Base R
Cheat Sheet

Getting Help

Accessing the help files

?mean
Get help of a particular function.
help.search('weighted mean')
Search the help files for a word or phrase.
help(package = 'dplyr')
Find help for a package.

More about an object

str(iris)
Get a summary of an object's structure.
class(iris)
Find the class an object belongs to.

Using Packages

install.packages('dplyr')
Download and install a package from CRAN.

library(dplyr)
Load the package into the session, making all its functions available to use.

dplyr::select
Use a particular function from a package.

data(iris)
Load a built-in dataset into the environment.

Working Directory

getwd()
Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

c(2, 4, 6)	2 4 6	Join elements into a vector
2:6	2 3 4 5 6	An integer sequence
seq(2, 3, by=0.5)	2.0 2.5 3.0	A complex sequence
rep(1:2, times=3)	1 2 1 2 1 2	Repeat a vector
rep(1:2, each=3)	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

sort(x)
Return x sorted.
rev(x)
Return x reversed.
table(x)
See counts of values.
unique(x)
See unique values.

Selecting Vector Elements

By Position

x[4]	The fourth element.
x[-4]	All but the fourth.
x[2:4]	Elements two to four.
x[-(2:4)]	All elements except two to four.
x[c(1, 5)]	Elements one and five.

By Value

x[x == 10]	Elements which are equal to 10.
x[x < 0]	All elements less than zero.
x[x %in% c(1, 2, 5)]	Elements in the set 1, 2, 5.

Named Vectors

x['apple']	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){
   Do something
}
```

Example

```
for (i in 1:4){
   j <- i + 10
   print(j)
}
```

While Loop

```
while (condition){
   Do something
}
```

Example

```
while (i < 5){
   print(i)
   i <- i + 1
}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print('Yes')
} else {
   print('No')
}
```

Functions

```
function_name <- function(var){
   Do something
   return(new_variable)
}
```

Example

```
square <- function(x){
   squared <- x*x
   return(squared)
}
```

Reading and Writing Data

Also see the **readr** package.

Input	Ouput	Description
df <- read.table('file.txt')	write.table(df, 'file.txt')	Read and write a delimited text file.
df <- read.csv('file.csv')	write.csv(df, 'file.csv')	Read and write a comma separated value file. This is a special case of read.table/ write.table.
load('file.RData')	save(df, file = 'file.Rdata')	Read and write an R data file, a file type special for R.

Conditions

a == b	Are equal	a > b	Greater than	a >= b	Greater than or equal to	is.na(a)	Is missing
a != b	Not equal	a < b	Less than	a <= b	Less than or equal to	is.null(a)	Is null

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.
`as.factor`	`'1', '0', '1', levels: '1', '0'`	Character strings with preset levels. Needed for some statistical models.

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.

`m[2, ]` - Select a row

`m[ , 1]` - Select a column

`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

Also see the **dplyr** package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

`df$x` 

`df[[2]]` 

Understanding a data frame

`View(df)`	See the full data frame.
`head(df)`	See the first 6 rows.

Matrix subsetting

`df[ , 2]` 

`df[2, ]` 

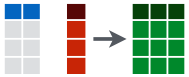
`df[2, 2]` 

`nrow(df)`
Number of rows.

`ncol(df)`
Number of columns.

`dim(df)`
Number of columns and rows.

`cbind` - Bind columns.



`rbind` - Bind rows.

Strings

Also see the **stringr** package.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.

`plot(x)`
Values of x in order.

`plot(x, y)`
Values of x against y.

`hist(x)`
Histogram of x.

Dates

See the **lubridate** package.

Advanced R
Cheat Sheet

Created by: Arianne Colton and Sean Chen

Environment Basics

Environment – **Data structure** (with two components below) that powers lexical scoping

Create environment: env1<-new.env()

1. **Named list** ("Bag of names") – each name points to an object stored elsewhere in memory.

   If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

   - Access with: **ls('env1')**
2. **Parent environment** – used to implement lexical scoping. If a name is not found in an environment, then R will look in its parent (and so on).
   - Access with: **parent.env('env1')**

Four special environments

1. **Empty environment** – ultimate ancestor of all environments
   - Parent: none
   - Access with: **emptyenv()**
2. **Base environment** - environment of the base package
   - Parent: empty environment
   - Access with: **baseenv()**
3. **Global environment** – the interactive workspace that you normally work in
   - Parent: environment of last attached package
   - Access with: **globalenv()**
4. **Current environment** – environment that R is currently working in (may be any of the above and others)
   - Parent: empty environment
   - Access with: **environment()**

Environments

Search Path

Search path – mechanism to look up objects, particularly functions.

- Access with : **search() –** lists all parents of the global environment (see Figure 1)
- Access any environment on the search path: **as.environment('package:base')**

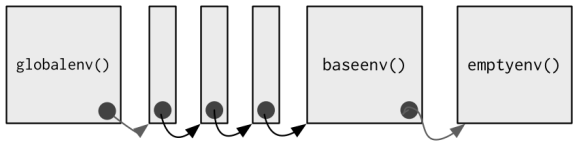


Figure 1 – The Search Path

- Mechanism : always start the search from global environment, then inside the latest attached package environment.
  - New package loading with **library()/require()** : new package is attached right after global environment. (See Figure 2)
  - Name conflict in two different package : functions with the same name, latest package function will get called.

search() :

'.GlobalEnv' ... 'Autoloads' 'package:base'

library(reshape2); search()

'.GlobalEnv' 'package:reshape2' ... 'Autoloads' 'package:base'

NOTE: Autoloads : special environment used for saving memory by only loading package objects (like big datasets) when needed

Figure 2 – Package Attachment

Binding Names to Values

Assignment – act of binding (or rebinding) a name to a value in an environment.

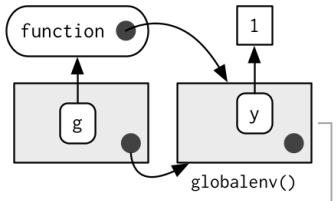
1. **<-** (Regular assignment arrow) – always creates a variable in the current environment
2. **<<-** (Deep assignment arrow) - modifies an existing variable found by walking up the parent environments

Warning: If <<- doesn't find an existing variable, it will create one in the global environment.

Function Environments

1. **Enclosing environment -** an environment where the function is created. It determines how function finds value.
   - Enclosing environment never changes, even if the function is moved to a different environment.
   - Access with: **environment('func1')**
2. **Binding environment -** all environments that the function has a binding to. It determines how we find the function.
   - Access with: **pryr::where('func1')**

   Example (for enclosing and binding environment):




