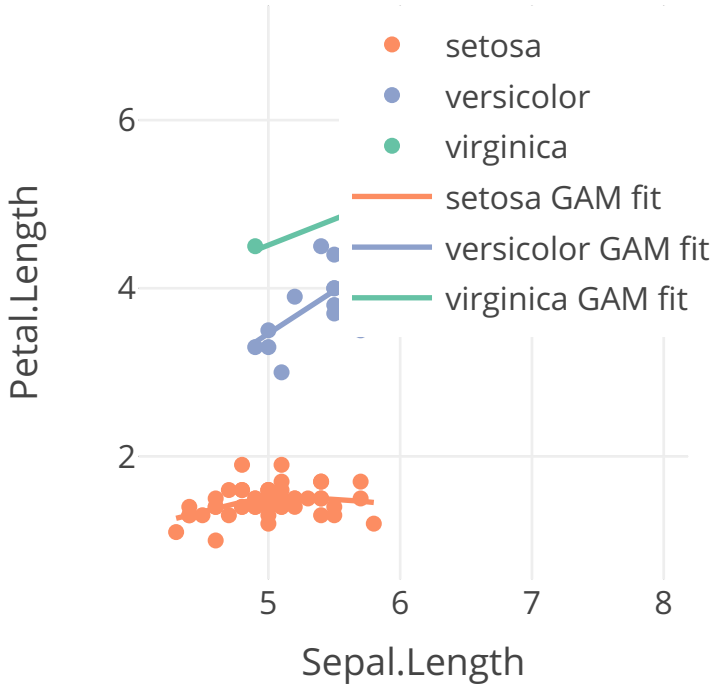
```
library(mgcv)
mod.gam <- gam(Petal.Length ~ s(Sepal.Length), data = iris)
plot_ly(iris, x = ~Sepal.Length) %>%
  add_trace(y = ~Petal.Length, type = "scatter", mode = "markers") %>%
  add_lines(y = mod.gam$fitted.values)
```



To get fit by group, you add all elements one after the other - one way would be this:

```
iris.bySpecies <- split(iris, iris$Species)
gam.fitted <- lapply(iris.bySpecies, function(i) {
  gam(Petal.Length ~ s(Sepal.Length), data = i)$fitted
})
index <- lapply(iris.bySpecies, function(i) order(i$Sepal.Length))
col <- c("#44A6AC", "#F4A362", "#3574A7")
.names <- names(iris.bySpecies)
p <- plot_ly()
for (i in seq_along(iris.bySpecies)) {
  p <- add_trace(p, x = ~Sepal.Length, y = ~Petal.Length,
                 type = "scatter", mode = "markers",
                 data = iris.bySpecies[[i]],
                 name = .names[i],
                 color = col[i])
}
for (i in seq_along(iris.bySpecies)) {
  p <- add_lines(p, x = iris.bySpecies[[i]]$Sepal.Length[index[[i]]],
                 y = gam.fitted[[i]][index[[i]]],
                 # type = "scatter", mode = "markers",
                 data = iris.bySpecies[[i]],
```

```
                name = paste(.names[i], "GAM fit"),
                color = col[i])
}
p
```



It's a lot of work, and that's why `dplot3()` exists.

0.136 Density plot

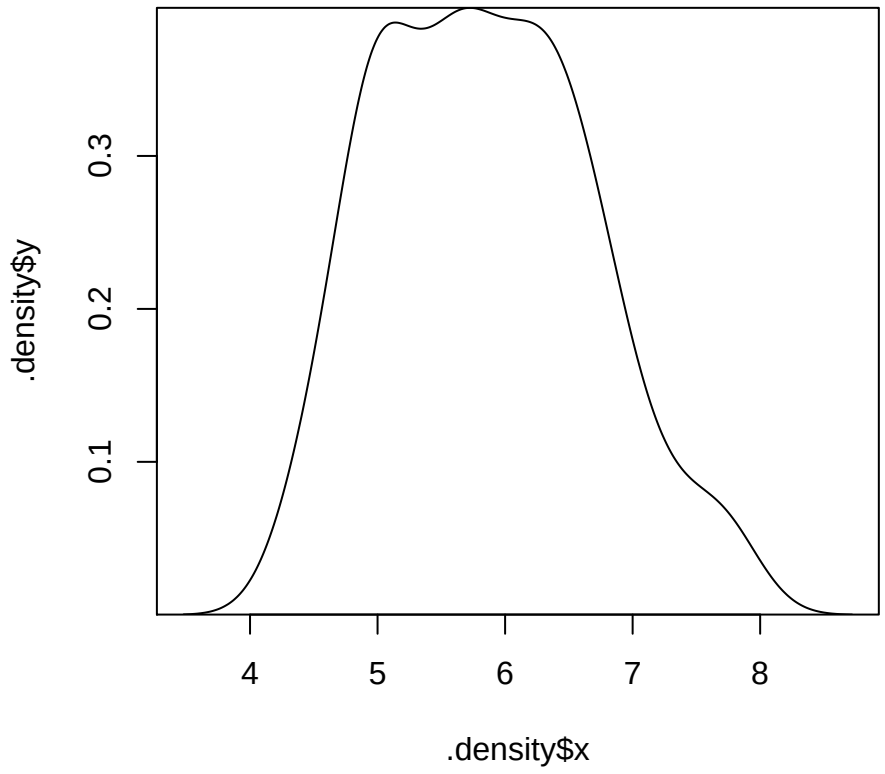
There is no builtin density plot, but you can get x and y coordinates from the `density` function and add a polygon:

0.136.1 base

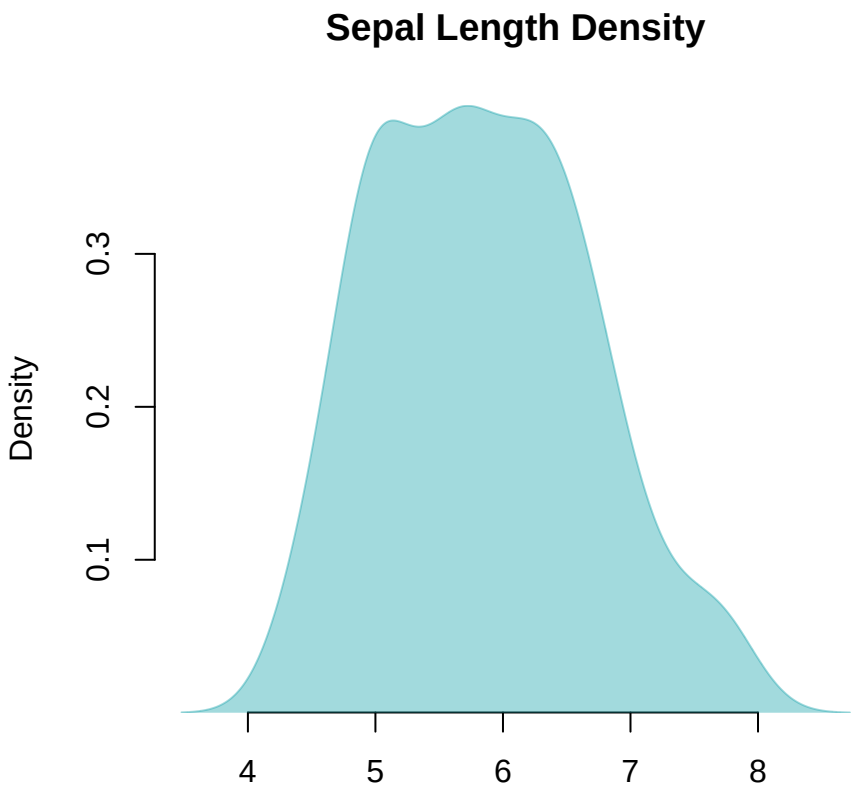
```
.density <- density(iris$Sepal.Length)
class(.density)
```

```
[1] "density"
```

```
plot(.density$x, .density$y,
     type = "l", yaxs = "i")
```

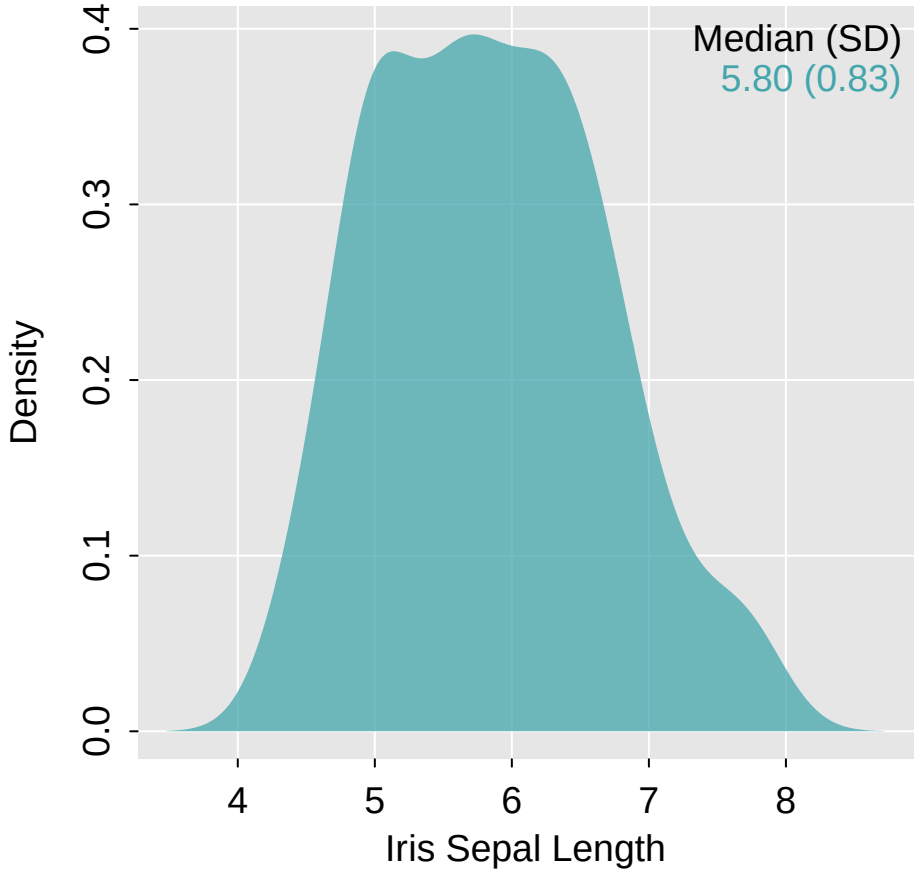



```
plot(.density$x, .density$y,
     type = 'l', yaxs = "i",
     bty = "n",
     xlab = "",  ylab = "Density",
     col = "#18A3AC66",
     main = "Sepal Length Density")
polygon(c(.density$x, rev(.density$x)), c(.density$y, rep(0, length(.den
        col = "#18A3AC66", border = NA)
```



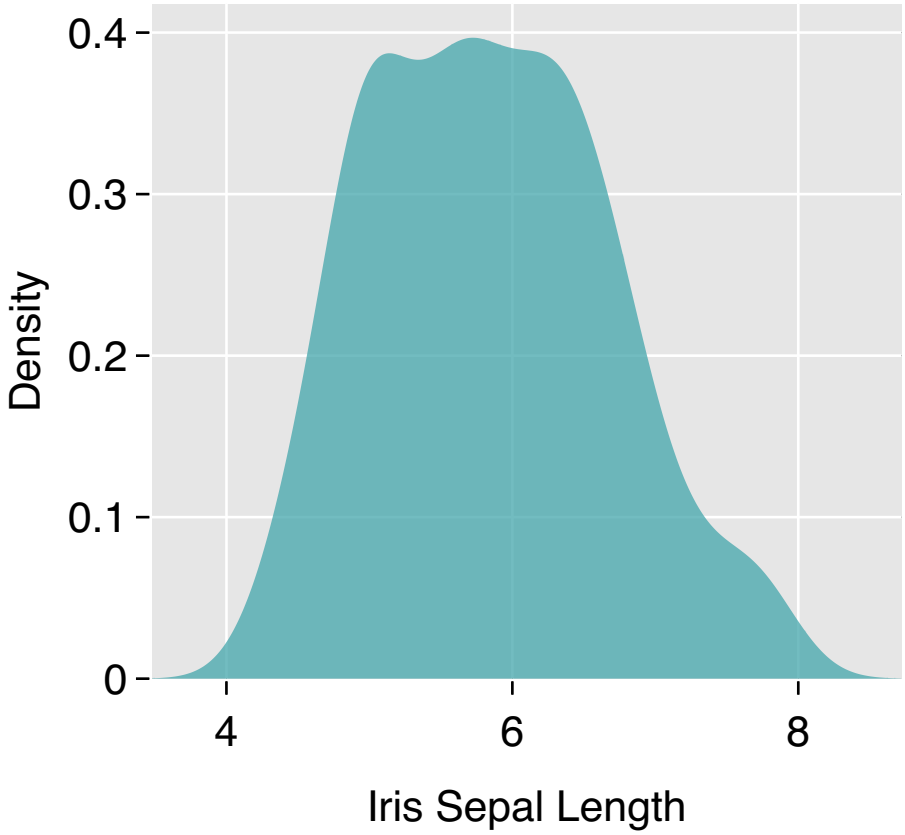
0.136.2 mplot3

```
mplot3.x(iris$Sepal.Length, 'density')
```



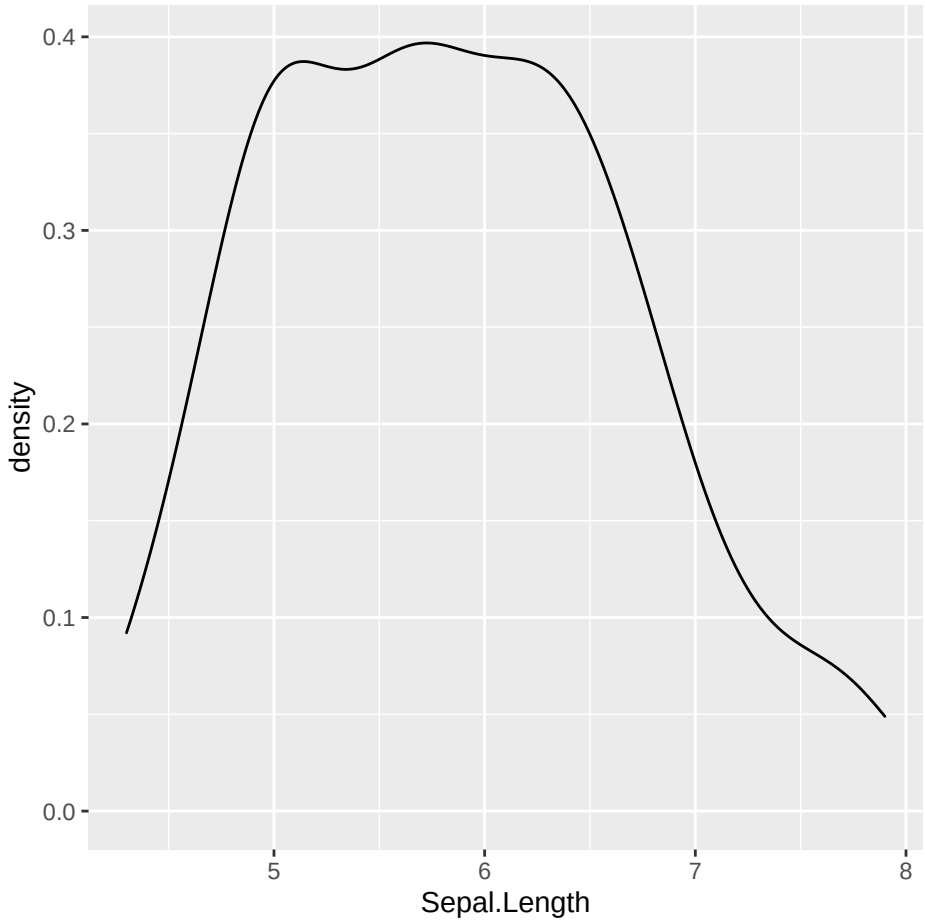
0.136.3 dplot3

```
dplot3.x(iris$Sepal.Length)
```



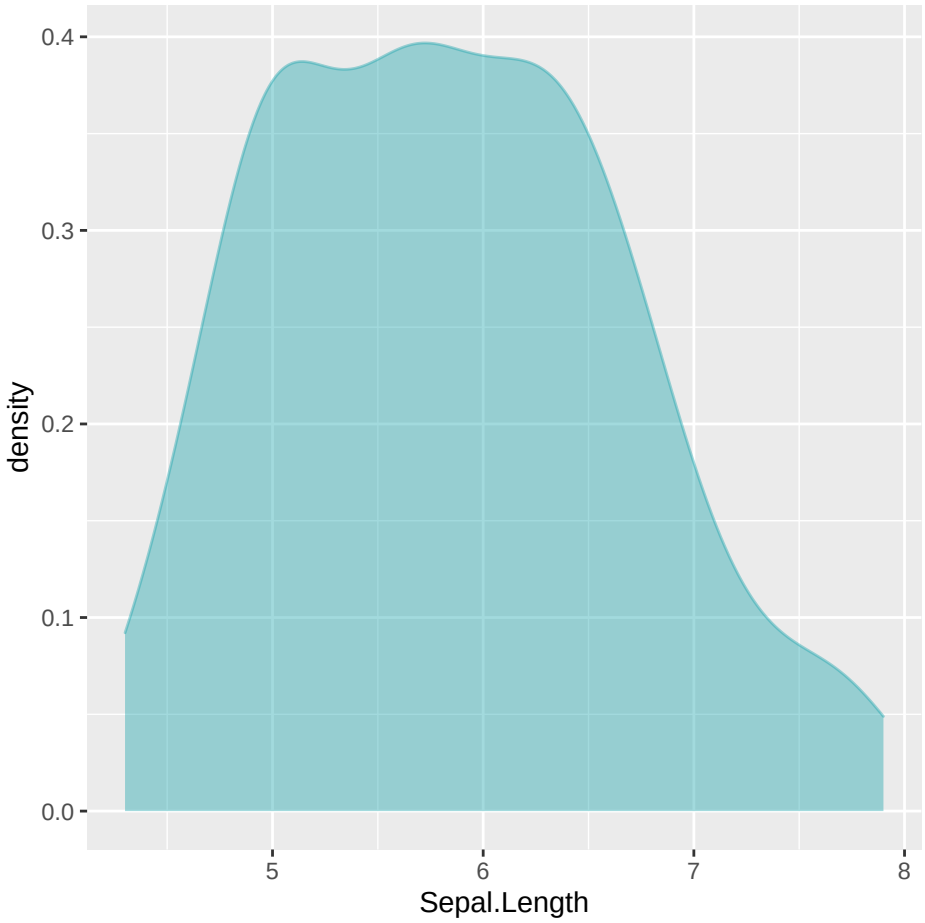
0.136.4 ggplot2

```
ggplot(iris, aes(x = Sepal.Length)) + geom_density()
```



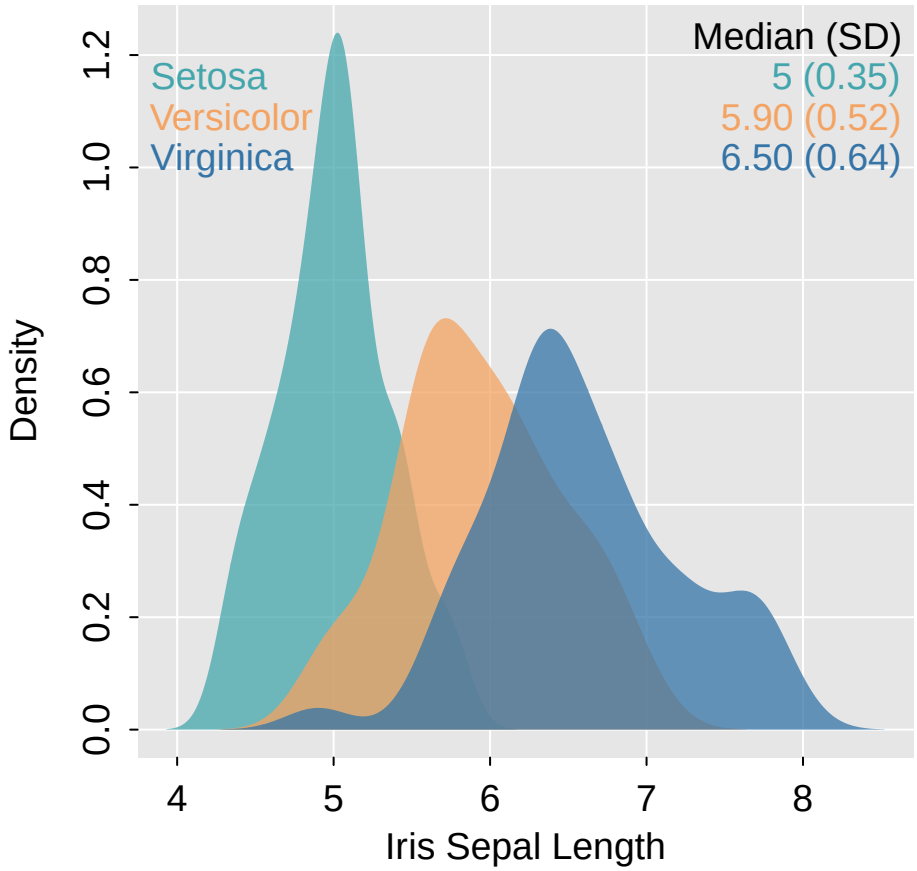
Add color:

```
ggplot(iris, aes(x = Sepal.Length)) + geom_density(color = "#18A3AC66",
```

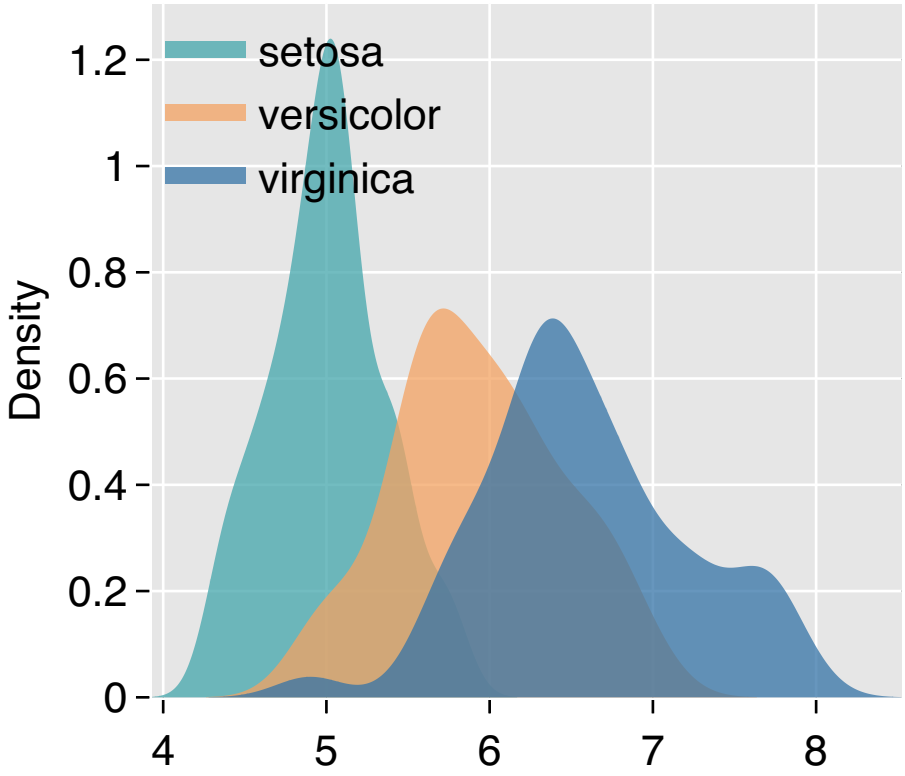


0.136.4.1 Grouped

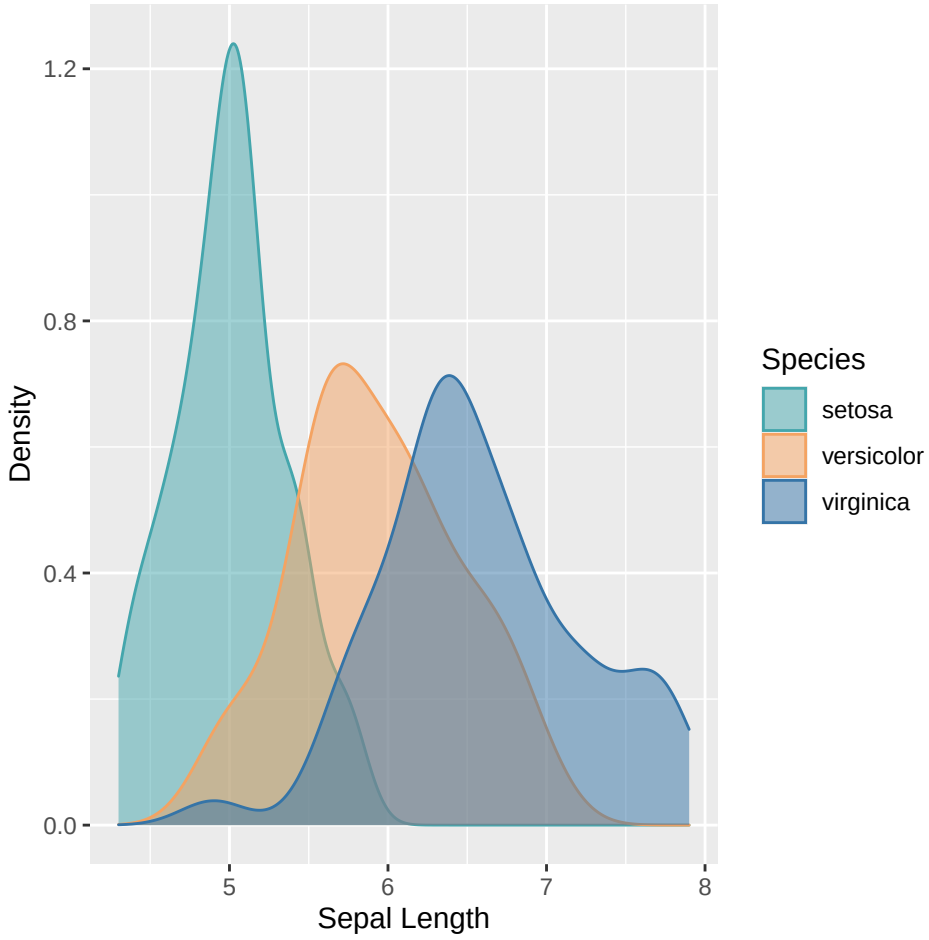
```
mplot3.x(iris$Sepal.Length, group = iris$Species)
```




```
dplot3.x(iris$Sepal.Length, group = iris$Species)
```




```
(ggplot(iris, aes(Sepal.Length, color = Species, fill = Species)) +
  geom_density(alpha = .5) +
  scale_color_manual(values = c("#44A6AC", "#F4A362", "#3574A7")) +
  scale_fill_manual(values = c("#44A6AC", "#F4A362", "#3574A7")) +
  labs(x = "Sepal Length", y = "Density"))
```

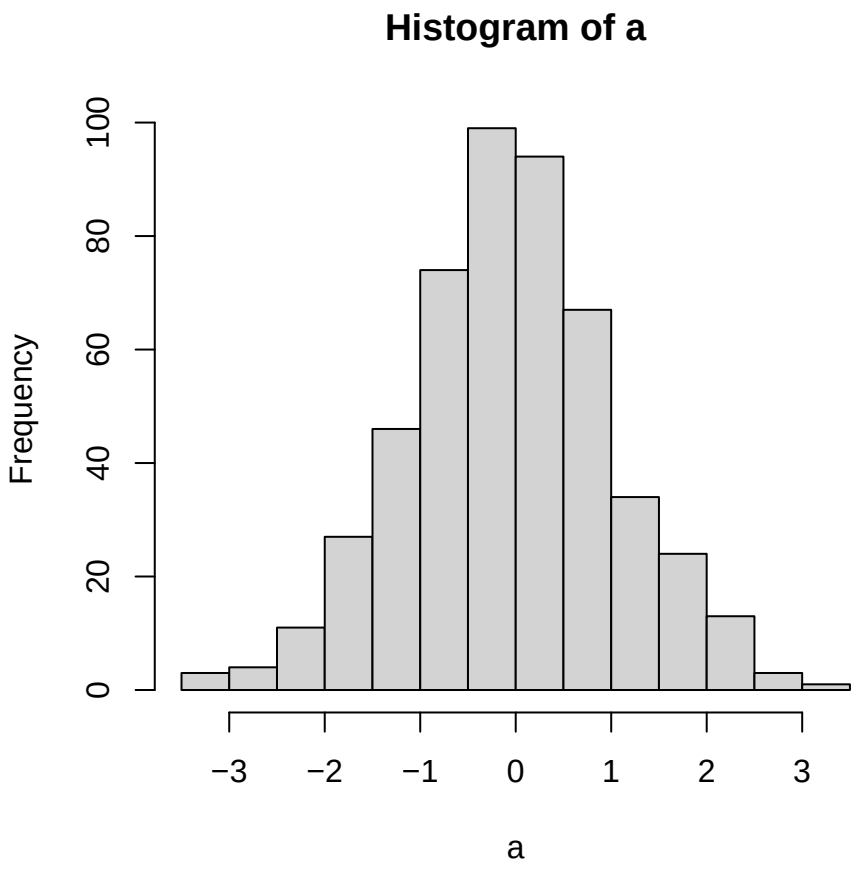


0.137 Histogram

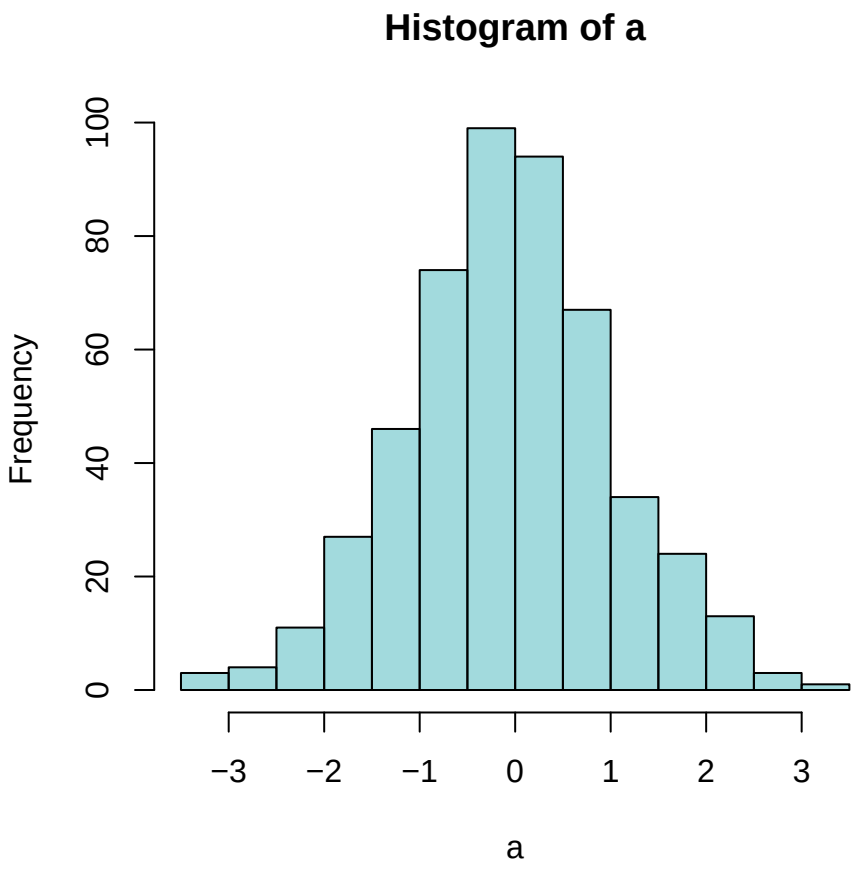
```
set.seed(2020)
a <- rnorm(500)
```

0.137.1 base

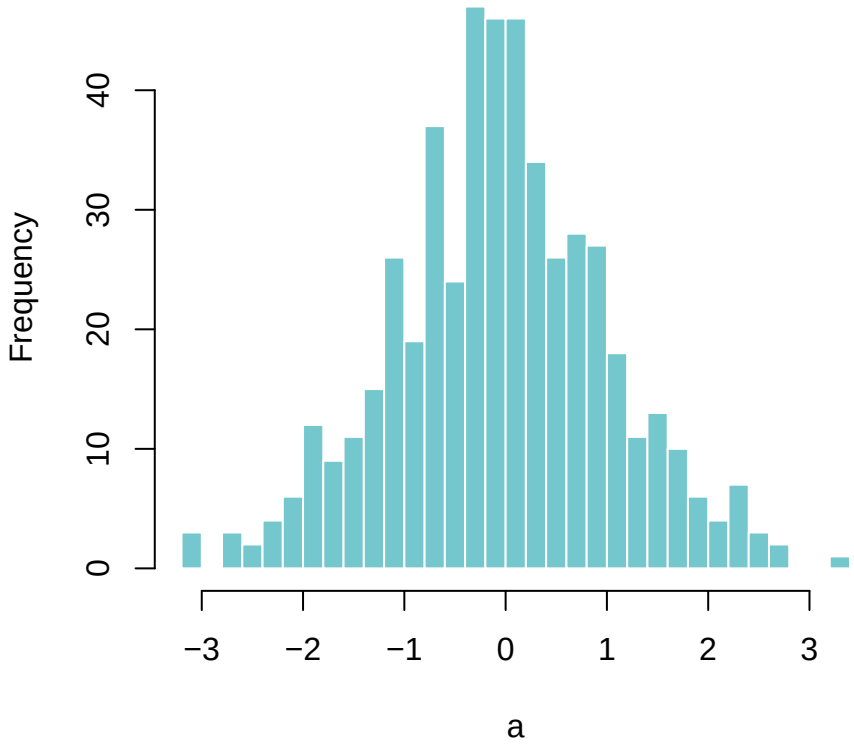
```
hist(a)
```




```
hist(a, col = "#18A3AC66")
```

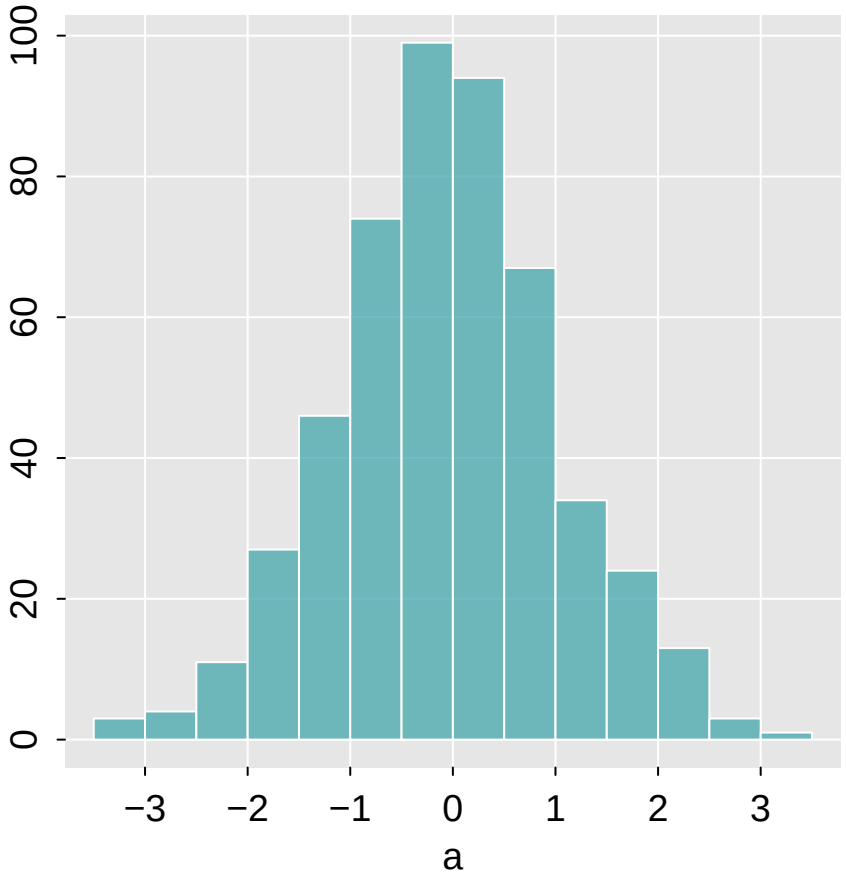



```
hist(a, col = "#18A3AC99", border = "white", main = "", breaks = 30)
```

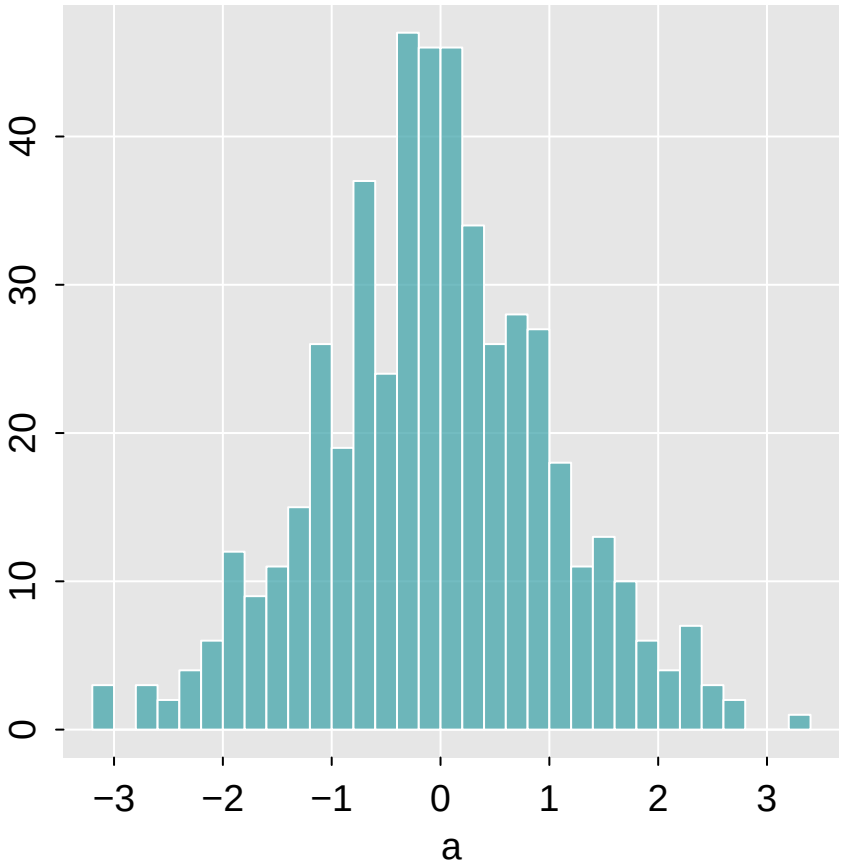


0.137.2 mplot3

```
mplot3.x(a, "histogram")
```

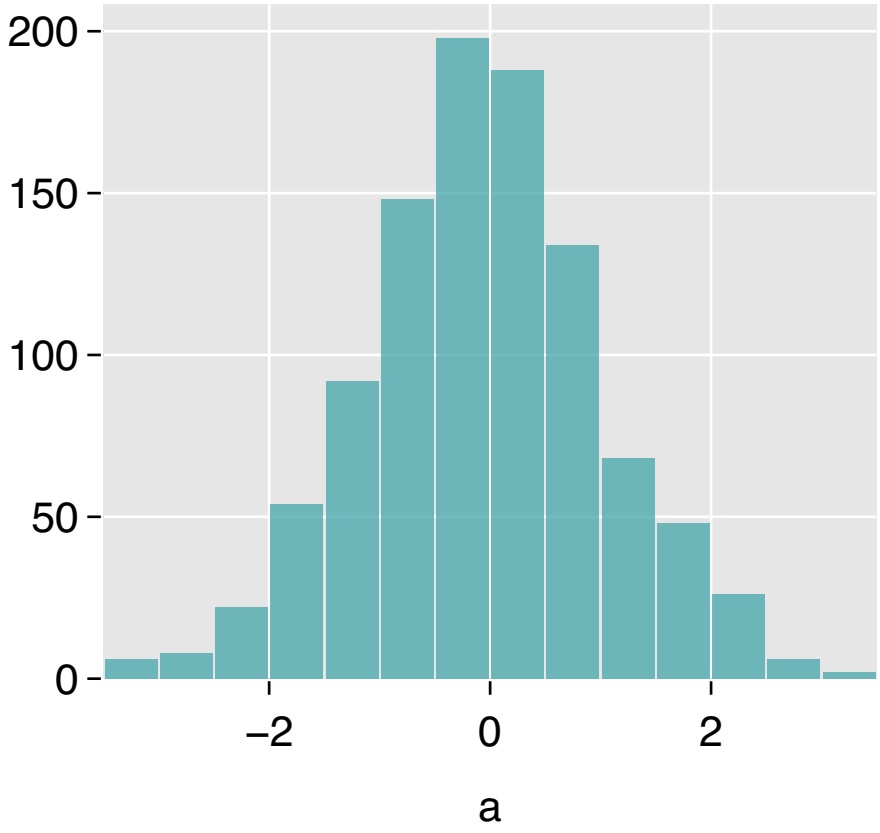



```
mplot3.x(a, "histogram", hist.breaks = 30)
```

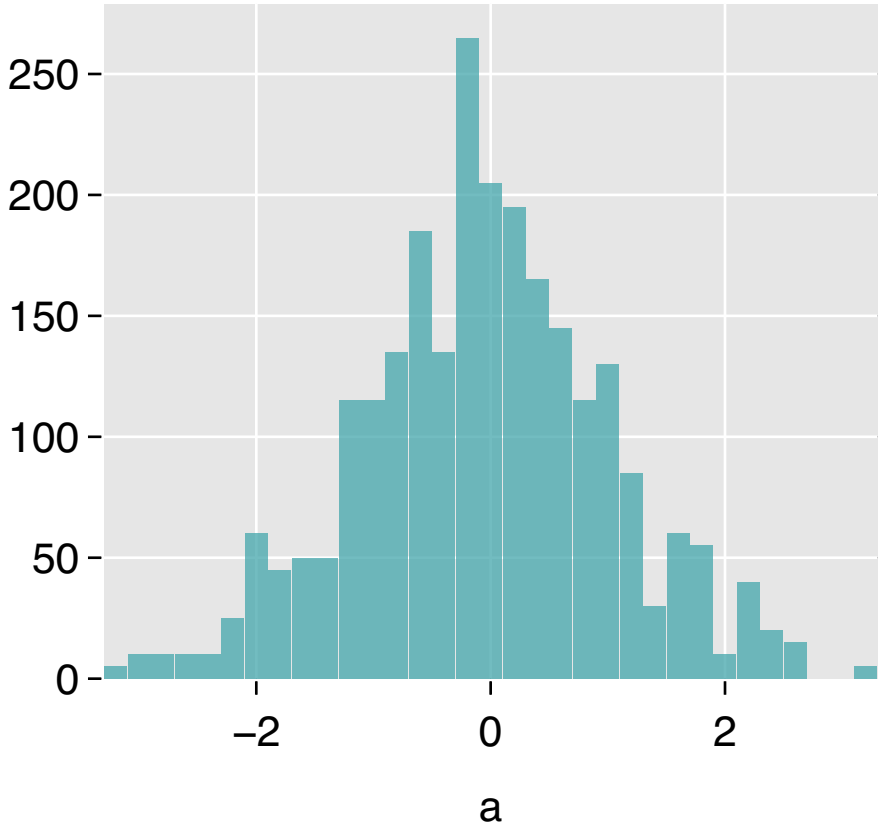


0.137.3 dplot3

```
dplot3.x(a, "hist")
```



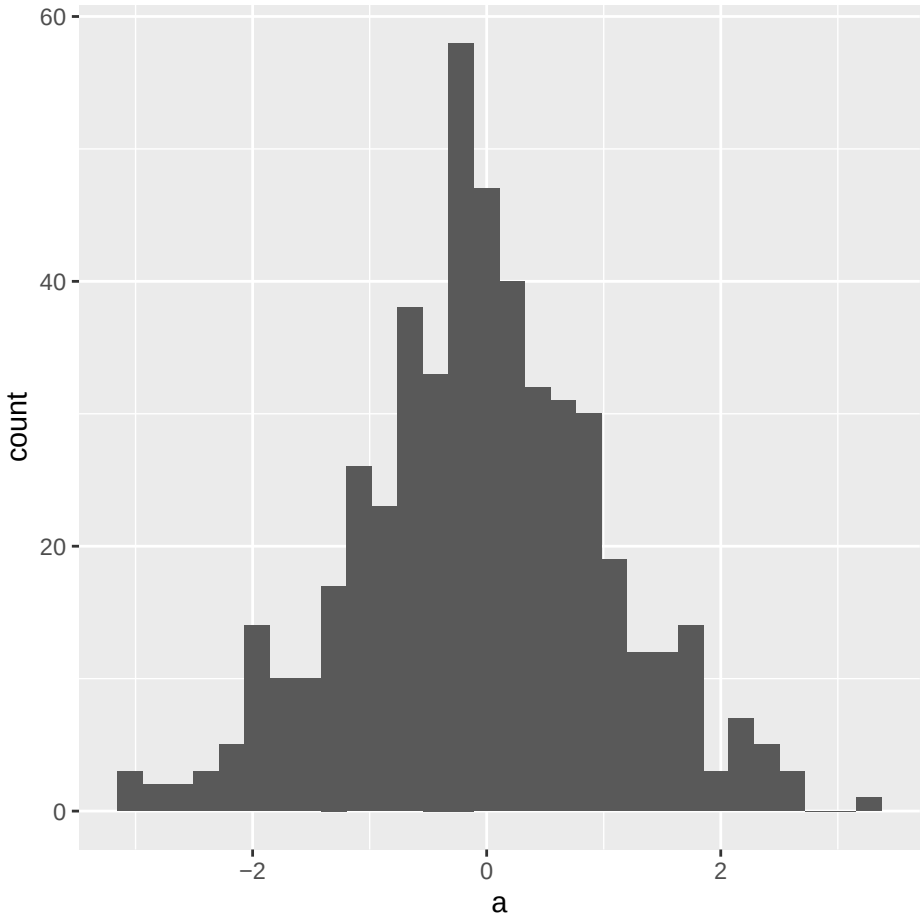

```
dplot3.x(a, "hist", hist.n.bins = 40)
```



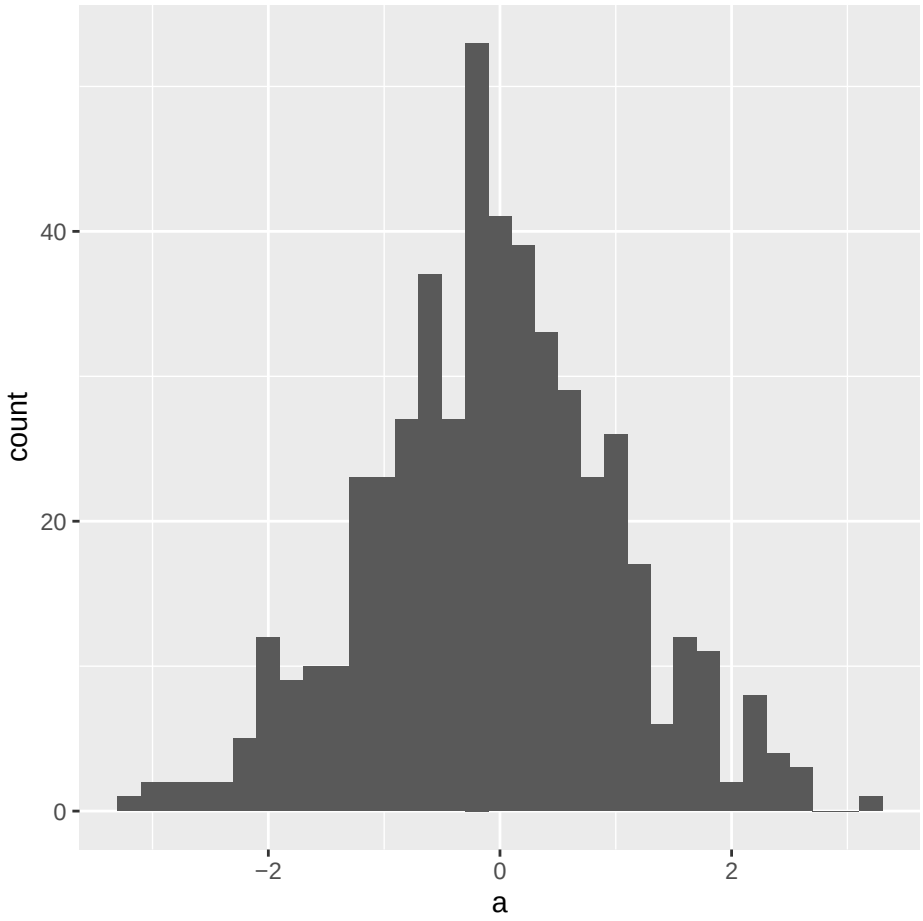
0.137.4 ggplot2

```
(p <- ggplot(mapping = aes(a)) + geom_histogram())
```

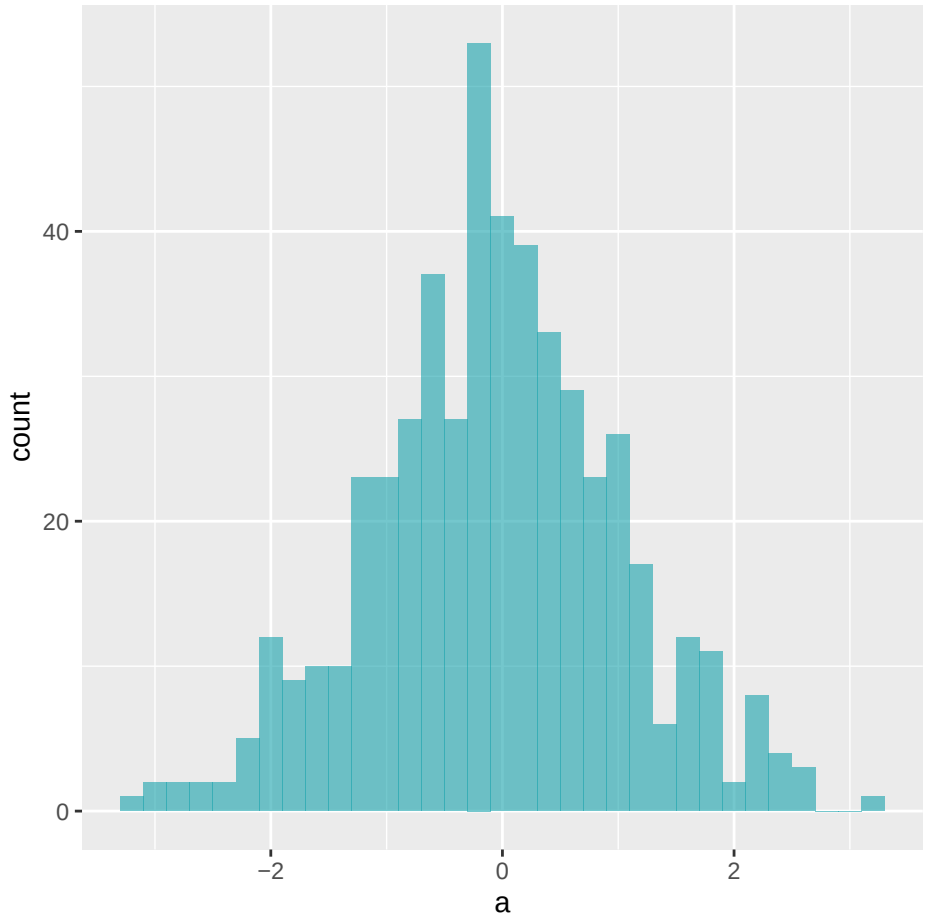
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
(p <- ggplot(mapping = aes(a)) + geom_histogram(binwidth = .2))
```

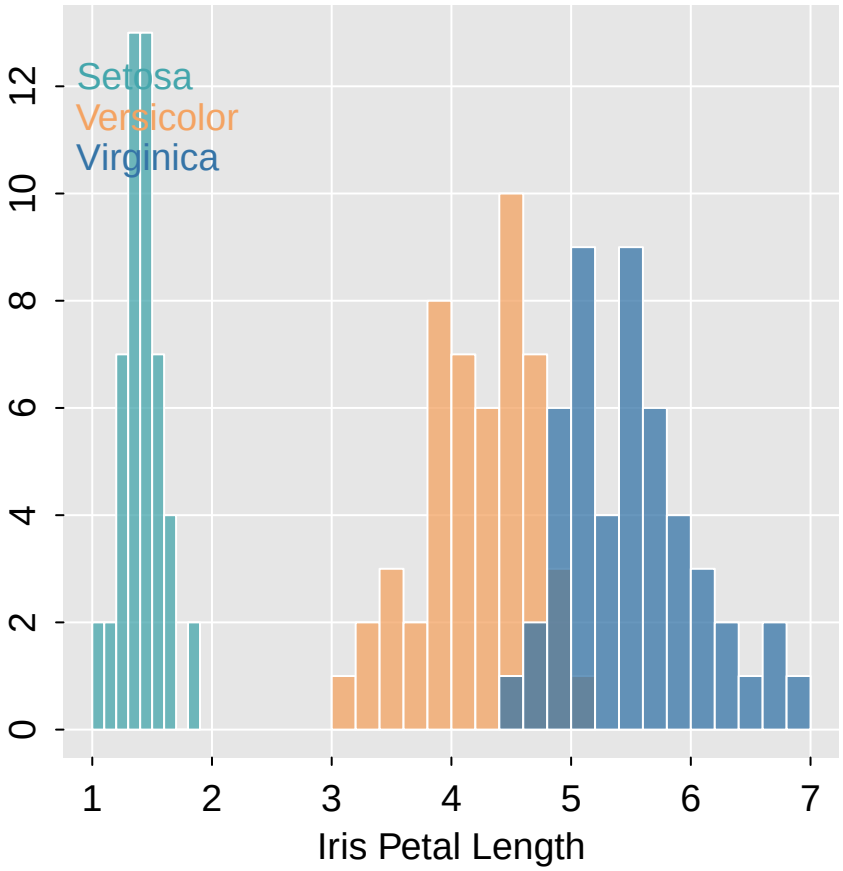