```
y <- 1
e <- new.env()
e$g <- function(x) x + y
```

- function g enclosing environment is the global environment,
- the binding environment is "e".

3. **Execution environment -** new created environments to host a function call execution.
   - Two parents :

     I. Enclosing environment of the function

     II. Calling environment of the function
   - Execution environment is thrown away once the function has completed.
4. **Calling environment** - environments where the function was called.
   - Access with: **parent.frame('func1')**
   - Dynamic scoping :
     - About : look up variables in the calling environment rather than in the enclosing environment
     - Usage : most useful for developing functions that aid interactive data analysis

Updated: 2/16

Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Note: R has no 0-dimensional or scalar types. Individual numbers or strings, are actually vectors of length one, NOT scalars.

Human readable description of any R data structure :

```
str(variable)
```

Every **Object** has a mode and a class

1. **Mode**: represents how an object is stored in memory
   - 'type' of the object from R's point of view
   - Access with: **typeof()**
2. **Class**: represents the object's abstract type
   - 'type' of the object from R's object-oriented programming point of view
   - Access with: **class()**

	typeof()	class()
strings or vector of strings	character	character
numbers or vector of numbers	numeric	numeric
list	list	list
data.frame	list	data.frame

Factors

1. Factors are built on top of integer vectors using two attributes :

```
class(x) -> 'factor'
levels(x)  # defines the set of allowed values
```

2. Useful when you know the possible values a variable may take, even if you don't see all values in a given dataset.

Warning on Factor Usage:

1. Factors look and often behave like character vectors, they are actually integers. Be careful when treating them like strings.
2. Most data loading functions automatically convert character vectors to factors. (Use argument stringAsFactors = FALSE to suppress this behavior)

Object Oriented (OO) Field Guide

Object Oriented Systems

R has three object oriented systems :

1. **S3** is a very casual system. It has no formal definition of classes. It implements generic function OO.
   - **Generic-function OO** - a special type of function called a generic function decides which method to call.

Example:	drawRect(canvas, 'blue')
Language:	R

   - **Message-passing OO** - messages (methods) are sent to objects and the object determines which function to call.

Example:	canvas.drawRect('blue')
Language:	Java, C++, and C#

2. **S4** works similarly to S3, but is more formal. Two major differences to S3 :
   - **Formal class definitions** - describe the representation and inheritance for each class, and has special helper functions for defining generics and methods.
   - **Multiple dispatch** - generic functions can pick methods based on the class of any number of arguments, not just one.
3. **Reference classes** are very different from S3 and S4:
   - **Implements message-passing OO** - methods belong to classes, not functions.
   - **Notation** - $ is used to separate objects and methods, so method calls look like **canvas$drawRect('blue')**.

S3

1. **About S3 :**
   - R's first and simplest OO system
   - Only OO system used in the base and stats package
   - Methods belong to functions, not to objects or classes.
2. **Notation :**
   - **generic.class()**

mean.Date()	Date method for the generic - mean()

3. **Useful 'Generic' Operations**
   - Get all methods that belong to the 'mean' generic:
     - **Methods('mean')**
   - List all generics that have a method for the 'Date' class :
     - **methods(class = 'Date')**
4. **S3 objects** are usually built on top of lists, or atomic vectors with attributes.
   - Factor and data frame are S3 class
   - Useful operations:

Check if object is an S3 object	is.object(x) & !isS4(x) or pryr::otype()
Check if object inherits from a specific class	inherits(x, 'classname')
Determine class of any object	**class(x)**

Base Type (C Structure)

R base types - the internal C-level types that underlie the above OO systems.

- **Includes :** atomic vectors, list, functions, environments, etc.
- **Useful operation :** Determine if an object is a base type (Not S3, S4 or RC) **is.object(x)** returns FALSE
- **Internal representation :** C structure (or struct) that includes :
  - Contents of the object
  - Memory Management Information
  - Type
    - Access with: **typeof()**

Functions

Function Basics

Functions – objects in their own right

All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	"map" of the location of the function's variables (see "Enclosing Environment")

Every operation is a function call

- +, for, if, [, $, {
- x + y is the same as `+`(x, y)

Note: the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names.

Lexical Scoping

What is Lexical Scoping?

- Looks up value of a symbol. (see "Enclosing Environment")
- **findGlobals() -** lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function "+"
```