```
(p <- ggplot(mapping = aes(a)) +
    geom_histogram(binwidth = .2, fill = "#18A3AC99"))
```

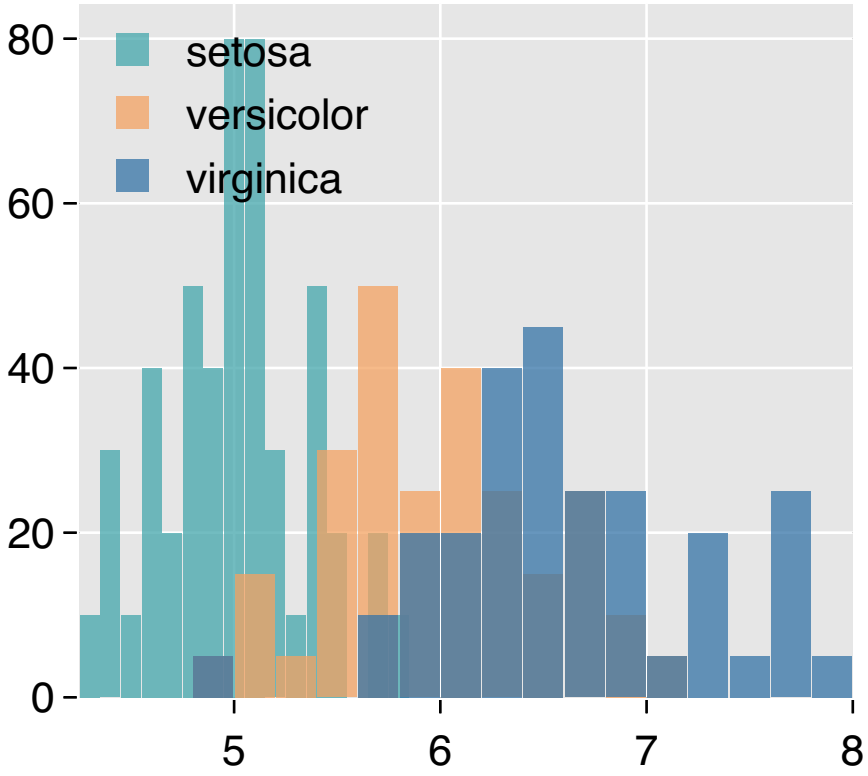


0.137.4.1 Grouped

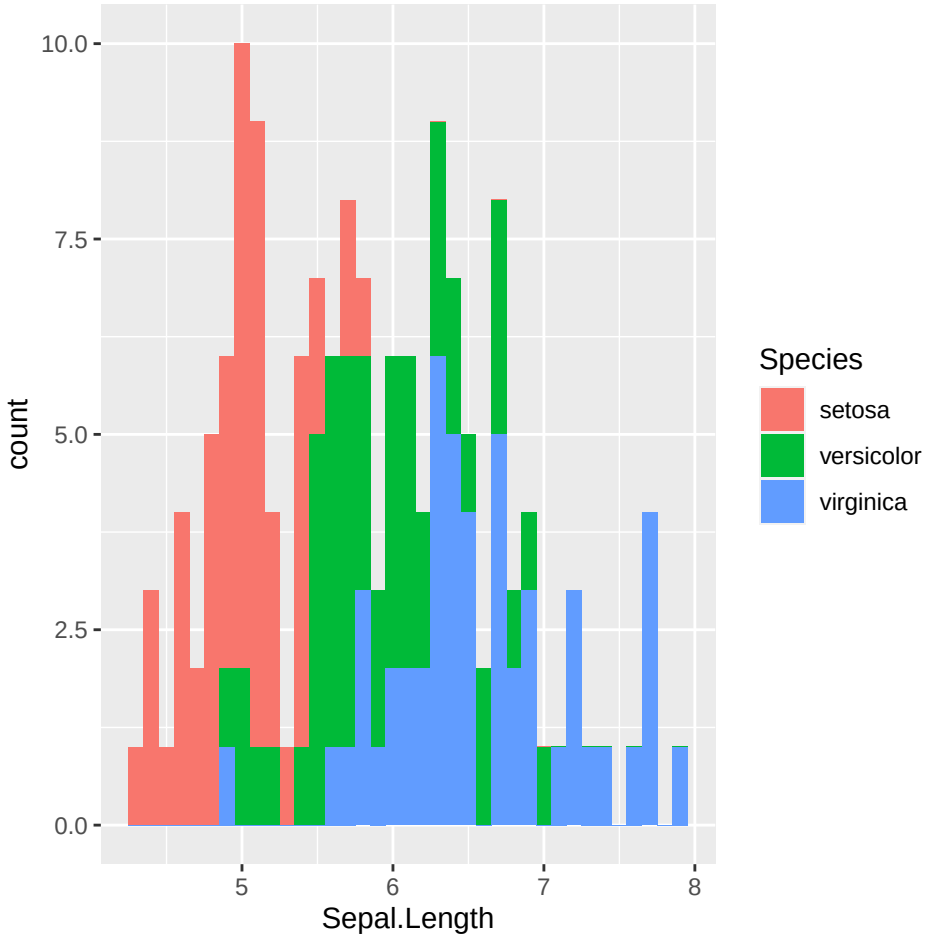
```
mplot3.x(iris$Petal.Length, 'h', group = iris$Species, hist.breaks = 10)
```




```
dplot3.x(iris$Sepal.Length, 'h', group = iris$Species)
```




```
ggplot(iris, aes(x = Sepal.Length, fill = Species)) +
  geom_histogram(binwidth = .1)
```

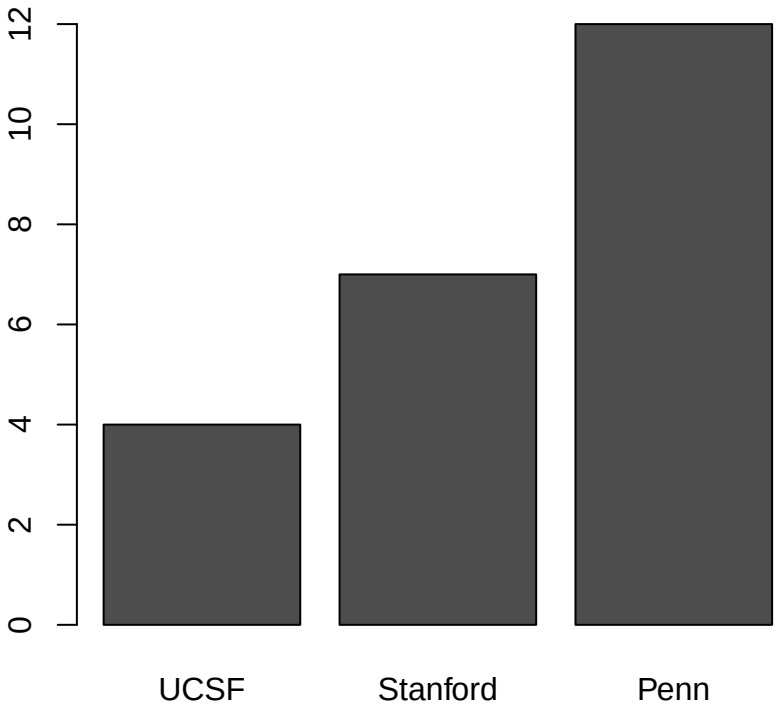


0.137.5 Barplot

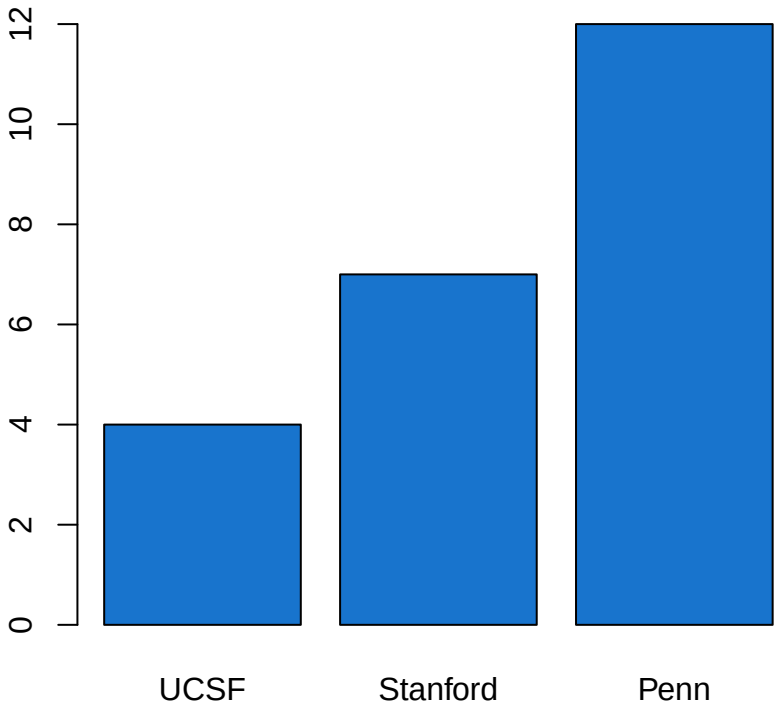
```
schools <- data.frame(UCSF = 4, Stanford = 7, Penn = 12)
```

0.137.6 base

```
barplot(as.matrix(schools))
```

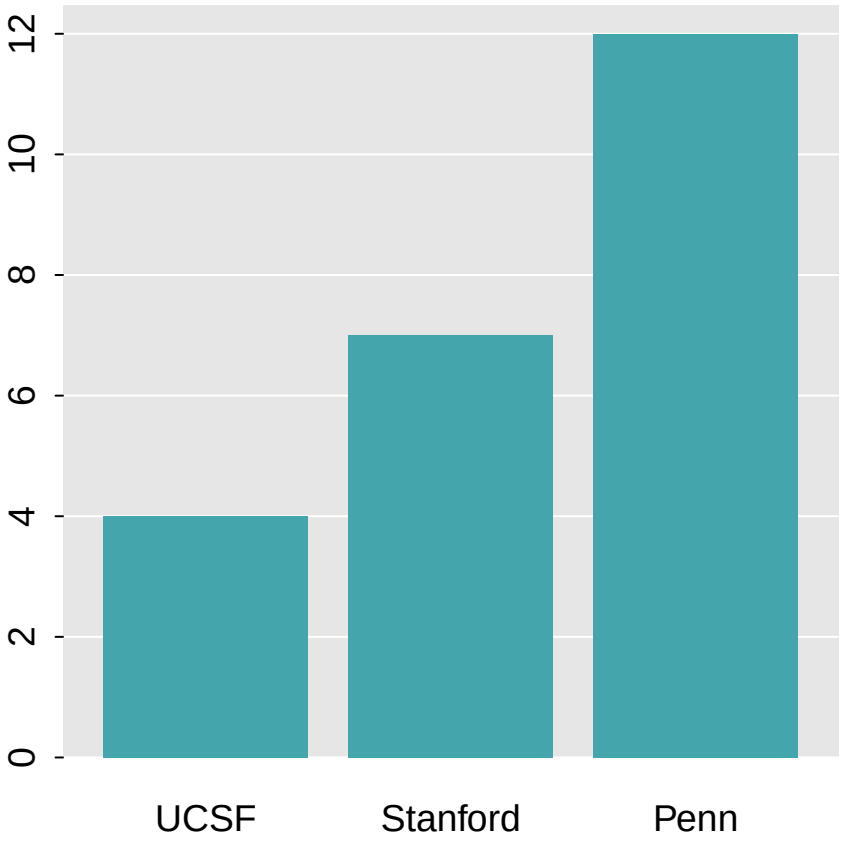



```
barplot(as.matrix(schools), col = "dodgerblue3")
```



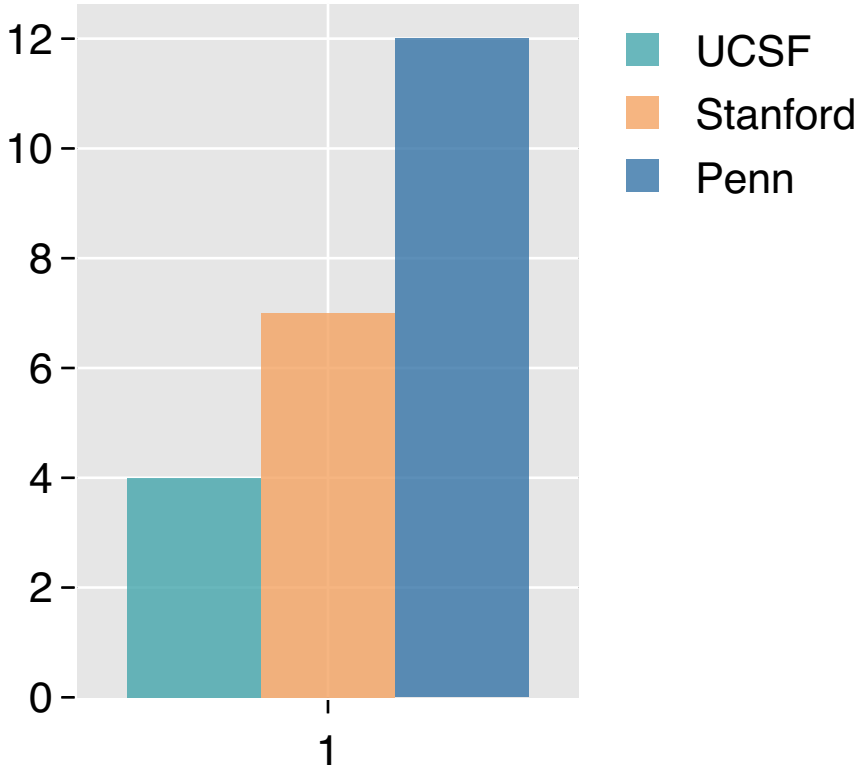
0.137.7 mplot3

```
mplot3.bar(schools)
```



0.137.8 dplot3

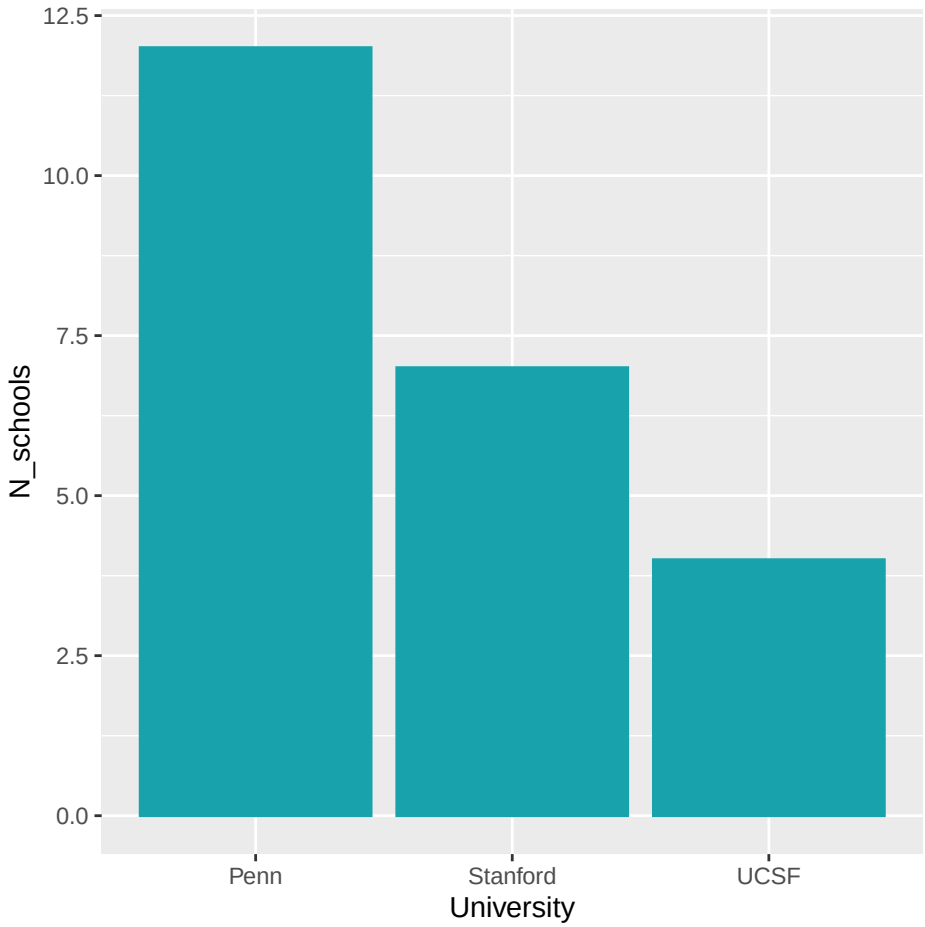
```
dplot3.bar(schools)
```



0.137.9 ggplot2

`ggplot` requires an explicit column in the data that define the categorical x-axis:

```
schools.df <- data.frame(University = colnames(schools),
                         N_schools = as.numeric(schools[1, ]))
ggplot(schools.df, aes(University, N_schools)) +
  geom_bar(stat = "identity", color = "#18A3AC", fill = "#18A3AC")
```

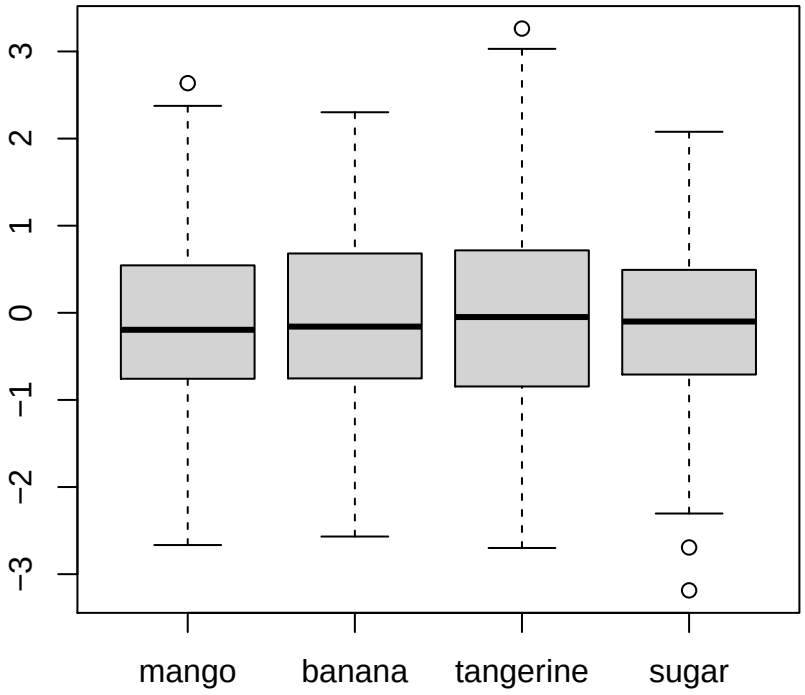


0.138 Box plot

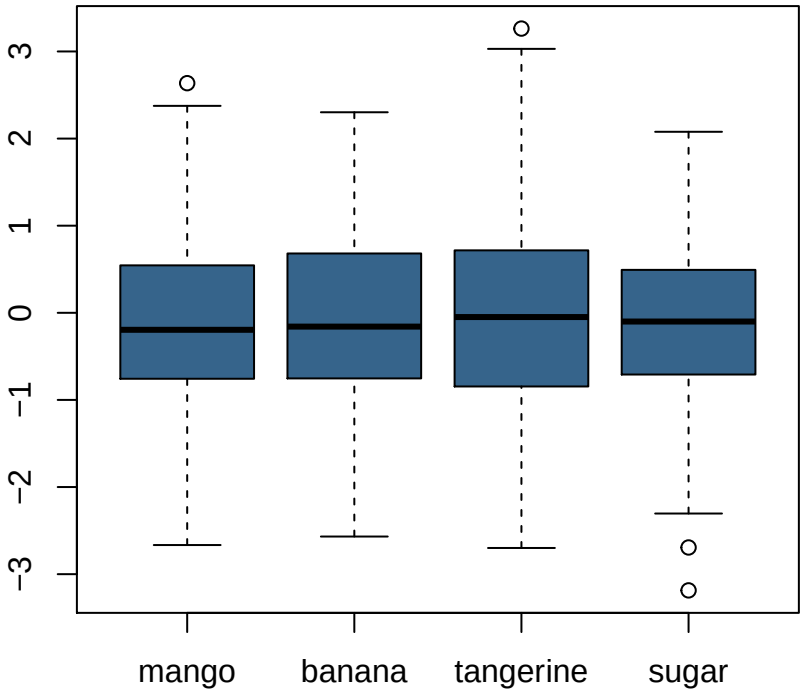
```
x <- rnormmat(200, 4, return.df = TRUE, seed = 2019)
colnames(x) <- c("mango", "banana", "tangerine", "sugar")
```

0.138.1 base

```
boxplot(x)
```

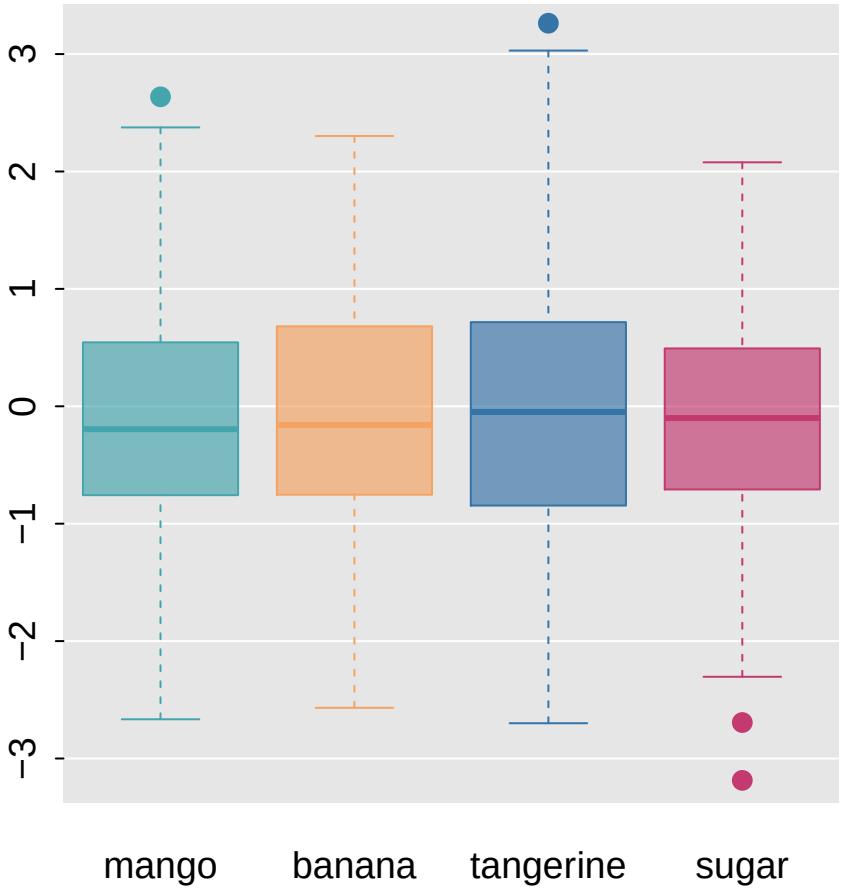



```
boxplot(x, col = "steelblue4")
```



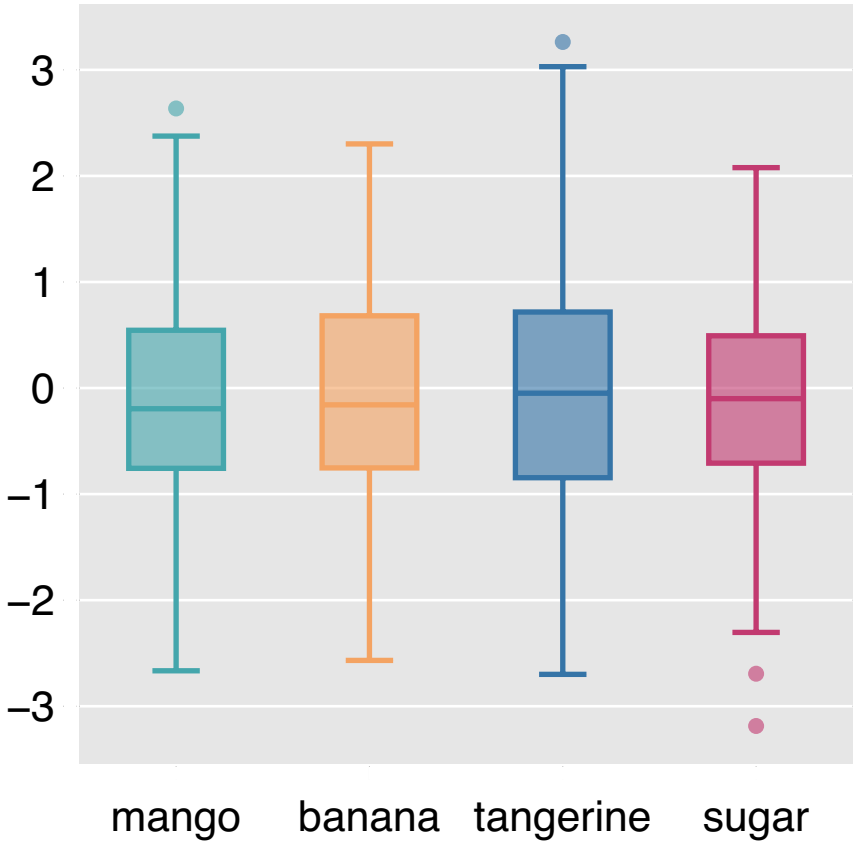
0.138.2 mplot3

```
mplot3.box(x)
```



0.138.3 dplot3

```
dplot3.box(x)
```

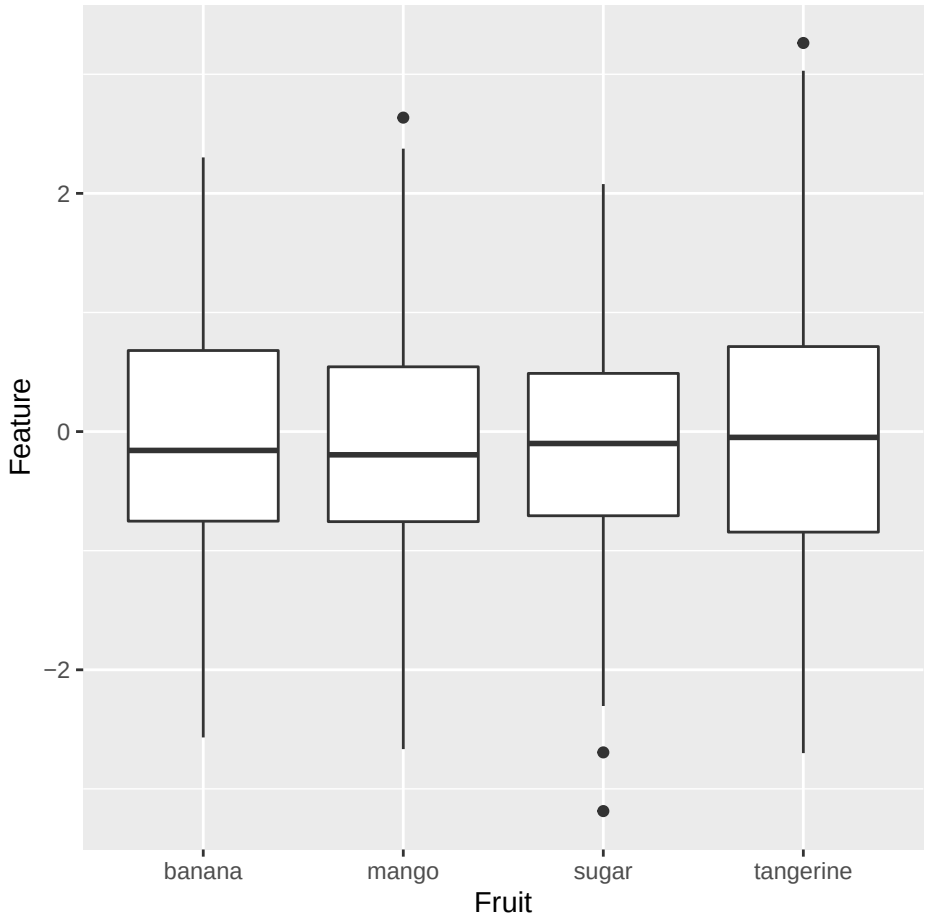


0.138.4 ggplot2

Again, ggplot requires an explicit categorical x-axis, which is this case means a conversion from wide to long dataset:

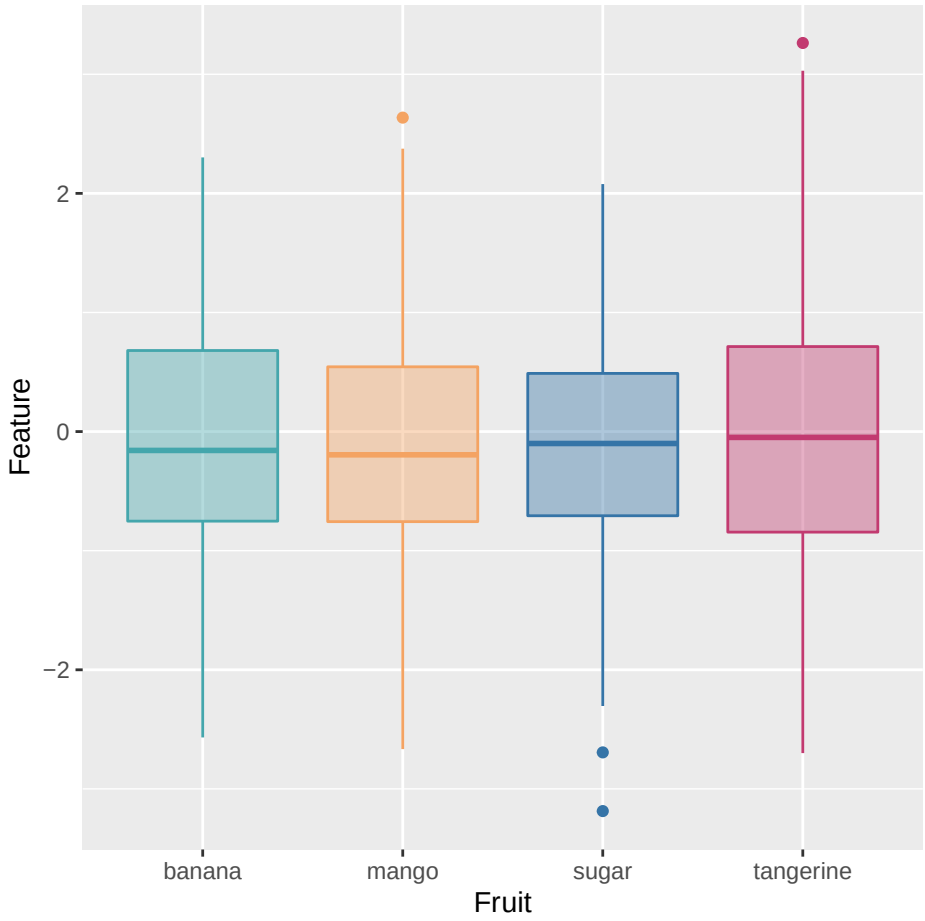
```
library(tidyr)
(x.long <- pivot_longer(x, 1:4, names_to = "Fruit", values_to = "Feature
```

```
(p <- ggplot(x.long, aes(Fruit, Feature)) + geom_boxplot())
```



Add some color:

```
(p <- ggplot(x.long, aes(Fruit, Feature)) +
   geom_boxplot(fill = c("#44A6AC66", "#F4A36266", "#3574A766", "#C23A7066"),
                colour = c("#44A6ACFF", "#F4A362FF", "#3574A7FF", "#C23A70FF")))
```



0.139 Heatmap

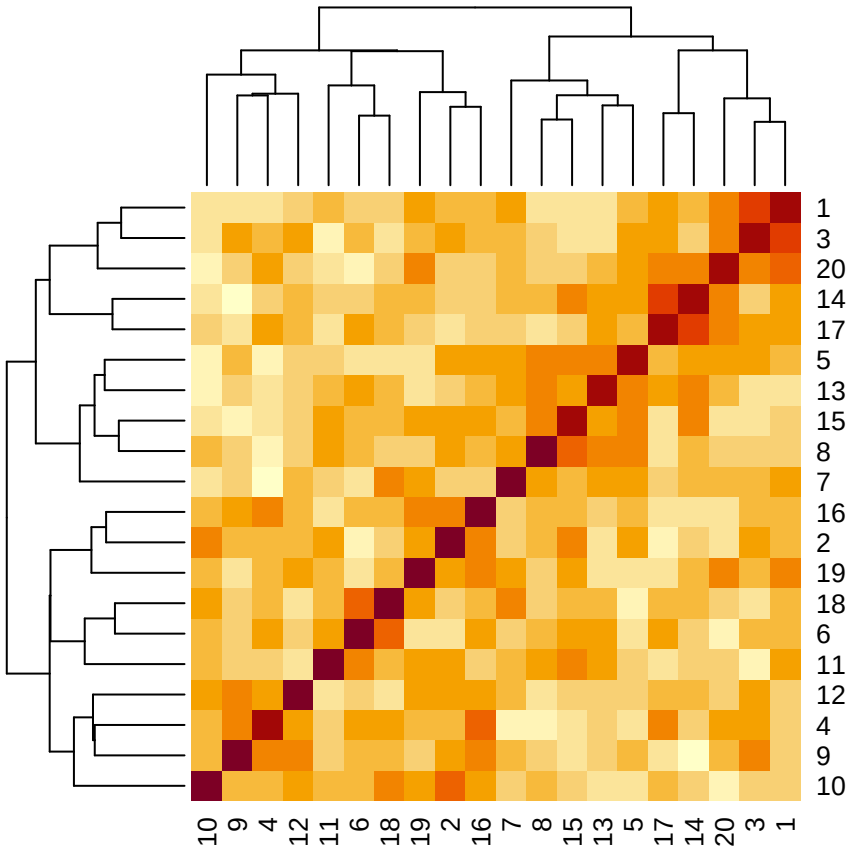
Let's create some synthetic correlation data:

```
x <- rnormmat(20, 20, seed = 2020)
x.cor <- cor(x)
```

0.139.1 base

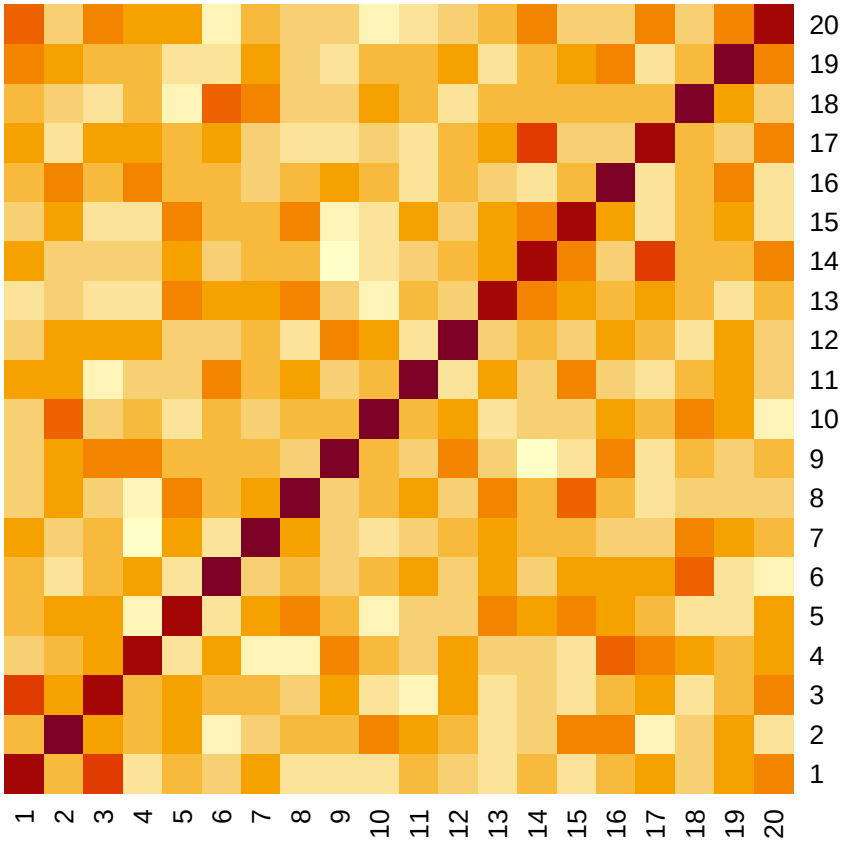
R has a great builtin heatmap function, which supports hierarchical clustering and plots the dendrogram in the margins by default:

```
heatmap(x.cor)
```



It may be a little surprising that clustering is on by default. To disable row and column dendrograms, set `Rowv` and `Colv` to `NA`:

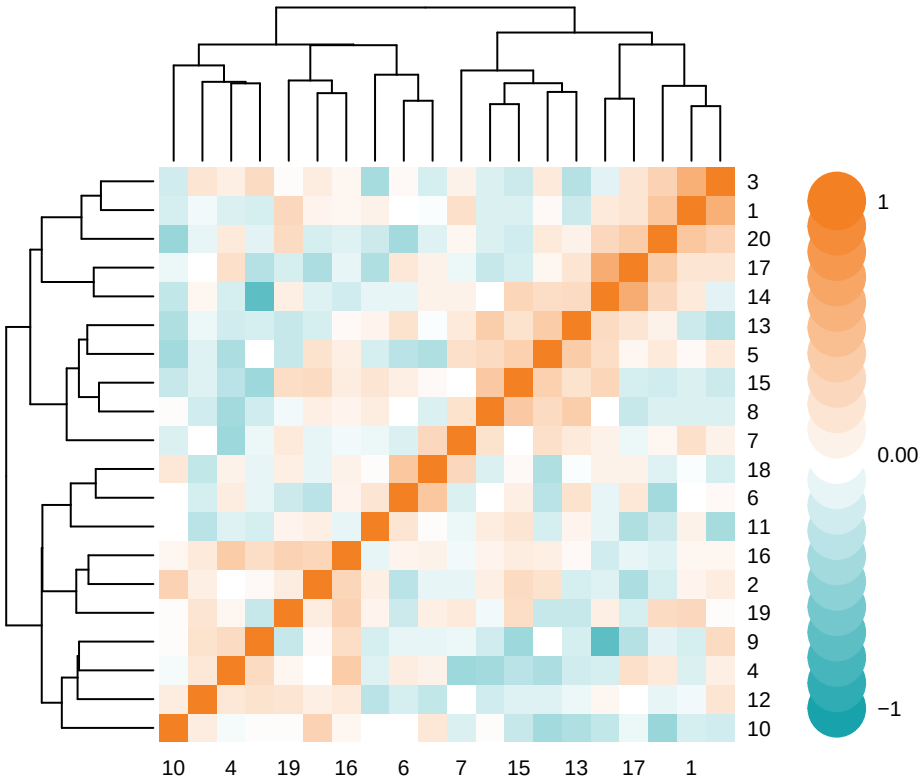
```
heatmap(x.cor, Rowv = NA, Colv = NA)
```



0.139.2 mplot3

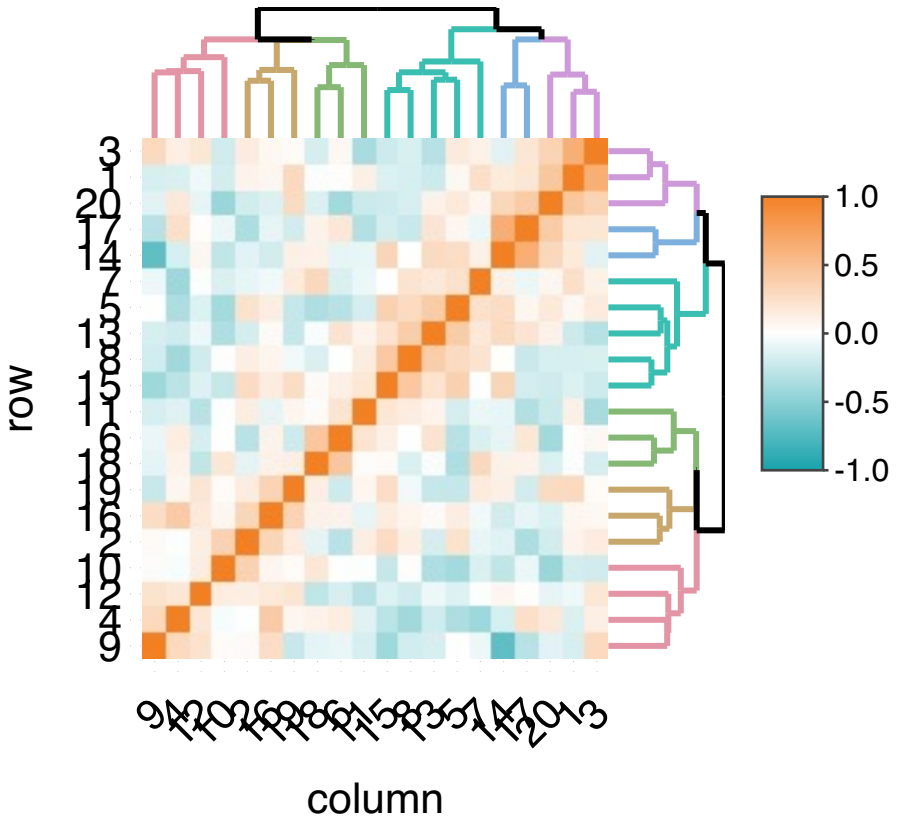
mplot3 adds a colorbar to the side of the heatmap. Notice there are 10 circles above and 10 circles below zero to represent 10% increments.

```
mplot3.heatmap(x.cor)
```



0.139.3 dplot3

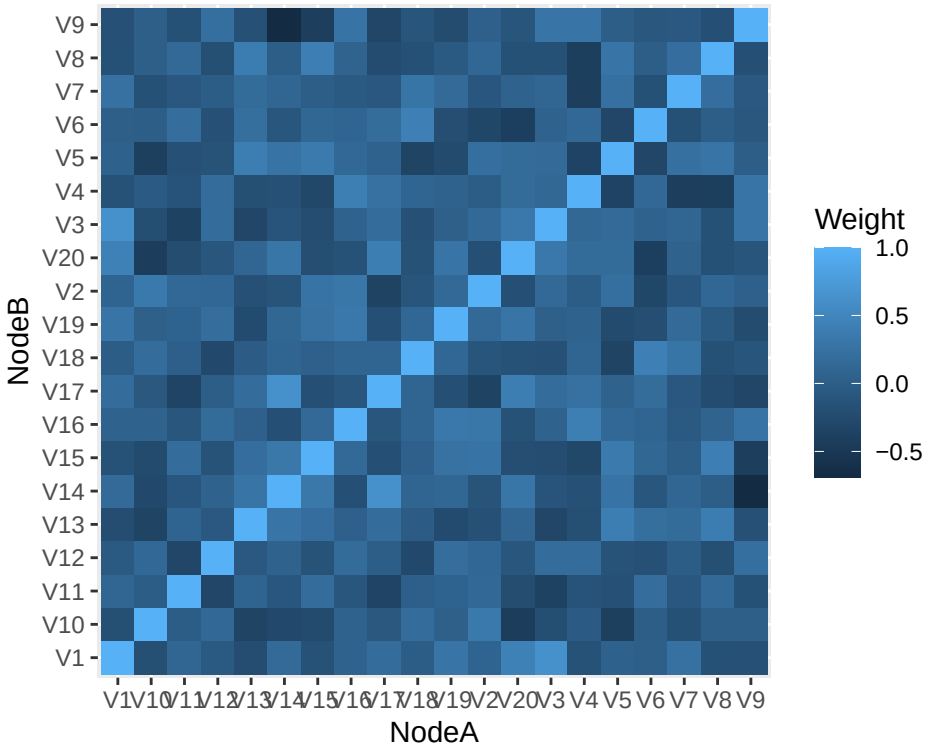
```
dplot3.heatmap(x.cor)
```



0.139.4 ggplot2

ggplot does not have a builtin heatmap function per se, but you can use `geom_tile` to build one. It also needs a data frame input in long form once again:

```
x.cor.dat <- as.data.frame(x.cor)
colnames(x.cor.dat) <- rownames(x.cor.dat) <- paste0("V", seq(20))
colnames(x.cor) <- rownames(x.cor) <- paste0("V", seq(20))
x.cor.long <- data.frame(NodeA = rownames(x.cor)[row(x.cor)],
                         NodeB = colnames(x.cor)[col(x.cor)],
                         Weight = c(x.cor))
(p <- ggplot(x.cor.long, aes(NodeA, NodeB, fill = Weight)) +
    geom_tile() + coord_equal())
```



0.140 Saving plots to file

0.140.1 base

You can save base graphics to disk using a number of different file formats. To do this, you have to:

- Open a graphic device - e.g. `pdf("path/to/xy_scatter.pdf")`
- Write to it - e.g. `plot(x, y)`
- Close graphic device - `dev.off()`

The following commands are used to open graphical devices that will save to a file of the corresponding type:

- `bmp(filename = "path/to/file", width = [in pixels], height = [in pixels])`
- `jpeg(filename = "path/to/file", width = [in pixels], height = [in pixels])`
- `png(filename = "path/to/file", width = [in pixels], height = [in pixels])`

- `tiff(filename = "path/to/file", width = [in pixels], height = [in pixels])`
- `svg(filename = "path/to/file", width = [in INCHES], height = [in INCHES]`
- `pdf(file = "path/to/file", width = [in INCHES], height = [in INCHES])`

Notice that when writing to a vector graphics format (svg and pdf), you defined width and height in inches, not pixels. Also, you specify `file` instead of `filename` in Notice the difference when writing to PDF: you define a `file` instead of a `filename`, and width and height are in INCHES, not pixels.
It is recommended to save plots in PDF format because it handles vector graphics therefore plots will scale, and it is easy to export to other graphics formats later on if needed.

```
pdf("~/Desktop/plot.pdf", width = 5, height = 5)
plot(iris$Sepal.Length, iris$Petal.Length,
     pch = 16,
     col = "#18A3AC66",
     cex = 1.8,
     bty = "n", # also try "l"
     xlab = "Sepal Length", ylab = "Petal Length")
dev.off()
```

Colors in R

```
library(rtemis)
```

```
 .:rtemis 0.8.1: Welcome, egenn
[x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
Documentation & vignettes: https://rtemis.lambdamd.org
```

Colors in R can be defined in many different ways:

- Using names: `colors()` gives all available options
- Using a hexadecimal[27] RGB[28] code in the form `#RRGGBBAA`, e.g. `#FF0000FF` for opaque red
- Using the `rgb(red, green, blue, alpha)` function (outputs a hex number)
- Using the `hsv(h, s, v, alpha)` function for the HSV color system[29] (also outputs a hex number)
- Using integers: these index the output of `palette()`, whose defaults can be changed by the user (e.g. `palette("cyan", "blue", "magenta", "red")`)

0.141 Color names

There is a long list of color names R understands, and can be listed using `colors()`. They can be passed directly as characters.
Shades of gray are provided as `gray0`/`grey0` (white) to `gray100`/`grey100` (black).
Absurdly wide PDFs with all built-in R colors, excluding the grays/greys, are available sorted alphabeticaly[30] and sorted by increasing Red and decreasing Green and Blue

[27] https://en.wikipedia.org/wiki/Hexadecimal
[28] https://en.wikipedia.org/wiki/RGB_color_model
[29] https://en.wikipedia.org/wiki/HSL_and_HSV
[30] https://rtemis.netlify.com/RColors.pdf

values[31]

0.142 Hexadecimal codes

Hexadecimal color codes are characters starting with the pound sign, followed by 4 pairs of hex codes representing Red, Green, Blue, and Alpha values. Since RGB values go from 0 to 255, hex goes from 00 to FF. You can convert decimal to hex using `as.hexmode`:

```
as.hexmode(0)
```

```
[1] "0"
```

```
as.hexmode(127)
```

```
[1] "7f"
```

```
as.hexmode(255)
```

```
[1] "ff"
```

The last two values for the alpha setting are optional: if not included, defaults to max (opaque)

0.143 RGB

```
rgb(0, 0, 1)
```

```
[1] "#0000FF"
```

Note the default `maxColorValue = 1`, set to 255 to use the usual RGB range of 0 to 255:

```
rgb(0, 0, 255, maxColorValue = 255)
```

```
[1] "#0000FF"
```

[31] https://rtemis.netlify.com/RColors_incRed_decBlueGreen.pdf

0.144 HSV

Color can also be parameterized using the hue, saturation, and value system (HSV[32]). Each range from 0 to 1.
Simplistically: Hue controls the color. Saturation 1 is max color and 0 is white. Value 1 is max color and 0 is black.

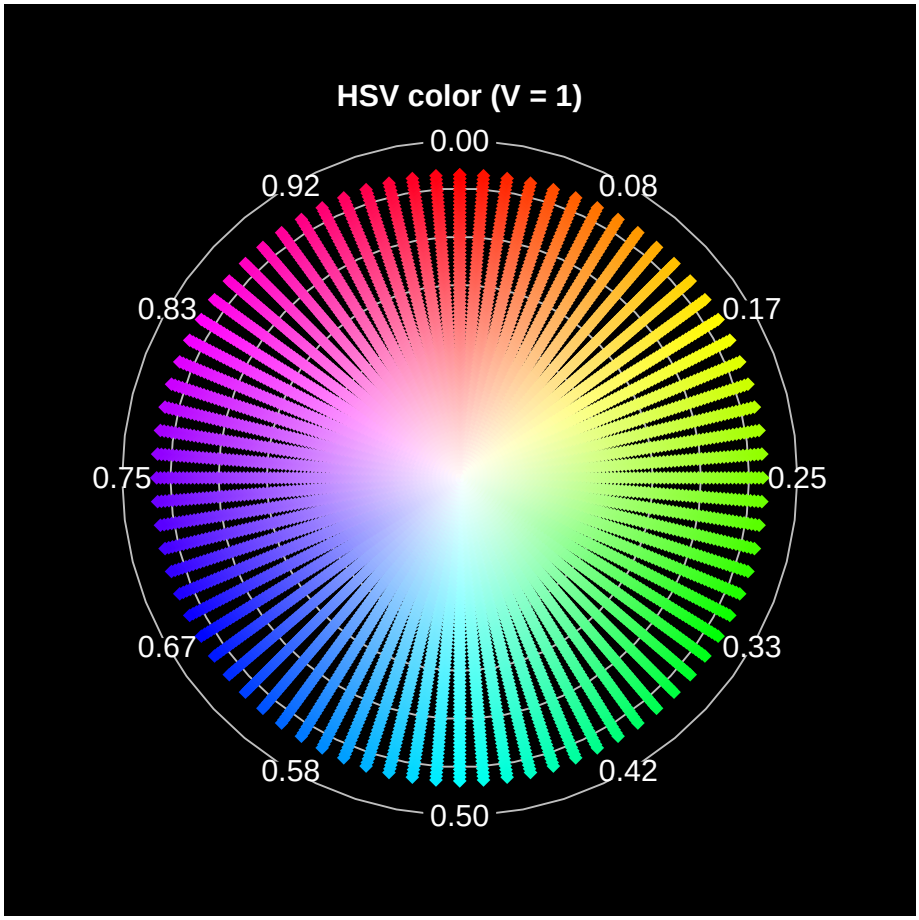
```
hsv(1, 1, 1)
```

```
[1] "#FF0000"
```

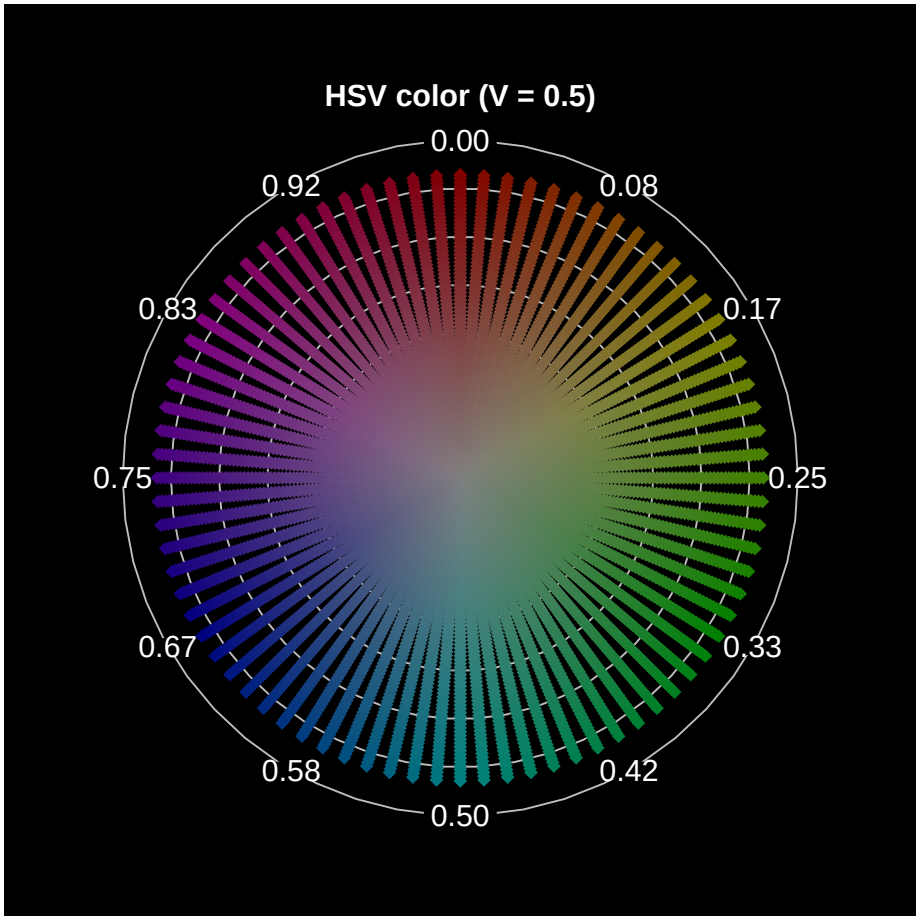
In the following plot, the values around the polar plot represent hue. Moving inwards to the center, saturation changes from 1 to 0.