- R relies on lexical scoping to find everything, even the + operator.

Function Arguments

Arguments – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

f() -> 'a' 'x'	ls() evaluated inside f
f(ls())	ls() evaluated in global environment

Return Values

- **Last expression evaluated or explicit return().**
  Only use explicit return() when returning early.
- **Return ONLY single object.**
  Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

Primitive Functions

What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (..., na.rm = FALSE)  .Primitive('sum')
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

Influx Functions

What are Influx Functions?

1. Function name comes in between its arguments, like + or –
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

Replacement Functions

What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name xxx <-
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```


Updated: 2/16

Subsetting

Subsetting returns a copy of the original data, NOT copy-on modified

Simplifying vs. Preserving Subsetting

1. **Simplifying subsetting**
   - Returns the **simplest** possible data structure that can represent the output
2. **Preserving subsetting**
   - Keeps the structure of the output the **same** as the input.
   - When you use drop = FALSE, it's preserving

	Simplifying*	Preserving
Vector	x[[1]]	x[1]
List	x[[1]]	x[1]
Factor	x[1:4, drop = T]	x[1:4]
Array	x[1,] or x[, 1]	x[1, , drop = F] or x[, 1, drop = F]
Data frame	x[, 1] or x[[1]]	x[, 1, drop = F] or x[1]

Simplifying behavior varies slightly between different data types:

1. **Atomic Vector**
   - x[[1]] is the same as x[1]
2. **List**
   - [ ] always returns a list
   - Use [[ ]] to get list contents, this returns a single value piece out of a list
3. **Factor**
   - Drops any unused levels but it remains a factor class
4. **Matrix or Array**
   - If any of the dimensions has length 1, that dimension is dropped
5. **Data Frame**
   - If output is a single column, it returns a vector instead of a data frame

Data Frame Subsetting

Data Frame – possesses the **characteristics of both lists and matrices**. If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices

1. **Subset with a single vector** : Behave like lists

```
df1[c('col1', 'col2')]
```

2. **Subset with two vectors** : Behave like matrices

```
df1[, c('col1', 'col2')]
```

The results are the same in the above examples, however, results are different if subsetting with only one column. (see below)

1. **Behave like matrices**

```
str(df1[, 'col1']) -> int [1:3]
```

   - Result: the result is a vector

2. **Behave like lists**

```
str(df1['col1']) -> 'data.frame'
```

   - Result: the result remains a data frame of 1 column

$ Subsetting Operator

1. **About Subsetting Operator**
   - Useful shorthand for [[combined with character subsetting

```
x$y is equivalent to x[['y', exact = FALSE]]
```

2. **Difference vs. [[**
   - $ does partial matching, [[does not

```
x <- list(abc = 1)
x$a -> 1       # since "exact = FALSE"
x[['a']] ->    # would be an error
```

3. **Common mistake with $**
   - Using it when you have the name of a column stored in a variable

```
var <- 'cyl'
x$var

# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

Examples

1. **Lookup tables** (character subsetting)

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)
lookup[x]
> m   f   u   f   f   m   m
> 'Male'  'Female'  NA  'Female'  'Female'  'Male'  'Male'
unname(lookup[x])
> 'Male'  'Female'  NA  'Female'  'Female'  'Male'  'Male'
```

2. **Matching and merging by hand** (integer subsetting)

Lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c('Excellent', 'Good', 'Poor'),
  fail = c(F, F, T)
)
```

First Method

```
id <- match(grades, info$grade)
info[id, ]
```

Second Method

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

3. **Expanding aggregated counts** (integer subsetting)
   - **Problem**: a data frame where identical rows have been collapsed into one and a count column has been added
   - **Solution**: rep() and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: rep(x, y) rep replicates the values in x, y times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

4. **Removing columns from data frames** (character subsetting)

There are two ways to remove columns from a data frame:

Set individual columns to NULL	df1$col3 <- NULL
Subset to return only columns you want	df1[c('col1', 'col2')]

5. **Selecting rows based on a condition** (logical subsetting)
   - This is the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$col1 == 5 & df1$col2 == 4, ]
```