```
mplot.hsv()
```

[32] https://en.wikipedia.org/wiki/HSL_and_HSV




```
mplot.hsv(v = .5)
```


HSV color (V = 0.5)
0.00
0.08
0.17
0.25
0.33
0.42
0.50
0.58
0.67
0.75
0.83
0.92

Timing & Profiling

Profiling your code involves timing the execution of different steps of a program. This is usually performed in order to identify bottlenecks that slow down the execution of your code and it helps you prioritize which parts to optimize. A

0.145 Time the execution of an expression with `system.time`

If you want to time how long it takes for an R expression to complete, you can use the base command `system.time`.
"elapsed" time is real time in seconds. "user" and "system" are time used by the CPU on different types of tasks (see `?proc.time`)

```
x <- rnorm(9999)
system.time({
    y <- vector("numeric", 9999)
    for (i in 1:9999) y[i] <- x[i]^3
})
```

```
   user  system elapsed
  0.008   0.001   0.020
```

```
system.time(x^3)
```

```
   user  system elapsed
  0.000   0.000   0.001
```

You can use `replicate()` to get a measure of time over multiple executions and average it:

```
library(mgcv)
```

```
Loading required package: nlme
```

```
This is mgcv 1.8-33. For overview type 'help("mgcv-package")'.
```

```
library(glmnet)
```

```
Loading required package: Matrix
```

```
Loaded glmnet 4.0-2
```

```
set.seed(2020)
x <- replicate(100, rnorm(5000))
y <- x[, 1]^2 + x[, 5]^3 + 12 + rnorm(5000)
dat <- data.frame(x, y)
fit.glm <- function(dat) mod <- glm(y ~ x, family = gaussian, data = dat
fit.gam <- function(dat) mod <- gam(y ~ x, family = gaussian, data = dat

system.time(replicate(1000, fit.glm))
```

```
   user  system elapsed
  0.002   0.000   0.005
```

```
system.time(replicate(1000, fit.gam))
```

```
   user  system elapsed
  0.002   0.000   0.003
```

0.146 Compare execution times of different expressions with `microbenchmark()`

`microbenchmark()` allows you to time the execution of multiple expressions with sub-millisecond accuracy. It will execute each command a number of times as defined by the `times` argument (default = 100), and output statistics of execution time per expression in nanoseconds. Using `plot()` on the output produces a boxplot comparing the time distributions.

```
library(microbenchmark)
```

To start, we compare two very simple and fast operations, using base and dplyr to add two columns of 1000 integers:

```
dat <- as.data.frame(matrix(1:2000, 1000))
dim(dat)
```

```
[1] 1000    2
```

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following object is masked from 'package:nlme':

    collapse

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
add2k <- microbenchmark(
  base = dat$V1 + dat$V2,
  dplyr = mutate(dat, a = V1 + V2))
```

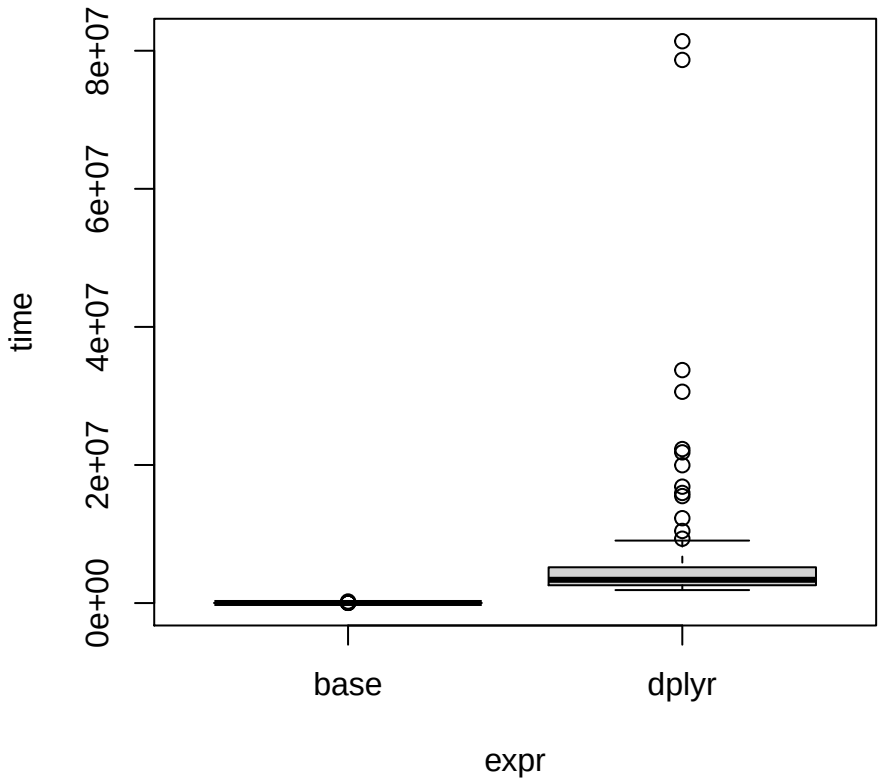
You can print microbenchmark's output:

```
add2k
```

```
Unit: microseconds
  expr    min     lq     mean  median      uq     max neval
  base   5.276  10.404  15.77369  11.524  16.2335  222.355  100
 dplyr 1879.996 2579.966 6785.43285 3371.792 5183.1905 81376.689  100
```

and plot it:

```
plot(add2k)
```



Now let's use the **nycflights13** dataset which includes data on 336776 flights that departed from any of the three NY area airports in 2013. Because the data comes as a tibble[33], we shall perform all operations on the tibble and a data.frame of the same data to compare.

```
library(nycflights13)
class(flights)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(flights)
```

```
[1] 336776     19
```

```
flightsDF <- as.data.frame(flights)
```

[33] https://tibble.tidyverse.org/

Compare performance base R vs. dplyr in calculating mean arrival delay by carrier using either a data.frame or a tibble:

```
dbc <- microbenchmark(
  df_aggregate = aggregate(flightsDF$arr_delay, by = list(flightsDF$carrier), me
  tb_aggregate = aggregate(flights$arr_delay, by = list(flights$carrier), mean,
  df_tapply = tapply(flightsDF$arr_delay, flightsDF$carrier, mean, na.rm = TRUE)
  tb_tapply = tapply(flights$arr_delay, flights$carrier, mean, na.rm = TRUE),
  df_dplyr = flightsDF %>% group_by(carrier) %>% summarize(mean(arr_delay, na.rm
  tb_dplyr = flights %>% group_by(carrier) %>% summarize(mean(arr_delay, na.rm =
```

```
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
```

```
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
```

```
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
```

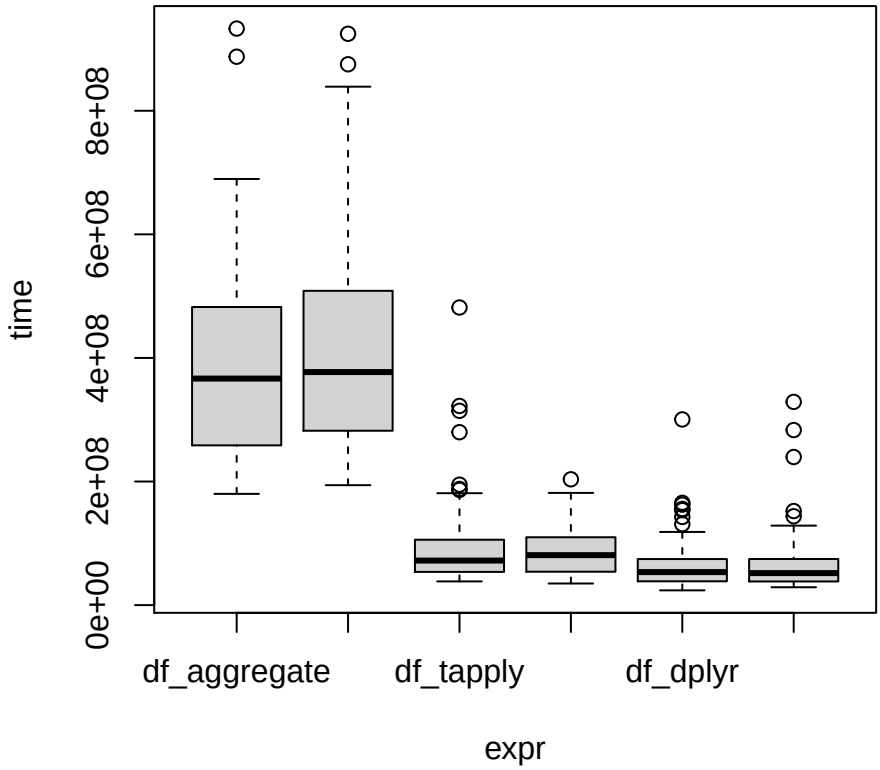
```
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
```

```
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
`summarise()` ungrouping output (override with `.groups` argument)
```

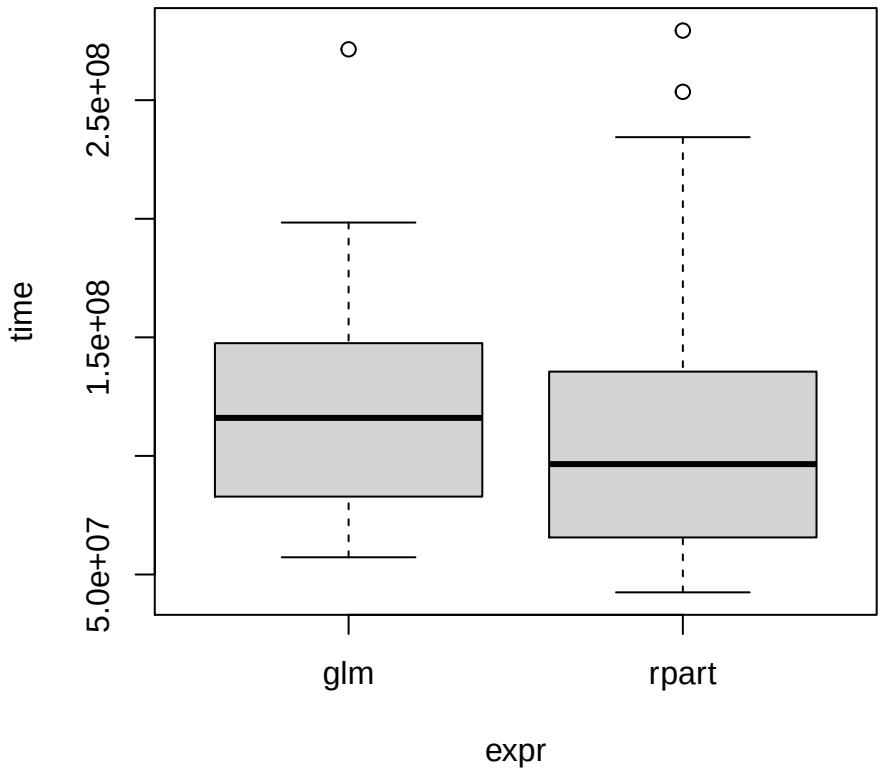
```
dbc
```

```
Unit: milliseconds
         expr       min        lq      mean    median        uq      max neval
 df_aggregate 180.01963 258.52203 390.49698 366.60796 482.44184 933.0200   100
 tb_aggregate 194.09733 282.16752 415.94577 377.13185 508.63546 924.6015   100
    df_tapply  38.35571  53.70661  91.36301  72.09985 105.78595 481.6390   100
    tb_tapply  34.97787  54.00179  87.36437  80.84533 109.81116 203.5741   100
     df_dplyr  23.88064  38.50721  63.63290  53.41112  74.56493 300.4602   100
     tb_dplyr  28.89073  38.25630  65.57653  51.82650  74.60587 328.8792   100
```

```
plot(dbc)
```




```
library(rpart)
data(Sonar, package = "mlbench")

glmVSrpart <- microbenchmark(
  glm = glm(Class ~ ., family = "binomial", Sonar),
  rpart = rpart(Class ~ ., Sonar, method = "class"),
  times = 50)

plot(glmVSrpart)
```



0.147 Profile a function with `profvis()`

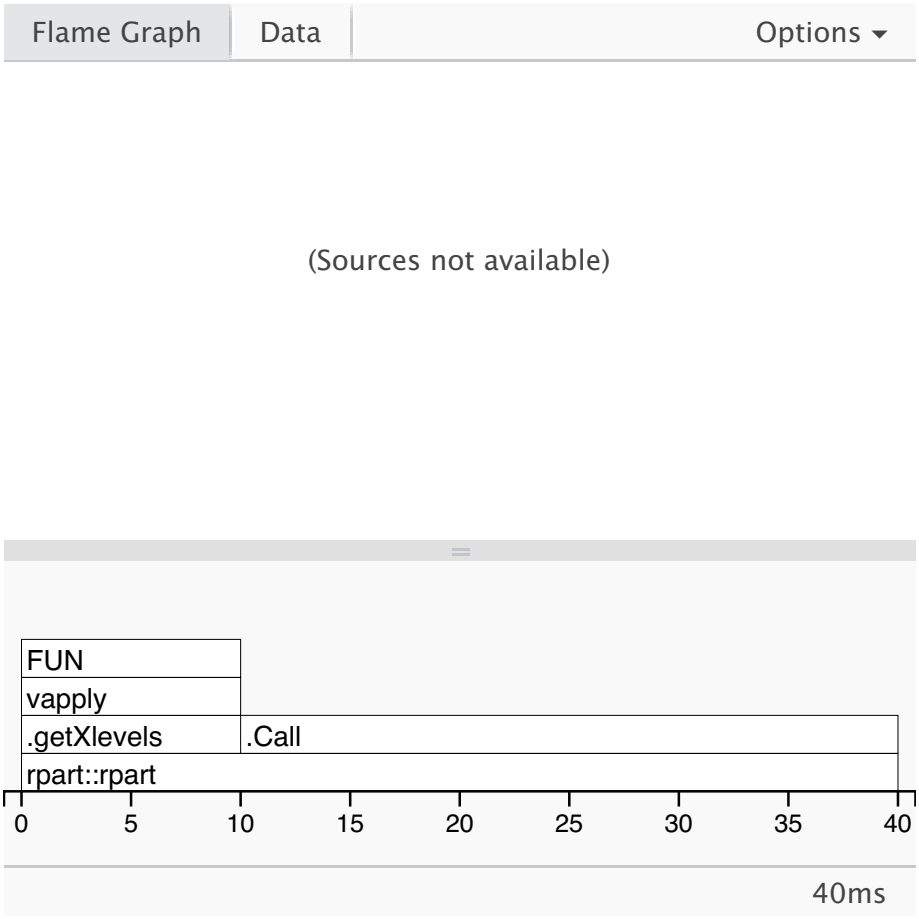
`profvis` provides an interactive output to visualize how much time is spent in different calls within an algorithm.

```
library(profvis)
library(rtemis)
```

```
  .:rtemis 0.8.0: Welcome, egenn
 [x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
 Documentation & vignettes: https://rtemis.lambdamd.org
```

```
data(Sonar, package = 'mlbench')
```

```
profvis(rpart::rpart(Class ~ ., Sonar))
```


Flame Graph
Data
Options
(Sources not available)
FUN
vapply
.getXlevels
.Call
rpart::rpart
0
5
10
15
20
25
30
35
40
40ms

Optimization with `optim()`

```
 .:rtemis 0.8.0: Welcome, egenn
[x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
Documentation & vignettes: https://rtemis.lambdamd.org
```

R provides a general purpose optimization tool, `optim()`. You can use it to estimate parameters that minimize any defined function.
Supervised and unsupervised learning involves defining a loss function to minimize or an objective function to minimize or maximize.
To learn how `optim()` works, let's write a simple function that returns linear coefficients by minimizing squared error.

0.148 Data

```
set.seed(2020)
x <- sapply(seq(10), function(i) rnorm(500))
y <- 12 + 1.5 * x[, 3] + 3.2 * x[, 7] + .5 * x[, 9] + rnorm(500)
```

0.149 GLM (`glm`, `s.GLM`)

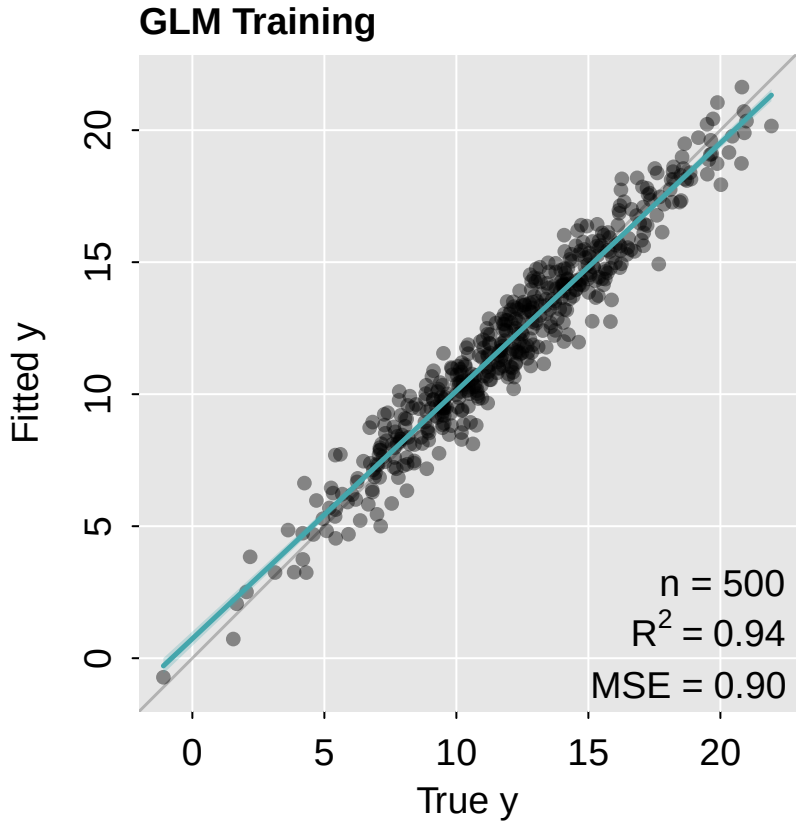
```
yx.glm <- glm(y ~ x)
summary(yx.glm)
```

```
Call:
glm(formula = y ~ x)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
```

```
-2.38739  -0.67391   0.00312   0.65531   3.08524

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 11.979070   0.043252 276.962   <2e-16 ***
x1           0.061798   0.040916   1.510   0.1316
x2          -0.003873   0.043271  -0.090   0.9287
x3           1.488113   0.042476  35.034   <2e-16 ***
x4           0.031115   0.044015   0.707   0.4800
x5           0.034217   0.043664   0.784   0.4336
x6           0.034716   0.042189   0.823   0.4110
x7           3.183398   0.040605  78.399   <2e-16 ***
x8          -0.034252   0.043141  -0.794   0.4276
x9           0.541219   0.046550  11.627   <2e-16 ***
x10          0.087120   0.044000   1.980   0.0483 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 0.9207315)

    Null deviance: 7339.42  on 499  degrees of freedom
Residual deviance:  450.24  on 489  degrees of freedom
AIC: 1390.5

Number of Fisher Scoring iterations: 2
```

Or, using *rtemis*:

```
mod.glm <- s.GLM(x, y)
```




```
summary(mod.glm$mod)
```

```
Call:
glm(formula = .formula, family = family, data = df.train, weights = .weights,
    na.action = na.action)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.38739  -0.67391   0.00312   0.65531   3.08524

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 11.979070   0.043252 276.962   <2e-16 ***
V1           0.061798   0.040916   1.510   0.1316
V2          -0.003873   0.043271  -0.090   0.9287
V3           1.488113   0.042476  35.034   <2e-16 ***
V4           0.031115   0.044015   0.707   0.4800
V5           0.034217   0.043664   0.784   0.4336
```

```
V6           0.034716   0.042189   0.823   0.4110
V7           3.183398   0.040605  78.399   <2e-16 ***
V8          -0.034252   0.043141  -0.794   0.4276
V9           0.541219   0.046550  11.627   <2e-16 ***
V10          0.087120   0.044000   1.980   0.0483 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 0.9207315)

    Null deviance: 7339.42  on 499  degrees of freedom
Residual deviance:  450.24  on 489  degrees of freedom
AIC: 1390.5

Number of Fisher Scoring iterations: 2
```

0.150 optim

Basic usage of `optim` to find values of parameters that minimize a function:

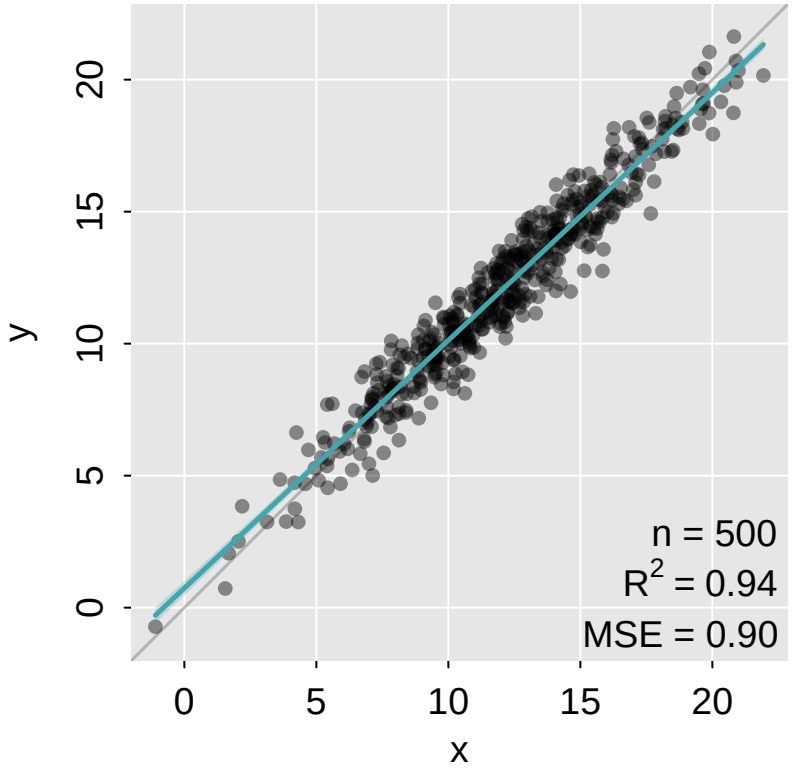
- Define a list of initial parameter values
- Define a loss function whose first argument is the above list of initial parameter values
- Pass parameter list and objective function to `optim`

In the following example, we wrap these three steps in a function called `linearcoeffs`, which will output the linear coefficients that minimize squared error, given a matrix/data.frame of features `x` and an outcome `y`. We also specify the optimization method to be used (See `?base::optim` for details):

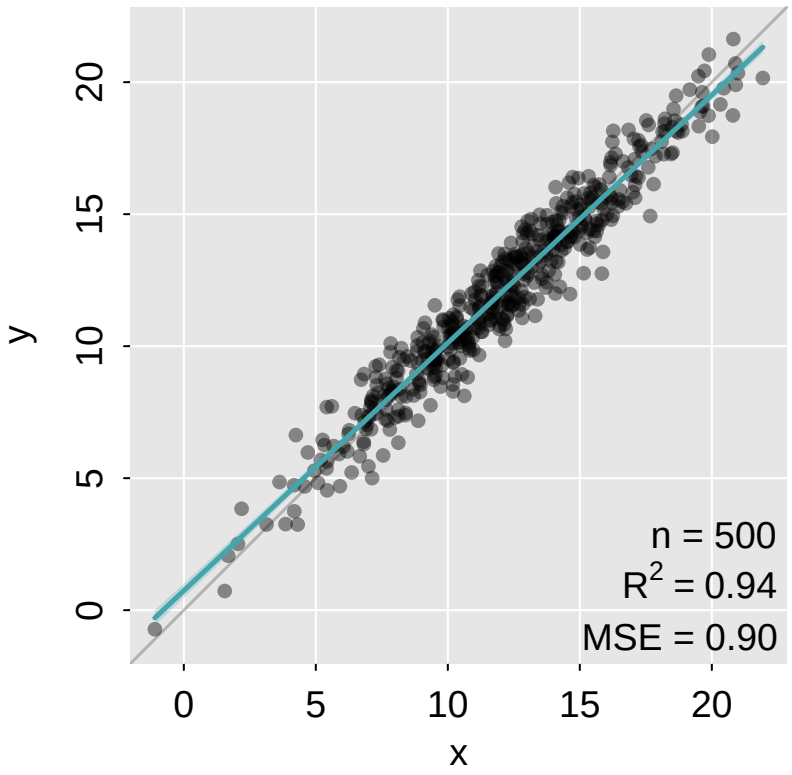
```
linearcoeffs <- function(x, y, method = "BFGS") {

  # 1. List of initial parameter values
  params <- as.list(c(mean(y), rep(0, NCOL(x))))
  names(params) <- c("Intercept", paste0("Coefficient", seq(NCOL(x))))

  # 2. Loss function: first argument is parameter list
  loss <- function(params, x, y) {
    estimated <- c(params[[1]] + x %*% unlist(params[-1]))
    mean((y - estimated)^2)
  }

  # 3. optim!
  coeffs <- optim(params, loss, x = x, y = y, method = method)
```

```
  # The values that minimize the loss function are stored in $par
  coeffs$par
}
```

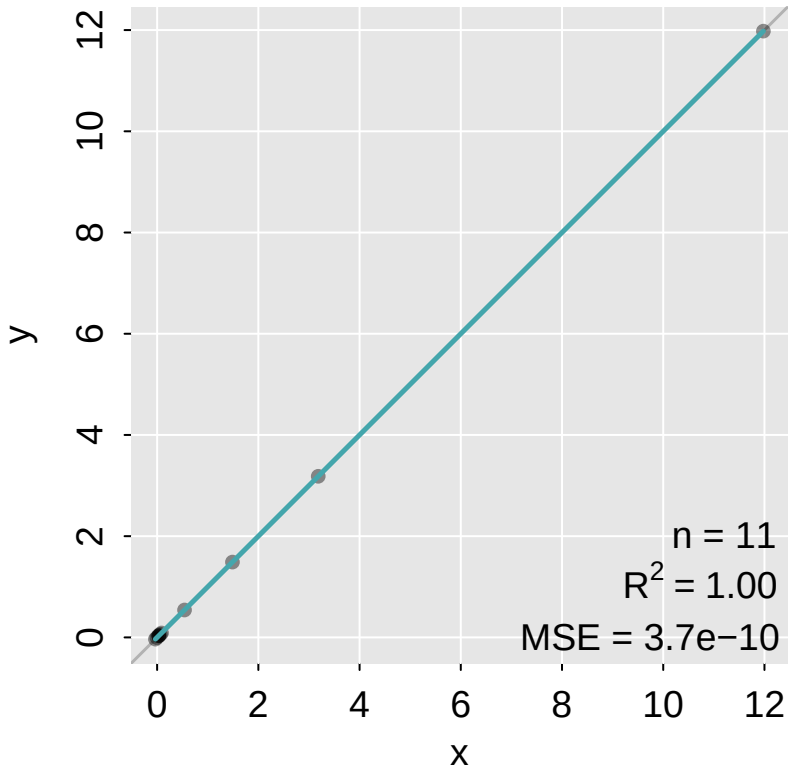
```
coeffs.optim <- linearcoeffs(x, y)
estimated.optim <- cbind(1, x) %*% coeffs.optim
mplot3.fit(y, estimated.optim)
```




```
coeffs.glm <- mod.glm$mod$coefficients
estimated.glm <- cbind(1, x) %*% coeffs.glm
mplot3.fit(y, estimated.glm)
```




```
mplot3.fit(coeffs.glm, coeffs.optim)
```


y
x
0
2
4
6
8
10
12
n = 11
R2 = 1.00
MSE = 3.7e−10

Resampling

```
library(rtemis)
```

```
 .:rtemis 0.8.0: Welcome, egenn
[x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
Documentation & vignettes: https://rtemis.lambdamd.org
```

Resampling refers to a collection of techniques for selecting cases from a sample. It is central to many machine learning algorithms and pipelines. The two core uses of resampling are model selection (tuning) and assessment. By convention, we use the terms training and validation sets when refering to model selection, and training and testing sets when refering to model assessment. The terminology is unfortunately not intuitive and has led to much confusion. Some people reverse the terms, but we use the terms training, validation, and testing as they are used in the Elements of Statistical Learning[34] (p. 222, Second edition, 12th printing)

0.151 Model Selection and Assessment

1. **Model Selection** aka **Hyperparameter tuning**
   Resamples of the training set are drawn. For each resample, a combination of hyperparameters is used to train a model. The mean validation-set error across resamples is calculated. The combination of hyperparameters with the minimum loss on average across validation-set resamples is selected to train the full training sample.
2. **Model assessment**
   Resamples of the full sample is split into multiple training - testing sets. A model is trained on each training set and its performance assessed on the corresponding test set. Model performance is averaged across all test sets.

Nested resampling or **nested crossvalidation** is the procedure where 1. and 2. are nested so that hyperparameter tuning (resampling of the training set) is performed

[34] https://web.stanford.edu/~hastie/ElemStatLearn/

within each of multiple training resamples and performance is tested in each corresponding test set. [elevate] performs automatic nested resampling and is one of the core supervised learning functions in **rtemis**.

0.152 The resample function

The `resample` function is responsible for all resampling in **rtemis**.

```
x <- rnorm(500)
res <- resample(x)
```

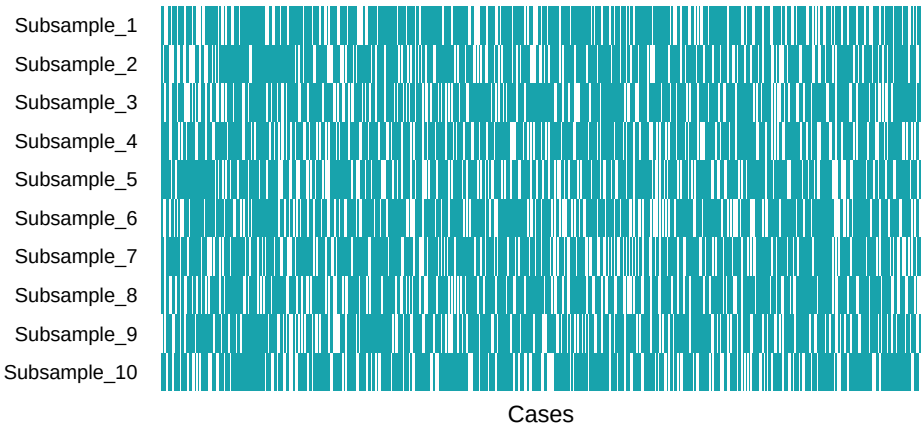
```
class(res)
```

```
[1] "resample" "list"
```

It outputs a list which is an S3 object of class `resample`, with print and plot methods.

```
res
```

```
plot(res)
```



The teal-colored lines represent the training cases selected for each resample, the white are testing cases (held out).

`resample` supports 5 types of resampling:

1. **k-fold crossvalidation (Stratified)**
   You split the cases into k sets (folds). Each set is used once as the validation or testing set. This means each cases is left out exactly once and there is no overlap

between different validation/test sets. In **rtemis**, the folds are also stratified by default on the outcome unless otherwise chosen. Stratification tries to maintain the full sample's distribution in both training and left-out sets. This is crucial for non-normally distributed continuous outcomes or imbalanced datasets. 10 is a common value for k, called 10-fold. Note that the size of the training and left-out sets depends on the sample size.

```
res.10fold <- resample(x, 10, "kfold")
```

2. **Stratified subsampling**
   Draw `n.resamples` stratified samples from the data given a certain probability (`train.p`) that each case belongs to the training set. Since you are randomly sampling from the full sample each time, there will be overlap in the test set cases, but you control the training-to-testing ratio and number of resamples independently, unlike in k-fold resampling.

```
res.25ss <- resample(x, 25, "strat.sub")
```

3. **Bootstrap**
   The bootstrap[35]: random sampling with replacement. Since cases are replicated, you should use bootstrap as the outer resampler if you will also have inner resampling for tuning, since the same case may end up in both training and validation sets.

```
res.100boot <- resample(x, 100, "bootstrap")
```

4. **Stratified Bootstrap**
   This is stratified subsampling with random replication of cases to match the length of the original sample. Same as the bootstrap, do not use if you will be further resampling each resample.

```
res.100sboot <- resample(x, 100, "strat.boot")
```

5. **Leave-One-Out-Crossvalidation (LOOCV)**
   This is k-fold crossvalidation where $k = N$, where N is number of data points/cases in the whole sample. It has been included for experimentation and completenes, but it is not recommended either for model selection or assessment over the other resampling methods.

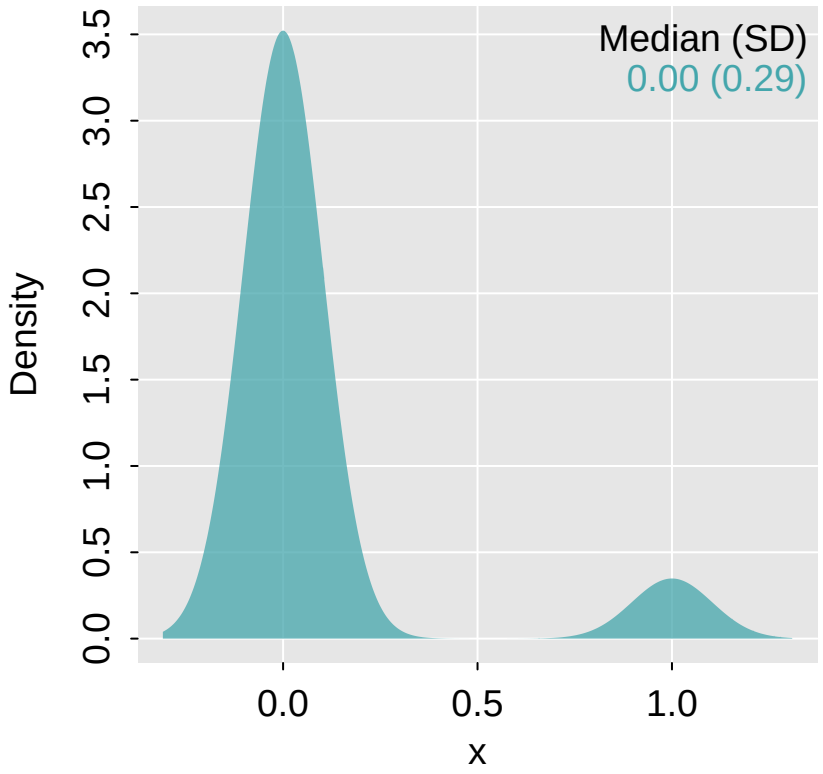
```
res.loocv <- resample(x, resampler = "loocv")
```

[35] `https://en.wikipedia.org/wiki/Bootstrapping_(statistics)`

0.153 Example: Stratified vs random sampling in a binomial distribution

Imagine `y` is the outcome of interest where events occur with a probability of .1 - a common scenario in many fields.

```
set.seed(2020)
x <- rbinom(100, 1, .1)
mplot3.x(x)
```




```
freq <- table(x)
prob <- freq[2] / sum(freq)
res.nonstrat <- lapply(seq(10), function(i) sample(seq(x), .75*length(x)
res.strat <- resample(x)
```

```
prob.nonstrat <- sapply(seq(10), function(i) {
  freq <- table(x[res.nonstrat[[i]]])
```

```
  freq[2]/sum(freq)
})
prob.strat <- sapply(seq(10), function(i) {
  freq <- table(x[res.strat[[i]]])
  freq[2]/sum(freq)
})
prob.nonstrat
```

```
      1       1       1       1       1       1       1
0.09333333 0.08000000 0.08000000 0.06666667 0.06666667 0.10666667 0.10666667
      1       1       1
0.10666667 0.09333333 0.08000000
```

```
sd(prob.nonstrat)
```

```
[1] 0.0156505
```

```
prob.strat
```

```
      1       1       1       1       1       1       1
0.08108108 0.08108108 0.08108108 0.08108108 0.08108108 0.08108108 0.08108108
      1       1       1
0.08108108 0.08108108 0.08108108
```

```
sd(prob.strat)
```

```
[1] 0
```

As expected, the random sampling resulted in different event probability in each resample, whereas stratified subsampling maintained a constant probability across resamples.

Introduction to the GLM

0.154 Generalized Linear Model (GLM)

The Generalized Linear Model is one of the most common and important models in statistics.
Let's look at an example using the GLM for regression. We will use the `mtcars` builtin dataset to predict horsepower (`hp`) of 32 cars from 10 other features:

```
str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
mod <- glm(hp ~ ., family = "gaussian", data = mtcars)
mod
```

```
Call:  glm(formula = hp ~ ., family = "gaussian", data = mtcars)

Coefficients:
(Intercept)          mpg          cyl         disp         drat           wt
     79.048       -2.063        8.204        0.439       -4.619      -27.660
       qsec           vs           am         gear         carb
     -1.784       25.813        9.486        7.216       18.749
```

```
Degrees of Freedom: 31 Total (i.e. Null);  21 Residual
Null Deviance:      145700
Residual Deviance: 14160    AIC: 309.8
```

The `glm()` function accepts a formula that defines the model.
The formula `hp ~ .` means "regress hp on all other variables". The `family` argument defines we are performing regression and the `data` argument points to the data frame where the covariates used in the formula are found.

For a gaussian output, we can also use the `lm()` function. There are minor differences in the output created, but the model is the same:

```
mod <- lm(hp ~ ., data = mtcars)
mod
```

```
Call:
lm(formula = hp ~ ., data = mtcars)

Coefficients:
(Intercept)          mpg          cyl         disp         drat           wt
    79.048       -2.063        8.204        0.439       -4.619      -27.660
       qsec           vs           am         gear         carb
     -1.784       25.813        9.486        7.216       18.749
```

Get summary of the model using `summary()`:

```
summary(mod)
```

```
Call:
lm(formula = hp ~ ., data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-38.681 -15.558   0.799  18.106  34.718

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  79.0484   184.5041   0.428  0.67270
mpg          -2.0631     2.0906  -0.987  0.33496
cyl           8.2037    10.0861   0.813  0.42513
disp          0.4390     0.1492   2.942  0.00778 **
drat         -4.6185    16.0829  -0.287  0.77680
wt          -27.6600    19.2704  -1.435  0.16591
qsec         -1.7844     7.3639  -0.242  0.81089
vs           25.8129    19.8512   1.300  0.20758
```

```
am            9.4863    20.7599   0.457  0.65240
gear          7.2164    14.6160   0.494  0.62662
carb         18.7487     7.0288   2.667  0.01441 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 25.97 on 21 degrees of freedom
Multiple R-squared:  0.9028,   Adjusted R-squared:  0.8565
F-statistic:  19.5 on 10 and 21 DF,  p-value: 1.898e-08
```

Note how R prints stars next to covariates whose p-values falls within certain limits, described right below the table of estimates.
Above, for example, the p-value for `disp` falls between 0.001 and 0.01 and therefore gets highlighted with 2 stars.

To extract the p-values of the intercept and each coefficient, we use `coef()` on `summary()`. The final (4th) column lists the p-values:

```
coef(summary(mod))
```

```
             Estimate  Std. Error    t value    Pr(>|t|)
(Intercept) 79.0483879 184.5040756  0.4284371 0.672695339
mpg         -2.0630545   2.0905650 -0.9868407 0.334955314
cyl          8.2037204  10.0861425  0.8133655 0.425134929
disp         0.4390024   0.1492007  2.9423609 0.007779725
drat        -4.6185488  16.0829171 -0.2871711 0.776795845
wt         -27.6600472  19.2703681 -1.4353668 0.165910518
qsec        -1.7843654   7.3639133 -0.2423121 0.810889101
vs          25.8128774  19.8512410  1.3003156 0.207583411
am           9.4862914  20.7599371  0.4569518 0.652397317
gear         7.2164047  14.6160152  0.4937327 0.626619355
carb        18.7486691   7.0287674  2.6674192 0.014412403
```

0.155 Mass-univariate analysis

There are many cases where we have a large number of predictors and, along with any other number of tests or models, we may want to regress our outcome of interest on each covariate, one at a time.

Let's create some synthetic data with 1000 cases and 100 covariates
The outcome is generated using just 4 of those 100 covariates and has added noise.

```
set.seed(2020)
n_col <- 100
```

```
n_row <- 1000
x <- as.data.frame(lapply(seq(n_col), function(i) rnorm(n_row)),
                   col.names = paste0("Feature_", seq(n_col)))
dim(x)
```

```
[1] 1000  100
```

```
y <- .7 + x[, 10] + .3 * x[, 20] + 1.3 * x[, 30] + x[, 50] + rnorm(500)
```

Let's fit a linear model regressing y on each column of x using `lm`:

```
mod.xy.massuni <- lapply(seq(x), function(i) lm(y ~ x[, i]))
length(mod.xy.massuni)
```

```
[1] 100
```

```
names(mod.xy.massuni) <- paste0("mod", seq(x))
```

To extract p-values for each model, we must find where exactly to look.
Let's look into the first model:

```
(ms1 <- summary(mod.xy.massuni$mod1))
```

```
Call:
lm(formula = y ~ x[, i])

Residuals:
    Min      1Q  Median      3Q     Max
-8.5402 -1.4881 -0.0618  1.4968  5.8152

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.61800    0.06878   8.985   <2e-16 ***
x[, i]       0.08346    0.06634   1.258    0.209
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.174 on 998 degrees of freedom
Multiple R-squared:  0.001584,  Adjusted R-squared:  0.0005831
F-statistic: 1.583 on 1 and 998 DF,  p-value: 0.2086
```

```
ms1$coefficients
```

```
             Estimate Std. Error  t value     Pr(>|t|)
(Intercept) 0.61800326 0.06878142 8.985032 1.266204e-18
x[, i]      0.08346393 0.06634074 1.258110 2.086464e-01
```

The p-values for each feature is stored in row 1, column 4 fo the coefficients matrix. Let's extract all of them:

```
mod.xy.massuni.pvals <- sapply(mod.xy.massuni, function(i) summary(i)$coefficien
```

Let's see which variable are significant at the 0.05:

```
which(mod.xy.massuni.pvals < .05)
```

```
 mod5 mod10 mod12 mod20 mod28 mod30 mod42 mod50 mod61 mod65 mod72 mod82 mod85
   5   10   12   20   28   30   42   50   61   65   72   82   85
mod91 mod94 mod99
   91    94    99
```

...and which are significant at the 0.01 level:

```
which(mod.xy.massuni.pvals < .01)
```

```
mod10 mod20 mod28 mod30 mod50
   10    20    28    30    50
```

0.156 Multiple comparison correction

We've performed a large number of tests and before reporting the results, we need to control for multiple comparisons[36].
To do that, we use R's `p.adjust()` function. It adjusts a vector of p-values to account for multiple comparisons using one of multiple methods. The default, and recommended, is the Holm method[37]. It ensures that `FWER < α`, i.e. controls the family-wise error rate[38], a.k.a. the probability of making one or more false discoveries (Type I errors)

[36] https://en.wikipedia.org/wiki/Multiple_comparisons_problem
[37] https://en.wikipedia.org/wiki/Holm%E2%80%93Bonferroni_method
[38] https://en.wikipedia.org/wiki/Family-wise_error_rate

```
mod.xy.massuni.pvals.holm_adjusted <- p.adjust(mod.xy.massuni.pvals)
```

Now, let's see which features' p-values survive the magical .05 threshold:

```
which(mod.xy.massuni.pvals.holm_adjusted < .05)
```

```
mod10 mod20 mod30 mod50
   10    20    30    50
```

These are indeed the correct features (not surprisingly, still reassuringly).

Supervised Learning

This is a very brief introduction to machine learning using the **rtemis**[39] package. **rtemis** includes a large number of functions for:

- Data preprocessing
- Unsupervised learning: clustering & dimensionality reduction
- Supervised learning: regression & classification
- Visualization: static & dynamic (interactive) plots

0.157 Installation

If you do not have the **remotes** package, install it first:

```
install.packages("remotes")
```

Install **rtemis**:

```
remotes::install_github("egenn/rtemis")
```

rtemis uses multiple packages under the hood. Since you would not ever need to use all of the functions that rely on all of the packages, they are not installed by default as that would be very wasteful. Every time you call an **rtemis** function, it first checks if the required packages are present, and if not it identifies which one/s is/are needed by the specific function so that you can install them and procede.

For this short tutorial, start by installing the following, if not already on your system:

```
install.packages("ranger")
```

Load **rtemis**:

[39] `https://rtemis.lambdamd.org`

```
library(rtemis)
```

```
  .:rtemis 0.8.0: Welcome, egenn
 [x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
 Documentation & vignettes: https://rtemis.lambdamd.org
```

0.158 Data Input for Supervised Learning

All **rtemis** supervised learning functions begin with `s.` ("supervised").
They accept the same first four arguments:
`x`, `y`, `x.test`, `y.test`
but are flexible and allow you to also provide combined (x, y) and (x.test, y.test) data frames, as explained below.

0.158.1 Scenario 1 (`x.train, y.train, x.test, y.test`)

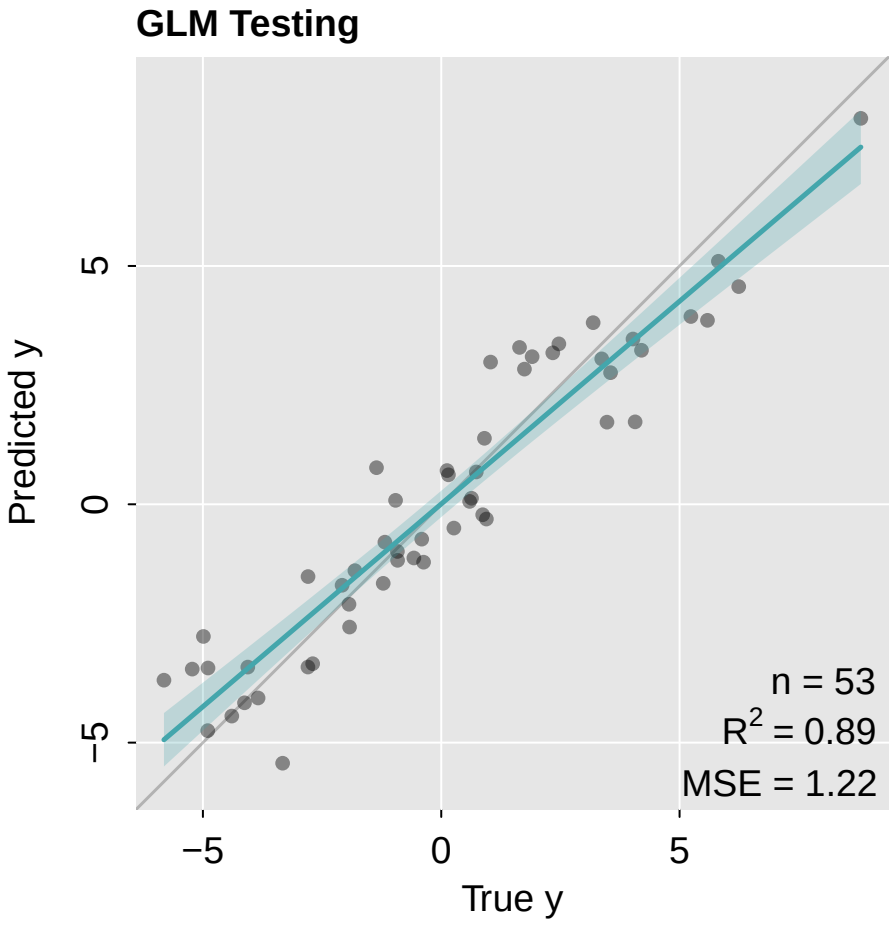
In the most straightforward case, provide each featureset and outcome individually:

- `x`: Training set features
- `y`: Training set outcome
- `x.test`: Testing set features (Optional)
- `y.test`: Testing set outcome (Optional)

```
x <- rnormmat(200, 10, seed = 2019)
w <- rnorm(10)
y <- x %*% w + rnorm(200)
res <- resample(y, seed = 2020)
```

```
x.train <- x[res$Subsample_1, ]
x.test <- x[-res$Subsample_1, ]
y.train <- y[res$Subsample_1]
y.test <- y[-res$Subsample_1]
```

```
mod.glm <- s.GLM(x.train, y.train, x.test, y.test)
```



0.158.2 Scenario 2: (`x.train, x.test`)

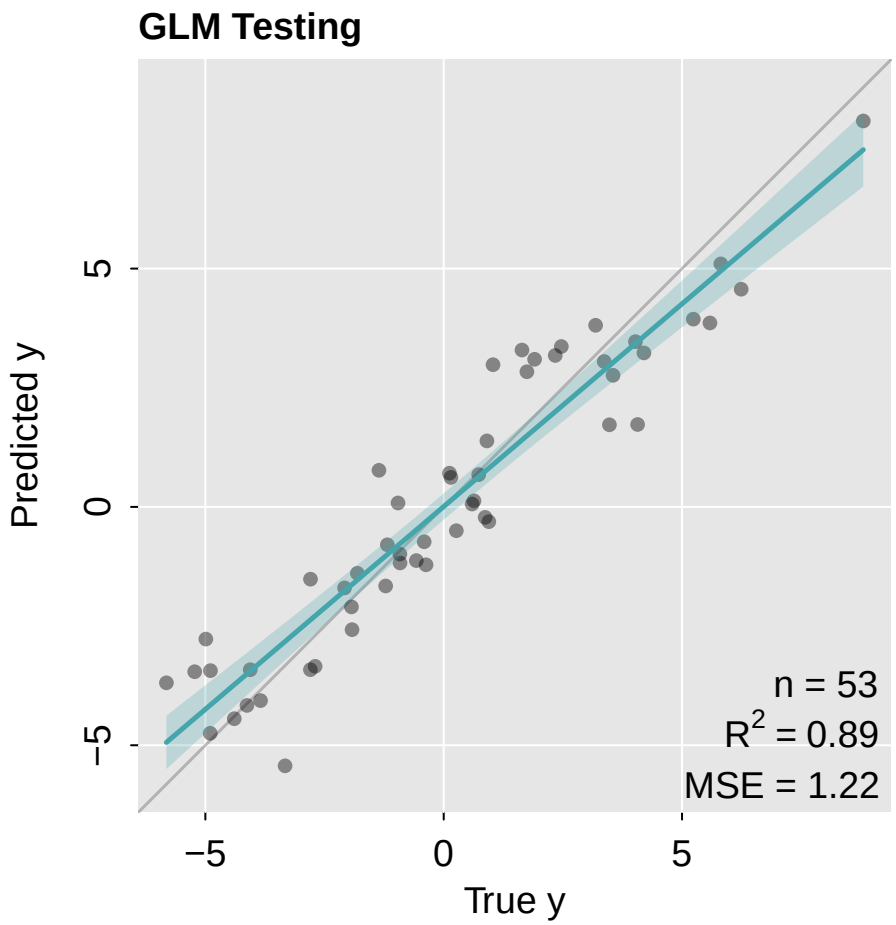
You can provide training and testing sets as a single data.frame each, where the last column is the outcome. Now `x` is the full training data and `y` the full testing data:

- `x`: data.frame(x.train, y.train)
- `y`: data.frame(x.test, y.test)

```
x <- rnormmat(200, 10, seed = 2019)
w <- rnorm(10)
y <- x %*% w + rnorm(200)
dat <- data.frame(x, y)
res <- resample(dat, seed = 2020)
```

```
dat.train <- dat[res$Subsample_1, ]
dat.test <- dat[-res$Subsample_1, ]
```

```
mod.glm <- s.GLM(dat.train, dat.test)
```



The `dataPrepare()` function will check data dimensions and determine whether data was input as separate feature and outcome sets or combined and ensure the correct number of cases and features was provided.