Updated: 2/16

Subsetting continued

Boolean Algebra vs. Sets
(Logical and Integer Subsetting)

1. **Using integer subsetting** is more effective when:
   - You want to find the first (or last) TRUE.
   - You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage.
2. **which()** - conversion from boolean representation to integer representation

```
which(c(T, F, T F)) -> 1 3
```

   - Integer representation length : is always <= boolean representation length
   - Common mistakes :
     I. Use **x[which(y)]** instead of **x[y]**
     II. **x[-which(y)]** is not equivalent to **x[!y]**

Recommendation:
Avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value

Subsetting with Assignment

1. All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
df1$col1[df1$col1 < 8] <- 0
```

2. Subsetting with nothing in conjunction with assignment :
   - Why : Preserve original object class and structure

```
df1[] <- lapply(df1, as.integer)
```

Debugging, Condition Handling and Defensive Programming

Debugging Methods

1. **traceback()** or **RStudio's error inspector**
   - Lists the sequence of calls that lead to the error
2. **browser()** or **RStudio's breakpoints tool**
   - Opens an interactive debug session at an arbitrary location in the code
3. **options(error = browser)** or **RStudio's "Rerun with Debug" tool**
   - Opens an interactive debug session where the error occurred
   - Error Options:

options(error = recover)

   - Difference vs. 'browser': can enter environment of any of the calls in the stack

options(error = dump_and_quit)

   - Equivalent to 'recover' for non-interactive mode
   - Creates **last.dump.rda** in the current working directory

In batch R process :

```
dump_and_quit <- function() {
    # Save debugging info to file last.dump.rda
    dump.frames(to.file = TRUE)
    # Quit R with error status
    q(status = 1)
}
options(error = dump_and_quit)
```

In a later interactive session :

```
load("last.dump.rda")
debugger()
```

Condition Handling of Expected Errors

1. **Communicating potential problems to users:**
   I. **stop()**
      - Action : raise fatal error and force all execution to terminate
      - Example usage : when there is no way for a function to continue
   II. **warning()**
      - Action : generate warnings to display potential problems
      - Example usage : when some of elements of a vectorized input are invalid
   III. **message()**
      - Action : generate messages to give informative output
      - Example usage : when you would like to print the steps of a program execution
2. **Handling conditions programmatically**:
   I. **try()**
      - Action : gives you the ability to continue execution even when an error occurs
   II. **tryCatch()**
      - Action : lets you specify handler functions that control what happens when a condition is signaled

```
result = tryCatch(code,
    error = function(c) "error",
    warning = function(c) "warning",
    message = function(c) "message"
)
```

Use conditionMessage(c) or c$message to extract the message associated with the original error.

Defensive Programming

Basic principle : "fail fast", to raise an error as soon as something goes wrong

1. **stopifnot()** or use 'assertthat' package - check inputs are correct
2. **Avoid subset(), transform() and with()** - these are non-standard evaluation, when they fail, often fail with uninformative error messages.
3. **Avoid [and sapply()** - functions that can return different types of output.
   - Recommendation : Whenever subsetting a data frame in a function, you should always use **drop = FALSE**

Updated: 2/16

R For Data Science *Cheat Sheet*

data.table

Learn Python for data science Interactively at www.DataCamp.com

data.table

data.table is an R package that provides a high-performance version of base R's `data.frame` with syntax and feature enhancements for ease of use, convenience and programming speed.

Load the package:

```
> library(data.table)
```

Creating A `data.table`

```
> set.seed(45L)
> DT <- data.table(V1=c(1L,2L),
                   V2=LETTERS[1:3],
                   V3=round(rnorm(4),4),
                   V4=1:12)
```

Create a `data.table` and call it `DT`

Subsetting Rows Using `i`

```
> DT[3:5,]
> DT[3:5]
> DT[V2=="A"]
> DT[V2 %in% c("A","C")]
```

Select 3rd to 5th row
Select 3rd to 5th row
Select all rows that have value `A` in column `V2`
Select all rows that have value `A` or `C` in column `V2`

Manipulating on Columns in `j`

```
> DT[,V2]
 [1] "A" "B" "C" "A" "B" "C" ...
> DT[,.(V2,V3)]
> DT[,sum(V1)]
[1] 18
> DT[,.(sum(V1),sd(V3))]
   V1        V2
1: 18 0.4546055
> DT[,.(Aggregate=sum(V1),
        Sd.V3=sd(V3))]
   Aggregate     Sd.V3
1:        18 0.4546055
> DT[,.(V1,Sd.V3=sd(V3))]

> DT[,.(print(V2),
        plot(V3),
        NULL)]
```

Return `V2` as a vector

Return `V2` and `V3` as a `data.table`
Return the sum of all elements of `V1` in a vector
Return the sum of all elements of `V1` and the std. dev. of `V3` in a `data.table`

The same as the above, with new names

Select column `V2` and compute std. dev. of `V3`, which returns a single value and gets recycled
Print column `V2` and plot `V3`

Doing `j` by Group