In either scenario, Regression will be performed if the outcome is numeric and Classification if the outcome is a factor.

0.159 Regression

0.159.1 Check Data with `checkData()`

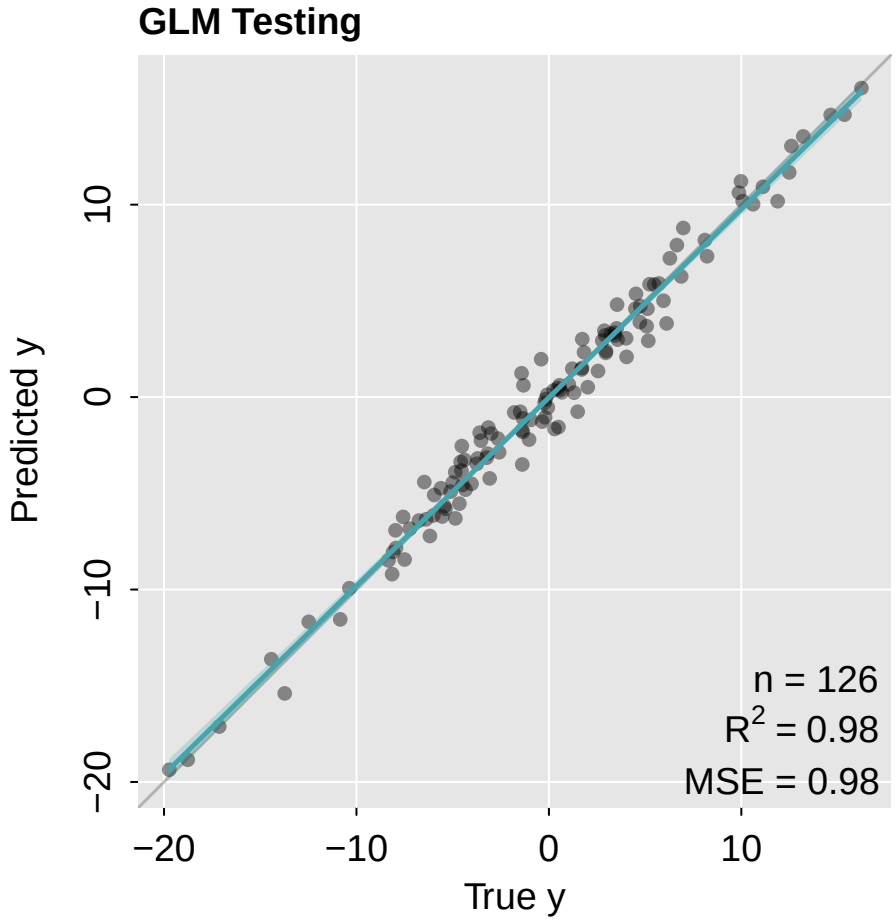
```
x <- rnormmat(500, 50, seed = 2019)
w <- rnorm(50)
y <- x %*% w + rnorm(500)
dat <- data.frame(x, y)
res <- resample(dat)
```

```
dat.train <- dat[res$Subsample_1, ]
dat.test <- dat[-res$Subsample_1, ]
```

```
checkData(x)
```

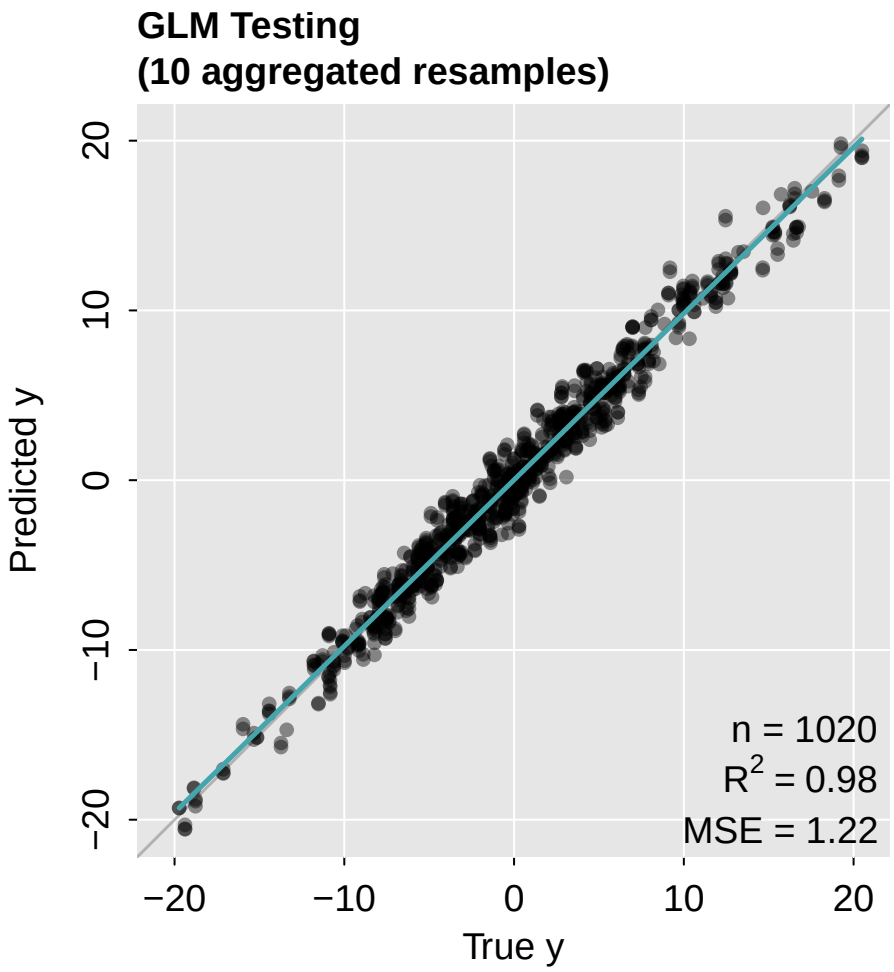
0.159.2 Single Model

```
mod <- s.GLM(dat.train, dat.test)
```



0.159.3 Crossvalidated Model

```
mod <- elevate(dat, mod = "glm")
```

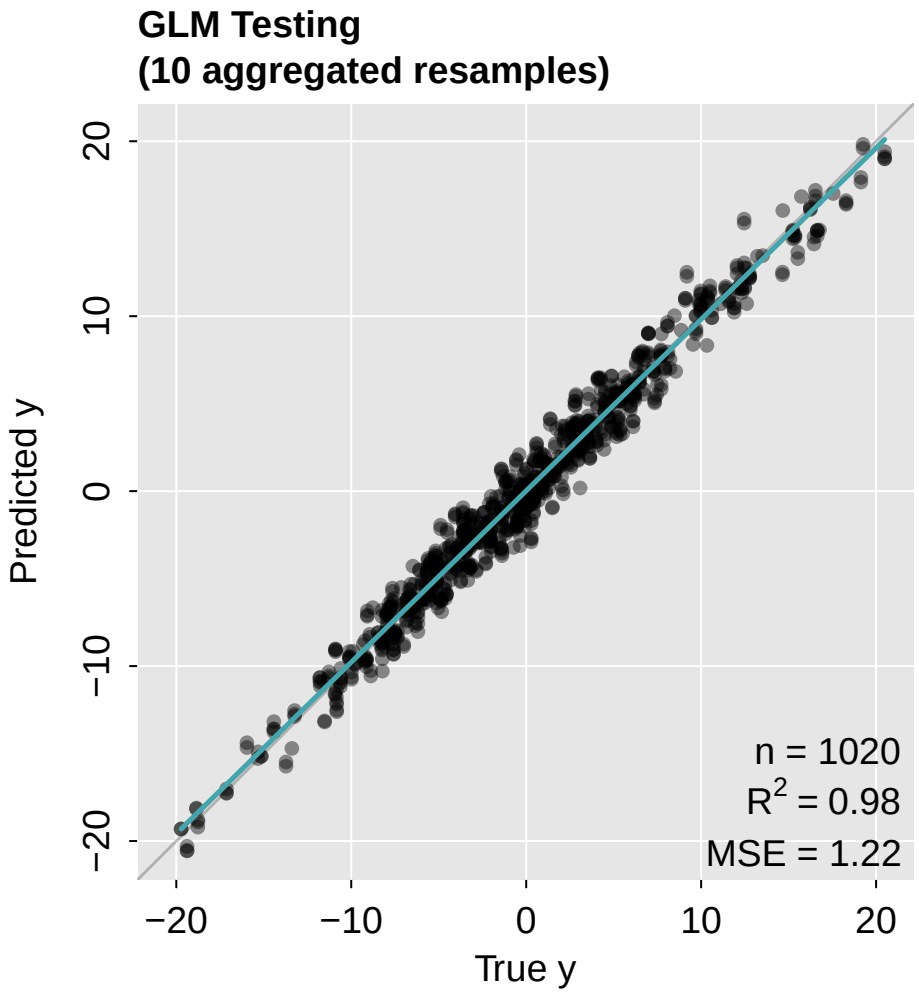


Use the `describe()` function to get a summary in (plain) English:

```
mod$describe()
```

```
Regression was performed using Generalized Linear Model. Model generalizability was
```

```
mod$plot()
```



0.160 Classification

0.160.1 Check Data

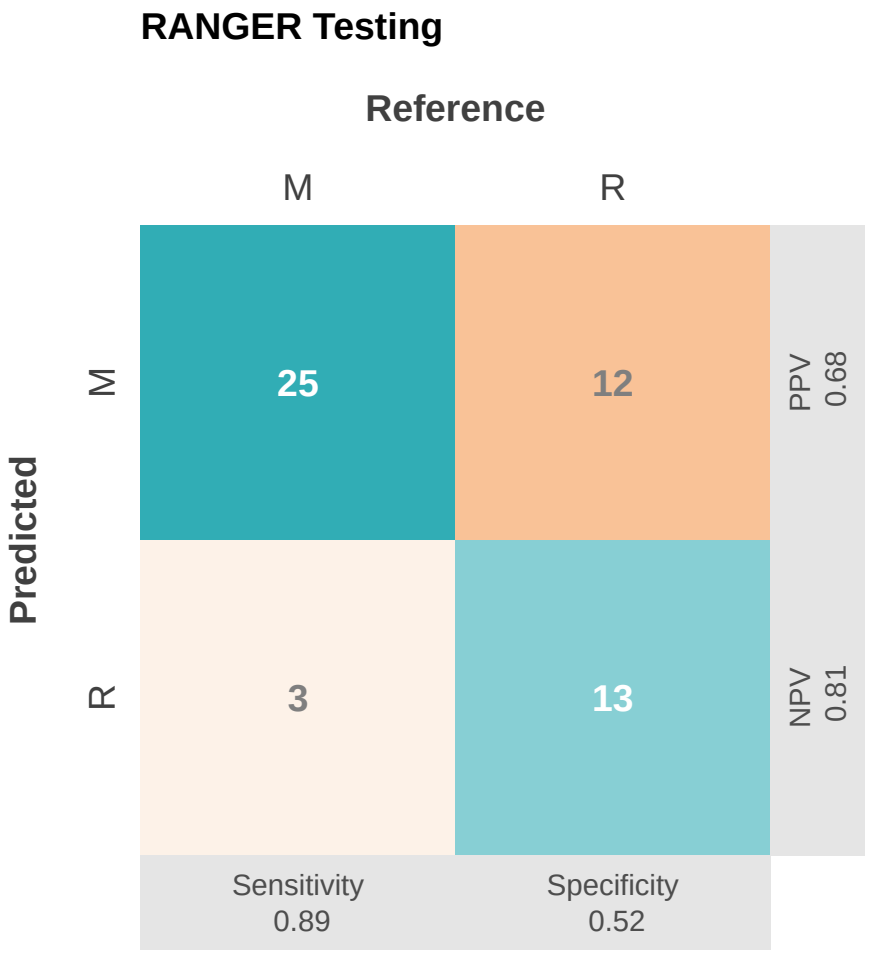
```
data(Sonar, package = 'mlbench')
checkData(Sonar)
```

```
res <- resample(Sonar)
```

```
sonar.train <- Sonar[res$Subsample_1, ]
sonar.test <- Sonar[-res$Subsample_1, ]
```

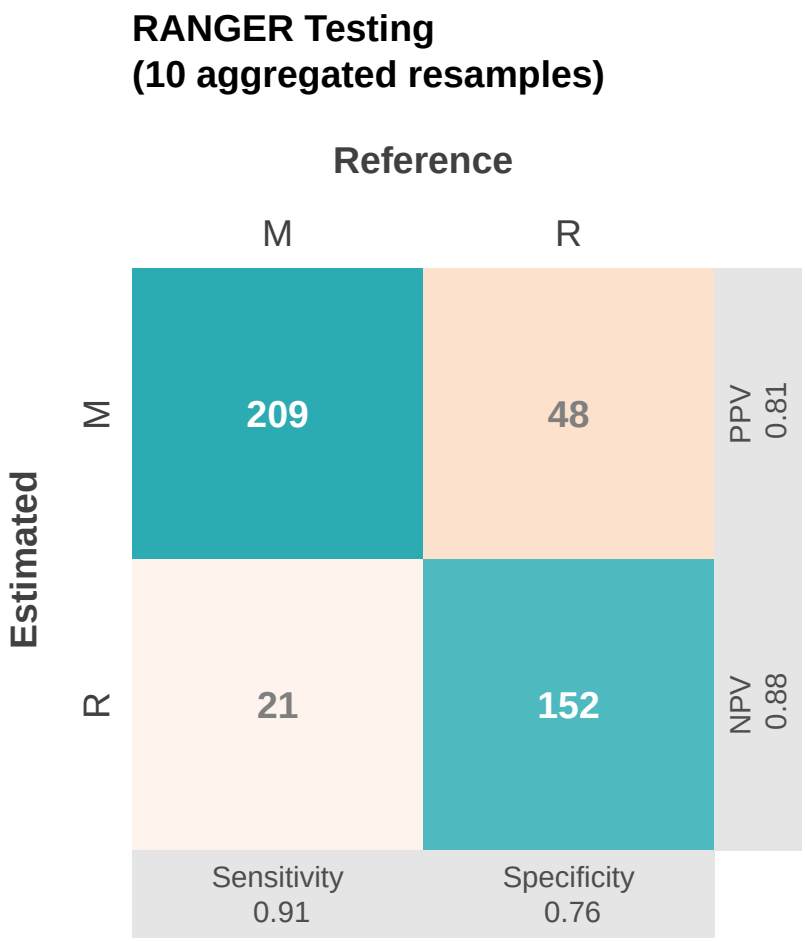
0.160.2 Single model

```
mod <- s.RANGER(sonar.train, sonar.test)
```



0.160.3 Crossvalidated Model

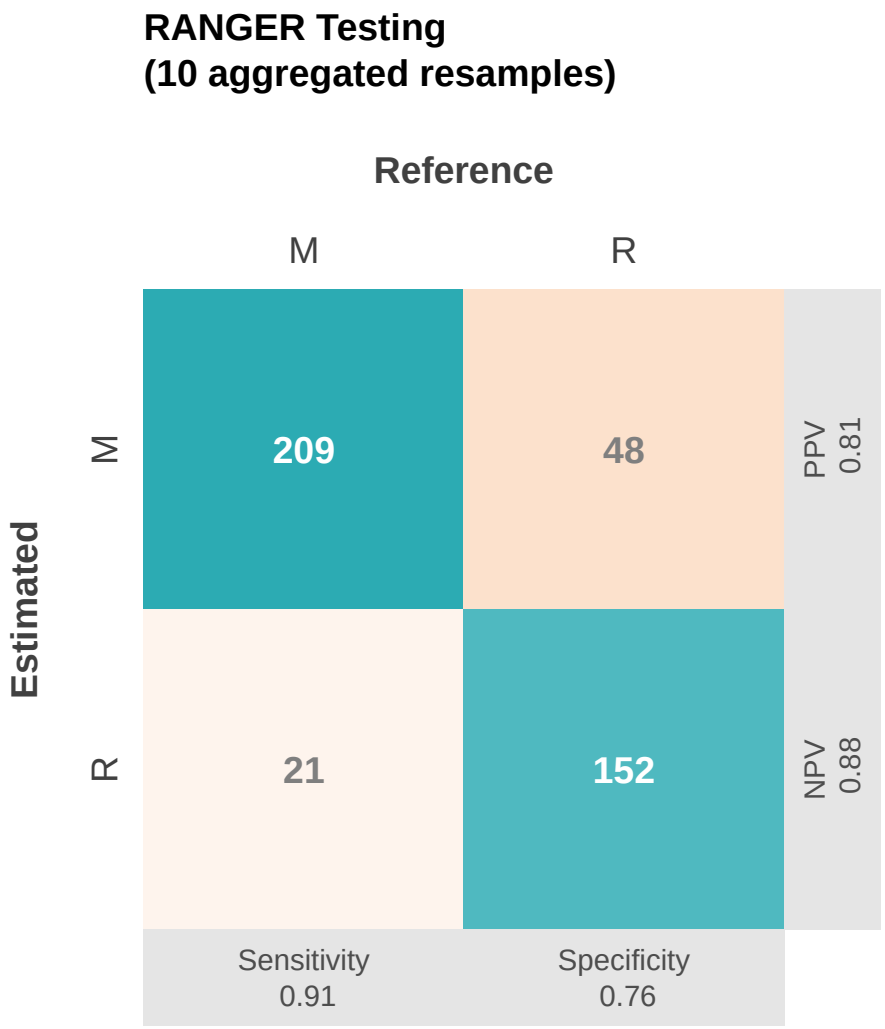
```
mod <- elevate(Sonar)
```



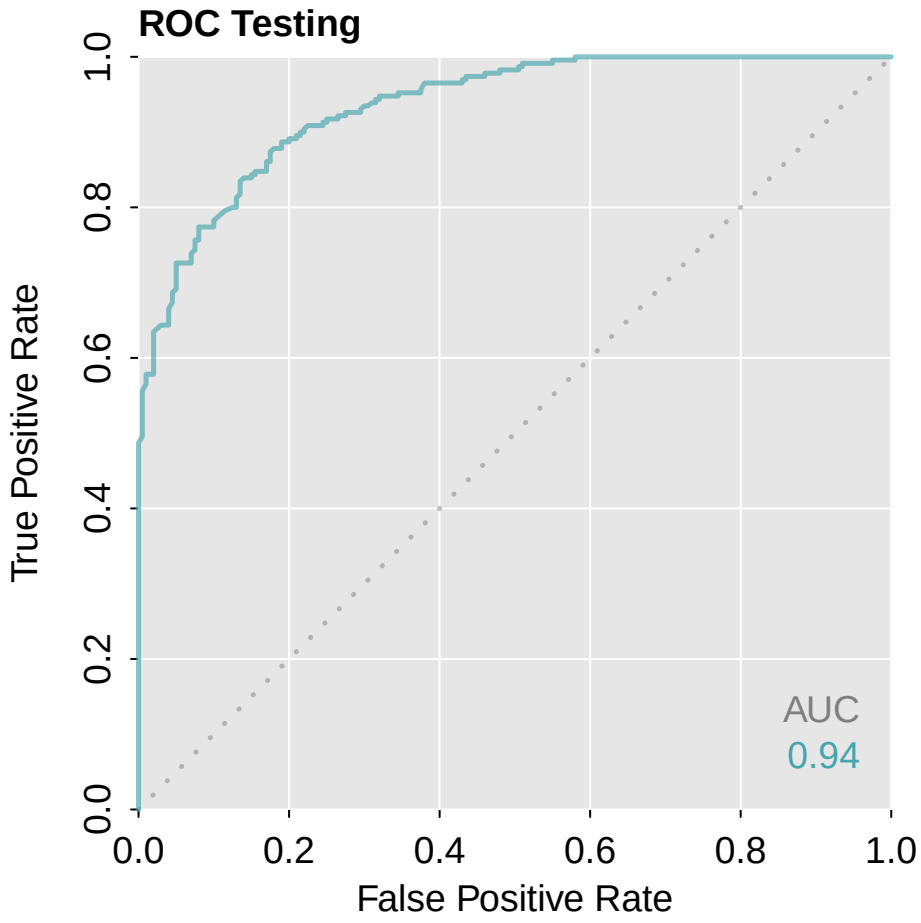

```
mod$describe()
```

```
Classification was performed using Random Forest (ranger). Model generaliz
```

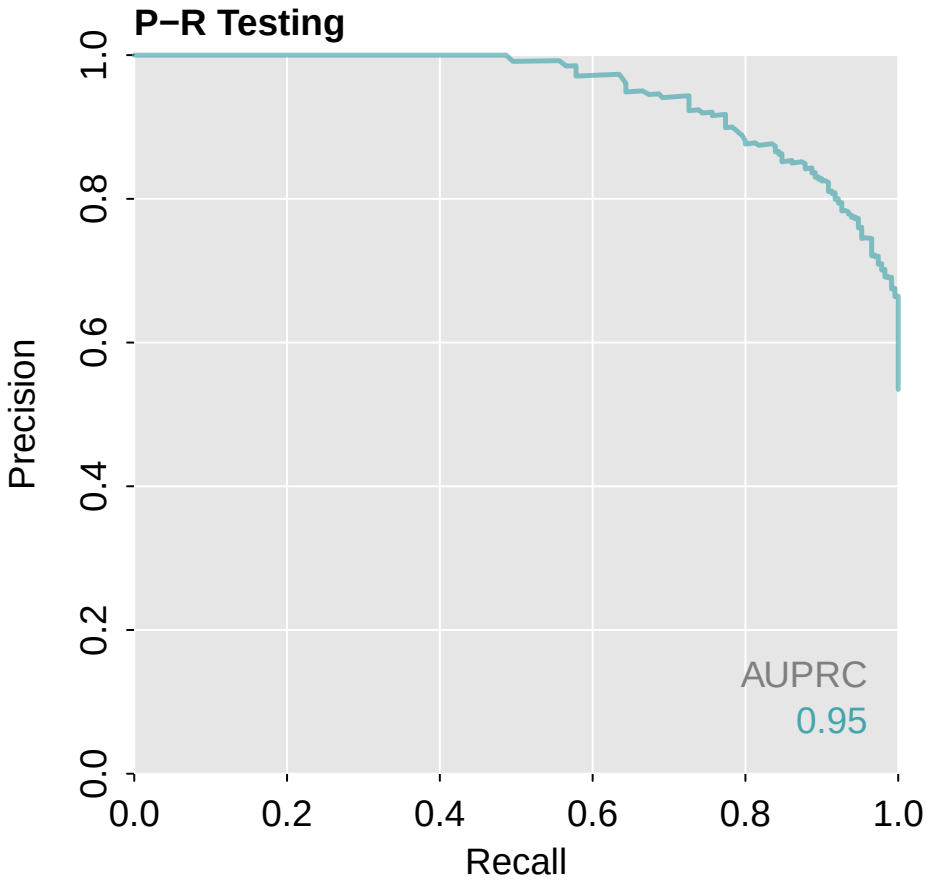
```
mod$plot()
```


RANGER Testing
(10 aggregated resamples)
Reference
M
R
Estimated
M
R
209
48
21
152
PPV
0.81
NPV
0.88
Sensitivity
0.91
Specificity
0.76

```
mod$plotROC()
```


ROC Testing
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC
0.94

```
mod$plotPR()
```



0.161 rtemis documentation

For more information on using **rtemis**, see the rtemis online documentation and vignettes[40]

[40] `https://rtemis.lambdamd.prg`

Unsupervised Learning

```
 .:rtemis 0.8.0: Welcome, egenn
[x86_64-apple-darwin17.0 (64-bit): Defaulting to 4/4 available cores]
Documentation & vignettes: https://rtemis.lambdamd.org
```

Unsupervised learning aims to learn relationships within a dataset without focusing at a particular outcome. You will often hear of unsupervised learning being performed on unlabeled data. To be clear, it means it does not use the labels to guide learning - whether labels are available or not. You might, for example, perform unsupervised learning ahead of supervised learning as we shall see later. Unsupervised learning includes a number of approaches, most of which can be divided into two categories:

- **Clustering**: Cases are grouped together based on some derived measure of similarity / distance metric.
- **Dimensionality Reduction / Matrix decomposition**: Variables are combined / projected into a lower dimensional space.