```
> DT[,.(V4.Sum=sum(V4)),by=V1]
   V1 V4.Sum
1:  1     36
2:  2     42
> DT[,.(V4.Sum=sum(V4)),
        by=.(V1,V2)]
> DT[,.(V4.Sum=sum(V4)),
        by=sign(V1-1)]
   sign V4.Sum
1:    0     36
2:    1     42
> DT[,.(V4.Sum=sum(V4)),
        by=.(V1.01=sign(V1-1))]
> DT[1:5,.(V4.Sum=sum(V4)),
          by=V1]
> DT[,.N,by=V1]
```

Calculate sum of `V4` for every group in `V1`

Calculate sum of `V4` for every group in `V1` and `V2`
Calculate sum of `V4` for every group in `sign(V1-1)`

The same as the above, with new name for the variable you're grouping by
Calculate sum of `V4` for every group in `V1` after subsetting on the first 5 rows
Count number of rows for every group in `V1`

General form: `DT[i, j, by]`

"Take `DT`, subset rows using `i`, then calculate `j` grouped by `by`"

Adding/Updating Columns By Reference in `j` Using `:=`

```
> DT[,V1:=round(exp(V1),2)]
> DT
     V1 V2      V3 V4
1: 2.72  A -0.1107  1
2: 7.39  B -0.1427  2
3: 2.72  C -1.8893  3
4: 7.39  A -0.3571  4
...
> DT[,c("V1","V2"):=list(round(exp(V1),2),
                         LETTERS[4:6])]
> DT[,':='(V1=round(exp(V1),2),
           V2=LETTERS[4:6])][]
        V1 V2      V3 V4
1:   15.18  D -0.1107  1
2: 1619.71  E -0.1427  2
3:   15.18  F -1.8893  3
4: 1619.71  D -0.3571  4
> DT[,V1:=NULL]
> DT[,c("V1","V2"):=NULL]
> Cols.chosen=c("A","B")
> DT[,Cols.Chosen:=NULL]

> DT[,(Cols.Chosen):=NULL]
```

`V1` is updated by what is after `:=`
Return the result by calling `DT`

Columns `V1` and `V2` are updated by what is after `:=`
Alternative to the above one. With `[]`, you print the result to the screen

Remove `V1`
Remove columns `V1` and `V2`

Delete the column with column name `Cols.chosen`
Delete the columns specified in the variable `Cols.chosen`

Indexing And Keys

```
> setkey(DT,V2)
> DT["A"]
   V1 V2      V3 V4
1:  1  A -0.2392  1
2:  2  A -1.6148  4
3:  1  A  1.0498  7
4:  2  A  0.3262 10
> DT[c("A","C")]
> DT["A",mult="first"]

> DT["A",mult="last"]

> DT[c("A","D")]
   V1 V2      V3 V4
1:  1  A -0.2392  1
2:  2  A -1.6148  4
3:  1  A  1.0498  7
4:  2  A  0.3262 10
5: NA  D      NA NA
> DT[c("A","D"),nomatch=0]
   V1 V2      V3 V4
1:  1  A -0.2392  1
2:  2  A -1.6148  4
3:  1  A  1.0498  7
4:  2  A  0.3262 10
> DT[c("A","C"),sum(V4)]

> DT[c("A","C"),
     sum(V4),
     by=.EACHI]
   V2 V1
1:  A 22
2:  C 30
> setkey(DT,V1,V2)
> DT[.(2,"C")]
   V1 V2      V3 V4
1:  2  C  0.3262  6
2:  2  C -1.6148 12
> DT[.(2,c("A","C"))]
   V1 V2      V3 V4
1:  2  A -1.6148  4
2:  2  A  0.3262 10
3:  2  C  0.3262  6
4:  2  C -1.6148 12
```

A key is set on `V2`; output is returned invisibly
Return all rows where the key column (set to `V2`) has the value `A`

Return all rows where the key column (`V2`) has value `A` or `C`
Return first row of all rows that match value `A` in key column `V2`

Return last row of all rows that match value `A` in key column `V2`

Return all rows where key column `V2` has value `A` or `D`

Return all rows where key column `V2` has value `A` or `D`

Return total sum of `V4`, for rows of key column `V2` that have values `A` or `C`
Return sum of column `V4` for rows of `V2` that have value `A`, and anohter sum for rows of `V2` that have value `C`

Sort by `V1` and then by `V2` within each group of `V1` (invisible)
Select rows that have value 2 for the first key (`V1`) and the value `C` for the second key (`V2`)

Select rows that have value 2 for the first key (`V1`) and within those rows the value `A` or `C` for the second key (`V2`)

Advanced Data Table Operations

```
> DT[.N-1]
> DT[,.N]
> DT[,.(V2,V3)]
> DT[,list(V2,V3)]
> DT[,mean(V3),by=.(V1,V2)]
   V1 V2      V1
1:  1  A  0.4053
2:  1  B  0.4053
3:  1  C  0.4053
4:  2  A -0.6443
5:  2  B -0.6443
6:  2  C -0.6443
```

Return the penultimate row of the `DT`
Return the number of rows
Return `V2` and `V3` as a `data.table`
Return `V2` and `V3` as a `data.table`
Return the result of `j`, grouped by all possible combinations of groups specified in `by`

`.SD` & `.SDcols`

```
> DT[,print(.SD),by=V2]
> DT[,.SD[c(1,.N)],by=V2]
> DT[,lapply(.SD,sum),by=V2]

> DT[,lapply(.SD,sum),by=V2,
     .SDcols=c("V3","V4")]
   V2     V3 V4
1:  A -0.478 22
2:  B -0.478 26
3:  C -0.478 30
> DT[,lapply(.SD,sum),by=V2,
     .SDcols=paste0("V",3:4)]
```

Look at what `.SD` contains
Select the first and last row grouped by `V2`
Calculate sum of columns in `.SD` grouped by `V2`

Calculate sum of `V3` and `V4` in `.SD` grouped by `V2`

Calculate sum of `V3` and `V4` in `.SD` grouped by `V2`

Chaining

```
> DT <- DT[,.(V4.SUM=sum(V4)),
           by=V1]
   V1 V4.Sum
1:  1     36
2:  2     42
> DT[V4.Sum>40]
> DT[,.(V4.Sum=sum(V4)),
       by=V1][V4.Sum>40]
   V1 V4.Sum
1:  2     42
2:  1     36
> DT[,.(V4.Sum=sum(V4)),
       by=V1][order(-V1)]
   V1 V4.Sum
1:  2     42
2:  1     36
```

Calculate sum of `V4`, grouped by `V1`

Select that group of which the sum is >40
Select that group of which the sum is >40 (chaining)

Calculate sum of `V4`, grouped by `V1`, ordered on `V1`

`set()`-Family

`set()`

Syntax: `for (i in from:to) set(DT, row, column, new value)`

```
> rows <- list(3:4,5:6)
> cols <- 1:2
> for(i in seq_along(rows))
    {set(DT,
         i=rows[[i]],
         j=cols[i],
         value=NA)}
```

Sequence along the values of `rows`, and for the values of `cols`, set the values of those elements equal to `NA` (invisible)

`setnames()`

Syntax: `setnames(DT,"old","new")[]`

```
> setnames(DT,"V2","Rating")
> setnames(DT,
           c("V2","V3"),
           c("V2.rating","V3.DC"))
```

Set name of `V2` to `Rating` (invisible)
Change 2 column names (invisible)

`setnames()`

Syntax: `setcolorder(DT,"neworder")`