In **rtemis**, clustering algorithms begin with `u.` and decomposition/dimensionality reduction algorithms begin with `d.` (We use `u.` because `c.` is reserved for the builtin R function)

0.162 Decomposition / Dimensionality Reduction

Use `decomSelect()` to get a listing of available decomposition algorithms:

```
decomSelect()
```

```
.:decomSelect
rtemis supports the following decomposition algorithms:

    Name                                  Description
     CUR                      CUR Matrix Approximation
   H2OAE                                H2O Autoencoder
```

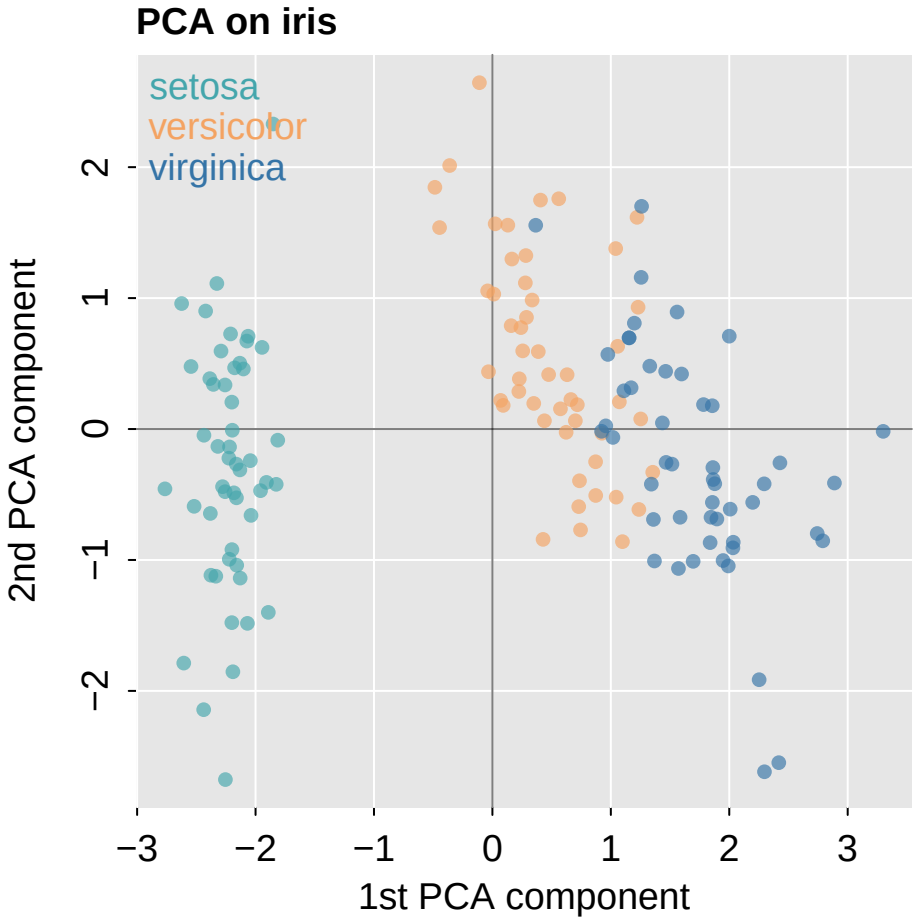
```
 H2OGLRM                H2O Generalized Low-Rank Model
     ICA                Independent Component Analysis
  ISOMAP                                        ISOMAP
    KPCA           Kernel Principal Component Analysis
     LLE                       Locally Linear Embedding
     MDS                       Multidimensional Scaling
     NMF             Non-negative Matrix Factorization
     PCA                   Principal Component Analysis
    SPCA           Sparse Principal Component Analysis
     SVD                   Singular Value Decomposition
    TSNE   t-distributed Stochastic Neighbor Embedding
    UMAP Uniform Manifold Approximation and Projection
```

We can further divide decomposition algorithms into linear (e.g. PCA, ICA, NMF) and nonlinear dimensionality reduction, (also called manifold learning, like LLE and tSNE).

0.162.0.1 Principal Component Analysic (PCA)

```
x <- iris[, 1:4]
iris.PCA <- d.PCA(x)
```

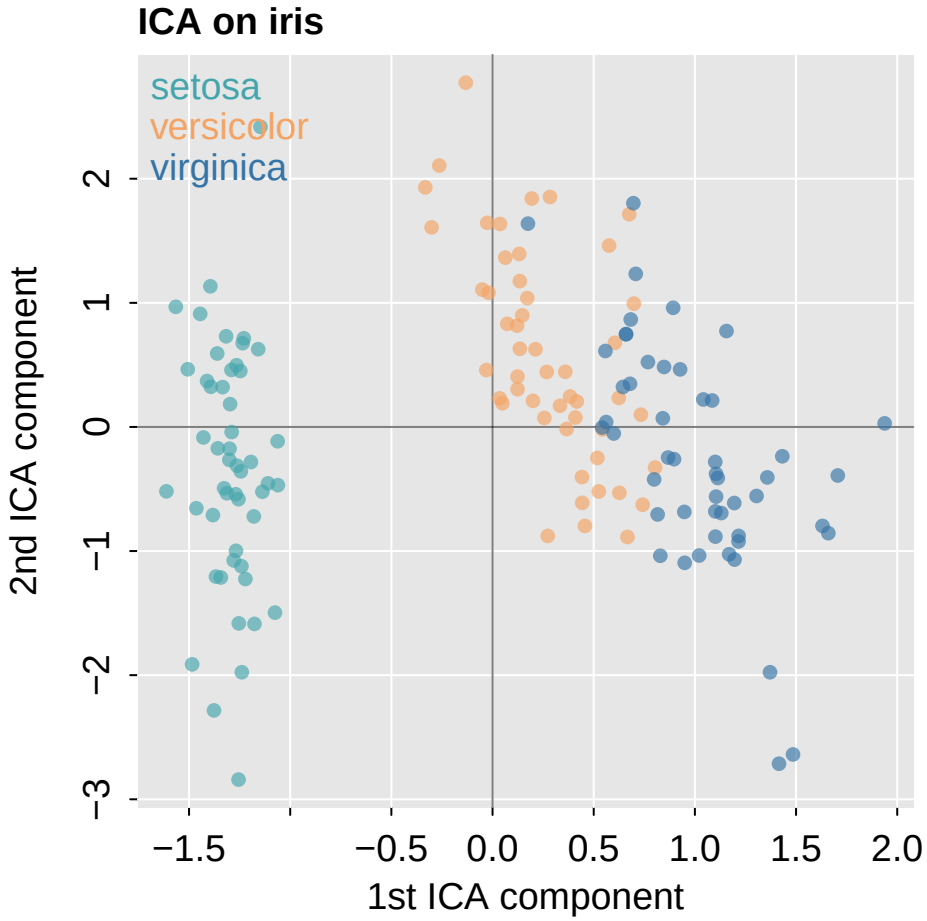
```
mplot3.xy(iris.PCA$projections.train[, 1], iris.PCA$projections.train[,
          xlab = "1st PCA component", ylab = "2nd PCA component", main =
```



0.162.0.2 Independent Component Analysis (ICA)

```
iris.ICA <- d.ICA(x, k = 2)
```

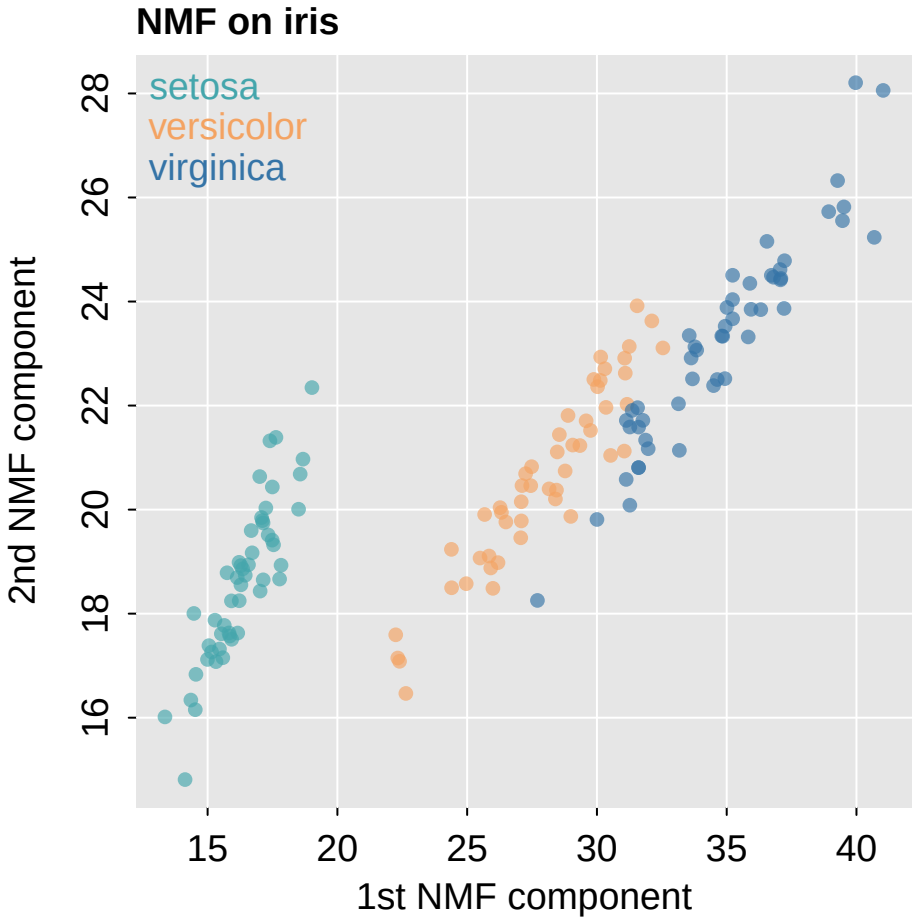
```
mplot3.xy(iris.ICA$projections.train[, 1], iris.ICA$projections.train[, 2], grou
          xlab = "1st ICA component", ylab = "2nd ICA component", main = "ICA on
```



0.162.0.3 Non-negative Matrix Factorization (NMF)

```
iris.NMF <- d.NMF(x, k = 2)
```

```
mplot3.xy(iris.NMF$projections.train[, 1], iris.NMF$projections.train[,
          xlab = "1st NMF component", ylab = "2nd NMF component", main =
```



0.163 Clustering

Use `clustSelect()` to get a listing of available clustering algorithms:

```
clustSelect()
```

```
.:clustSelect
rtemis supports the following clustering algorithms:

     Name                                  Description
   CMEANS                      Fuzzy C-means Clustering
      EMC             Expectation Maximization Clustering
   HARDCL                       Hard Competitive Learning
```

```
   HOPACH Hierarchical Ordered Partitioning And Collapsing Hybrid
H2OKMEANS                                H2O K-Means Clustering
   KMEANS                                    K-Means Clustering
     NGAS                                 Neural Gas Clustering
      PAM                          Partitioning Around Medoids
     PAMK          Partitioning Around Medoids with k estimation
     SPEC                                 Spectral Clustering
```

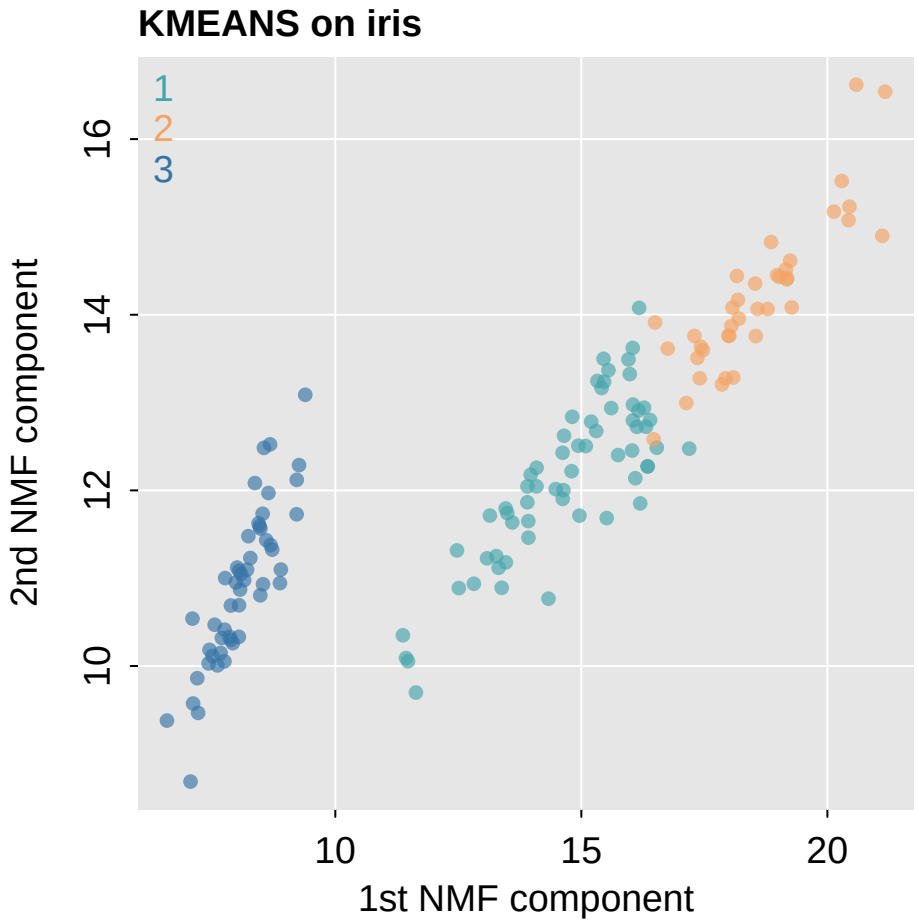
Let's cluster iris and we shall also use an NMF decomposition as we saw above to project to 2 dimensions.
We'll use two of the most popular clustering algorithms, K-means and PAM, aka K-medoids.

```
x <- iris[, 1:4]
iris.NMF <- d.NMF(x, k = 2)
```

0.163.1 K-Means

```
iris.KMEANS <- u.KMEANS(x, k = 3)
```

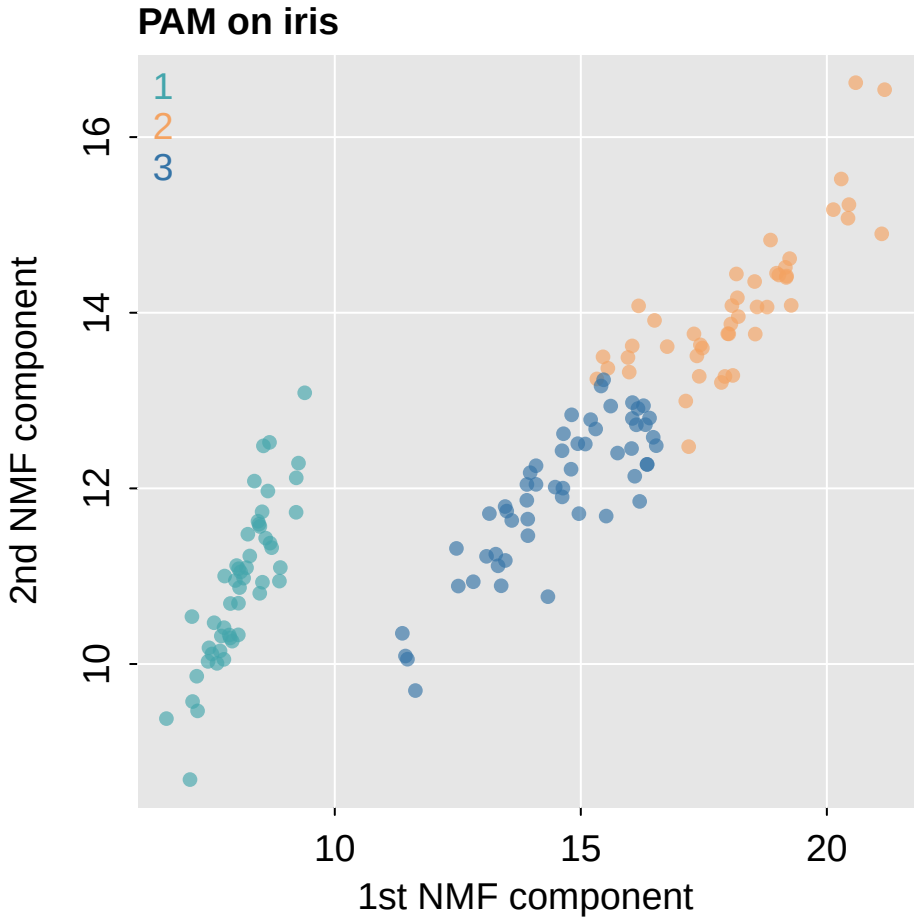
```
mplot3.xy(iris.NMF$projections.train[, 1], iris.NMF$projections.train[,
          group = iris.KMEANS$clusters.train,
          xlab = "1st NMF component", ylab = "2nd NMF component", main =
```



0.163.2 Partitioning Around Medoids with k estimation (PAMK)

```
iris.pamk <- u.PAMK(x, krange = 3:10)
```

```
mplot3.xy(iris.NMF$projections.train[, 1], iris.NMF$projections.train[, 2],
          group = iris.pamk$clusters.train,
          xlab = "1st NMF component", ylab = "2nd NMF component", main = "PAM on
```


PAM on iris
1
2
3
2nd NMF component
16
14
12
10
10
15
20
1st NMF component

Git & GitHub: the basics

`git` is famously powerful and notoriously complex. This is a very brief introduction to a very small subset of `git`'s functionality. Multiple online resources can help you delve into `git` in considerably more depth.

First, some important definitions:

- **Git**: System that tracks changes to code from multiple users
  - Free and open source distributed version control system
  - Developed in 2005 by Linus Torvalds to support Linux kernel development
  - Used by 87.2% of developers as of 2018, according to Stack Overflow
- **Repository**: Data (code) + metadata (i.e. log of changes over time)
  - Data structure that holds metadata for a set of directories / files (set of commit objects, historical record of changes)
- **GitHub**: Online service that holds Git repositories (public & private)
  - Git repository hosting service
  - Largest source code host in the world: > 40M users, > 100M repositories
  - Acquired by Microsoft for $7.5 billion in 2018.

0.164 Installing git

Check if you system already includes an installation of git. If not you can download it from the official git website[41]

0.165 Basic git usage

In the system terminal, all git commanda begin with `git` and are followed by a command name:

[41] `https://git-scm.com/downloads`

0.165.1 Cloning ("Downloading")

Download a repository to your computer for the first time. Replace "user" with the username and "repo" with the repository name.

```
git clone https://github.com/user/repo.git
```

This will clone the remote repository to a folder name 'repo'. You can optionally provide a different folder name after the URL.

To update a previously cloned repository:

```
git pull
```

0.165.2 Pushing ("Uploading")

Get info on local changes to repository:

```
git status
```

Working locally, stage new or modified files:

```
git add /path/to/file
```

Still working locally, commit changes with an informative message:

```
git commit –m Fixed this or added that
```

(Note that the previous steps did not require an internet connection - this one does) Push one or multiple commits to remote repository:

```
git push
```

0.165.3 Collaborating

The main way of contributing to a project is by a) making a new "branch" of the repository, b) making your edits, and c) either merging to master yourself or requesting your edits be merged by the owner/s of the repository. This allows multiple people to work on the codebase without getting in each other's way.

0.165.4 Branching and merging

Scenario: you are working on **your own** project, hosted on its own repository. You want to develop a new feature, which may take some time to code and test before you make it part of your official project code.
* Create a new branch, e.g. `devel` * Work in your new branch until all testing is successful * Merge back to `master` branch

Always from your system terminal, from within a directory in your repository: Create a new branch:

```
git branch devel
```

Switch to your new branch:

```
git checkout devel
```

Work on your code, using git add/commit/push as per usual.
When you are done testing and are happy to merge back to master:

```
git checkout master
git merge devel
git push
```

All the commits performed while you were working in the `devel` branch will be included in that last git push from `master`.

0.165.5 Pull request

Scenario: You are contributing to a repository along with other collaborators. You want to suggest a new feature is added to the code:

- Create a new branch, e.g. `mynewfeature`
- Work in new branch until you are ready happy to share and testing is complete
- Go on to the repository website, select your branch and perform a "Pull request" asking that the changes in your `mynewfeature` branch are merged into `master`
- The repository owner/s will review the request and can merge

0.166 Gists

GitHub also offers a very convenient pastebin[42]-like service called Gist, which lets you quickly and easily share code snippets.

To share some R code using a gist:

- Vist the gist site[43].
- Write in/copy-paste some code
- Add a name including a `.R` suffix at the top left of the entry box
- Copy-paste the URL to share with others

0.167 Git Resources

Git and GitHub are very powerful and flexible, with a great deal of functionality. Some resources to learn (a great deal) more:

- Git cheat sheet[44]
- GitHub guides[45] # Pro Git Book[46] by Scott Chacon and Ben Straub

0.168 Git and GitHub for open and reproducible science

It is recommended to create a new GitHub repository for each new research project. It may be worthwhile creating a new repository when it's time to publish a paper, to include all final working code that should accompany the publication (and e.g. exclude all trial-and-error, testing, etc. code). As Always, make sure to follow journal requirements for reporting data deposition (includes code) and accessibility.

0.169 Applications with builtin git support

Many applications support git, and allow you to pull / add / commit / push and more directly from the app using their GUI.
A couple of interest for the R user:

- RStudio[47] Out trusty IDE has a Git panel enabled when a project is in a directory that's part of a git repository

[42] https://en.wikipedia.org/wiki/Pastebin
[43] https://gist.github.com/
[44] https://education.github.com/git-cheat-sheet-education.pdf
[45] https://guides.github.com/
[46] https://git-scm.com/book/en/v2
[47] https://rstudio.com/

- The Atom[48] editor GitHub's own feature-packed text editor is naturally built around git and GitHub support. It offers its own package manager with access to a large and growing ecosystem of packages. Packages are available that transform Atom to a very capable and customizable IDE.

[48] https://atom.io

Introduction to the system shell

This is a very brief introduction to some of the most commonly used shell commands. A shell is a command line interface allowing access to an operating system's services. Multiple different shells exist, the most popular is probably **bash** (default in most Linux installations), which **zsh** was recently made the default in MacOS (was bash previously). The commands listed here will work similarly in all/most shells.

0.170 Common shell commands

The first thing to look for in a new environment is the help system. In the shell, this is accessed with `man`:

- `man`: Print the manual pages

```
man man
```

- `pwd`: Print working directory (the directory you are currently in)

```
pwd
```

- `cd`: Set working directory to `/path/to/dir`

```
cd /path/to/dir
```

- `mv`: Move `file` from `/current/dir/` to `/new/dir`

```
mv /current/dir/file /new/dir
```

- `mv`: Rename `file` to `newfilename`

```
mv /current/dir/file /current/dir/newfilename
```

- `cp`: Make a copy of `file` from `currentPath` into `altPath`

```
cp /currentPath/file /altPath/file
```

- `mkdir`: Create a new directory named 'newdir'

```
mkdir /path/to/newdir
```

- `rmdir`: Remove (i.e. delete) `uselessFile`

```

```

- `rm`: Remove (i.e. delete) `uselessFile`

```
rm /path/to/uselessFile
```

- `cat`: Print contents of `file` to the console

```
cat /path/to/file
```

- `uname`: Get system information

```
uname -a
```

- `whoami`: When you forget the basics

```
whoami
```

0.171 Running system commands within R

You can execute any system command within R using the `system()` command:

```
system("uname -a")
```

Resources

0.172 R Project

The R Manuals[49] include a number of resources, including:

- Introduction to R[50]
- CRAN task views[51] offer curated lists of packages by topic

0.173 R markdown

- R Markdown: The Definitive Guide[52] by Yihui Xie, J. J. Allaire, Garrett Grolemund
- bookdown: Authoring Books and Technical Documents with R Markdown[53]: how to make websites like this one you are on right now

0.174 Documentation

- Documentation with roxygen2[54]

[49] https://cran.r-project.org/manuals.html
[50] https://cran.r-project.org/doc/manuals/r-release/R-intro.html
[51] https://cran.r-project.org/web/views/
[52] https://bookdown.org/yihui/rmarkdown/
[53] https://bookdown.org/yihui/bookdown/
[54] https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html

0.175 R for data science

- R Programming for Data Science[55] by Roger D. Peng, based mostly on base R, and also covers the basics of **dplyr**.
- R for Data Science[56] by Hadley Wickham & Garrett Grolemund, based on the tidyverse[57]
- Data wrangling, exploration, and analysis with R[58]

0.176 Graphics

0.176.1 ggplot2

- ggplot2[59]

0.176.2 Plotly

- Plotly R API[60]
- Interactive web-based data visualization with R, plotly, and shiny[61]

0.177 Advanced R

- Efficient R Programming[62] by Colin Gillespie & Robin Lovelace
- High performance functions with Rcpp[63]

0.178 Git and GitHub

- GitHub guides[64]
- Pro Git Book[65] by Scott Chacon and Ben Straub

[55] https://bookdown.org/rdpeng/rprogdatascience/
[56] https://r4ds.had.co.nz
[57] https://www.tidyverse.org
[58] https://stat545.com/
[59] https://ggplot2.tidyverse.org/
[60] https://plot.ly/r/
[61] https://plotly-r.com
[62] https://bookdown.org/csgillespie/efficientR/
[63] http://adv-r.had.co.nz/Rcpp.html
[64] https://guides.github.com/
[65] https://git-scm.com/book/en/v2

0.179 Machine Learning

- An Introduction to Statistical Learning[66] offers an accessible view of core learning algorithms, without being math-heavy.
- Elements of Statistical Learning[67] offers a deeper and more extensive view on learning algorithms.
- Machine Learning with rtemis[68]

[66] https://www-bcf.usc.edu/~gareth/ISL/
[67] https://web.stanford.edu/~hastie/ElemStatLearn/
[68] https://rtemis.lambdamd.org/

Bibliography

Bengtsson, H. (2019). *matrixStats: Functions that Apply to Rows and Columns of Matrices (and to Vectors)*. R package version 0.55.0.

Buuren, S. v. and Groothuis-Oudshoorn, K. (2010). mice: Multivariate imputation by chained equations in r. *Journal of statistical software*, pages 1–68.

Gennatas, E. D. (2017). Towards precision psychiatry: Gray matter development and cognition in adolescence.

Murrell, P. (2018). *R graphics*. CRC Press.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2017). plotly: Create interactive web graphics via 'plotly. js'. *R package version*, 4(1):110.

Stekhoven, D. J. and Bühlmann, P. (2012). Missforest—non-parametric missing value imputation for mixed-type data. *Bioinformatics*, 28(1):112–118.

Wickham, H. (2011). ggplot2. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(2):180–185.

Wilkinson, L. (2012). The grammar of graphics. In *Handbook of Computational Statistics*, pages 375–414. Springer.

Wright, M. N. and Ziegler, A. (2015). ranger: A fast implementation of random forests for high dimensional data in c++ and r. *arXiv preprint arXiv:1508.04409*.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.21.2.