```
> setcolorder(DT,
              c("V2","V1","V4","V3"))
```

Change column ordering to contents of the specified vector (invisible)

How Big is Your Graph?

An R Cheat Sheet

Introduction

All functions that open a device for graphics will have **height** and **width** arguments to control the size of the graph and a **pointsize** argument to control the relative font size. In **knitr**, you control the size of the graph with the chunk options, **fig.width** and **fig.height**. This sheet will help you with calculating the size of the graph and various parts of the graph within R.

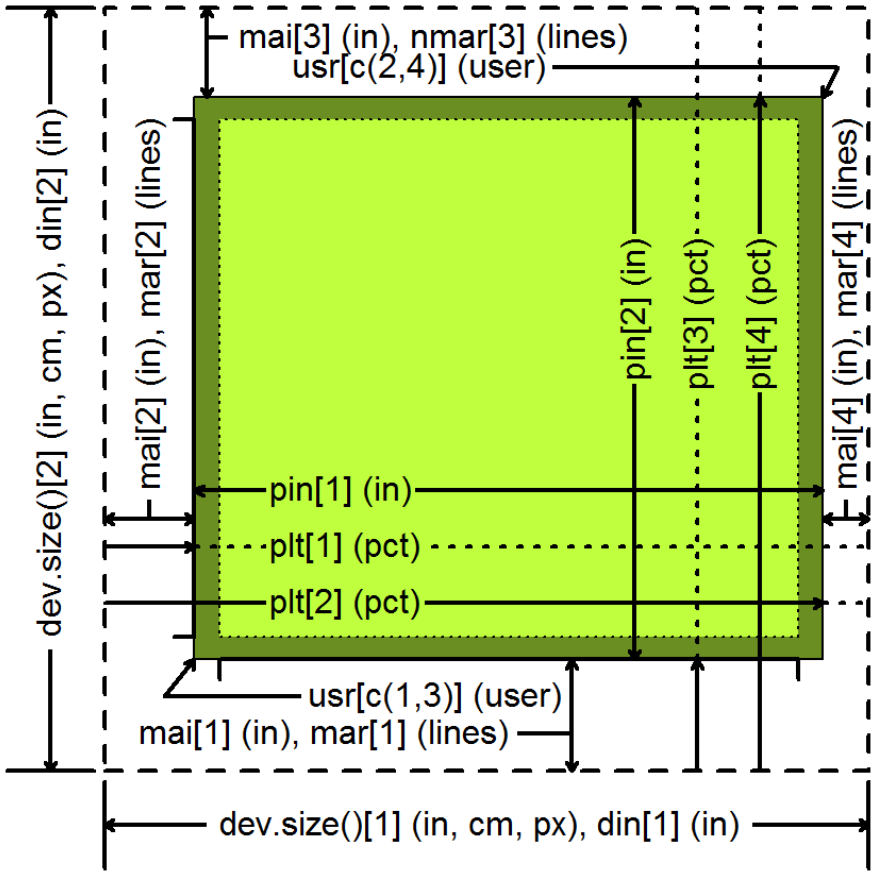
Your graphics device

dev.size() (width, height)
par("din") *(r.o.)* (width, height) in inches

Both the **dev.size** function and the **din** argument of **par** will tell you the size of the graphics device. The **dev.size** function will report the size in

1. inches (**units="in"**), the default
2. centimeters (**units="cm"**)
3. pixels (**units="px"**)

Like several other **par** arguments, **din** is read only *(r.o.)* meaning that you can ask its current value (**par("din")**) but you cannot change it (**par(din=c(5,7)** will fail).



Your plot margins

par("mai") (bottom, left, top, right) in inches
par("mar") (bottom, left, top, right) in lines

Margins provide you space for your axes, axis, labels, and titles.

A "line" is the amount of vertical space needed for a line of text.

If your graph has no axes or titles, you can remove the margins (and maximize the plotting region) with

par(mar=rep(0,4))

Your plotting region

par("pin") (width, height) in inches
par("plt") (left, right, bottom, top) in pct

The **pin** argument **par** gives you the size of the plotting region (the size of the device minus the size of the margins) in inches.

The **plt** argument gives you the percentage of the device from the left/bottom edge up to the left edge of the plotting region, the right edge, the bottom edge, and the top edge. The first and third values are equivalent to the percentage of space devoted to the left and bottom margins. Subtract the second and fourth values from 1 to get the percentage of space devoted to the right and top margins.

Your x-y coordinates

par("usr") (xmin, ymin, xmax, ymax)

Your x-y coordinates are the values you use when plotting your data. This normally is not the same as the values you specified with the **xlim** and **ylim** arguments in **plot**. By default, R adds an extra 4% to the plotting range (see the dark green region on the figure) so that points right up on the edges of your plot do not get partially clipped. You can override this by setting **xaxs="i"** and/or the **yaxs="i"** in **par**.

Run **par("usr")** to find the minimum X value, the maximum X value, the minimum Y value, and the maximum Y value. If you assign new values to **usr**, you will update the x-y coordinates to the new values.

Getting a square graph

par("pty")

You can produce a square graph manually by setting the width and height to the same value and setting the margins so that the sum of the top and bottom margins equal the sum of the left and right margins. But a much easier way is to specify **pty="s"**, which adjusts the margins so that the size of the plotting region is always square, even if you resize the graphics window.

Converting units

For many applications, you need to be able to translate user coordinates to pixels or inches. There are some cryptic shortcuts, but the simplest way is to get the range in user coordinates and measure the proportion of the graphics device devoted to the plotting region.

```
user.range <- par("usr")[c(2,4)] -
  par("usr")[c(1,3)]

region.pct <- par("plt")[c(2,4)] -
  par("plt")[c(1,3)]

region.px <-
  dev.size(units="px") * region.pct

px.per.xy <- region.px / user.range
```

To convert a horizontal or distance from the x-coordinate value to pixels, multiply by **px.per.xy[1]**. To convert a vertical distance, multiply by **region.px.per.xy[2]**. To convert a diagonal distance, you need to invoke Pyhthagoras.

```
a.px <- x.dist*px.per.xy[1]
b.px <- y.dist*px.per.xy[2]
c.px <- sqrt(a.px^2+b.px^2)
```

To rotate a string to match the slope of a line segment, you need to convert the distances to pixels, calculate the arctangent, and convert from radians to degrees.

```
segments(x0, y0, x1, y1)
delta.x <- (x1 – x0) * px.per.xy[1]
delta.y <- (y1 – y0) * px.per.xy[y]
angle.radians <- atan2(delta.y, delta.x)
angle.degrees <- angle.radians * 180 / pi
text(x1, y1, "TEXT", srt=angle.degrees)
```

Panels

par("fig") (width, height) in pct
par("fin") (width, height) in inches

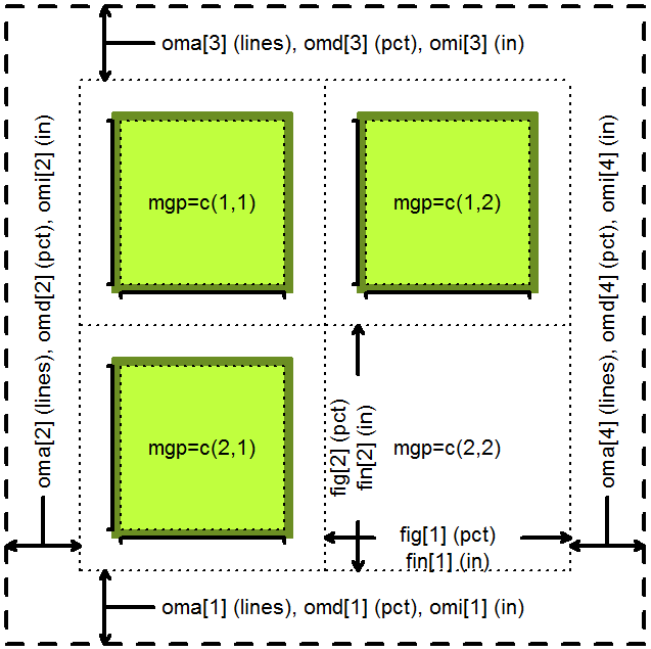
If you display multiple plots within a single graphics window (e.g., with the **mfrow** or **mfcol** arguments of **par** or with the **layout** function), then the **fig** and **fin** arguments will tell you the size of the current subplot window in percent or inches, respectively.

par("oma") (bottom, left, top, right) in lines
par("omd") (bottom, left, top, right) in pct
par("omi") (bottom, left, top, right) in inches

Each subplot will have margins specified by **mai** or **mar**, but no outer margin around the entire set of plots, unless you specify them using **oma**, **omd**, or **omi**. You can place text in the outer margins using the **mtext** function with the argument **outer=TRUE**.

par("mgp") (r, c) or (r, c, maxr, maxc)

The **mgp** argument of **par** will allow you to jump to a subplot in a particular row and column. If you query with **par("mfg")**, you will get the current row and column followed by the maximum row and column.



Character and string sizes

strheight()

The **strheight** functions will tell you the height of a specified string in inches (**units="inches"**), x-y user coordinates (**units="user"**) or as a percentage of the graphics device (**units="figure"**).

For a single line of text, **strheight** will give you the height of the letter "M". If you have a string with one of more linebreaks ("\n"), the **strheight** function will measure the height of the letter "M" plus the height of one or more additional lines. The height of a line is dependent on the line spacing, set by the **lheight** argument of **par**. The default line height (**lheight=1**), corresponding to single spaced lines, produces a line height roughly 1.5 times the height of "M".

strwidth()

The **strwidth** function will produce different widths to individual characters, representing the proportional spacing used by most fonts (a "W" using much more space than an "i"). For the width of a string, the **strwidth** function will sum up the lengths of the individual characters in the string.

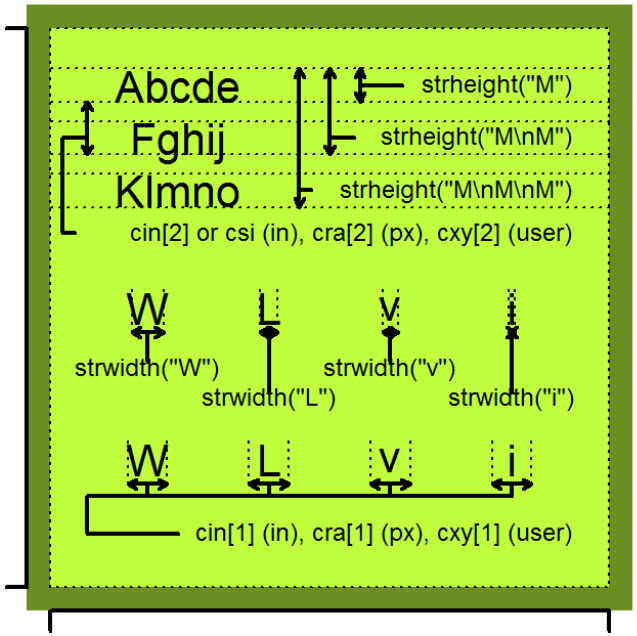
par("cin") *(r.o.)* (width, height) in inches
par("csi") *(r.o.)* height in inches
par("cra") *(r.o.)* (width, height) in pixels
par("cxy") *(r.o.)* (width, height) in xy coordinates

The single value returned by the **csi** argument of **par** gives you the height of a line of text in inches. The second of the two values returned by **cin**, **cra**, and **cxy** gives you the height of a line, in inches, pixels, or xy (user) coordinates.

The first of the two values returned by the **cin**, **cra**, and **cxy** arguments to **par** gives you the approximate width of a single character, in inches, pixels, or xy (user) coordinates. The width, very slightly smaller than the actual width of the letter "W", is a rough estimate at best and ignores the variable with of individual letters.

These values are useful, however, in providing fast ratios of the relative sizes of the differing units of measure

px.per.in <- par("cra") / par("cin")
px.per.xy <- par("cra") / par("cxy")
xy.per.in <- par("cxy") / par("cin")



If your fonts are too big or too small

Fixing this takes a bit of trial and error.

1. Specify a larger/smaller value for the **pointsize** argument when you open your graphics device.

2. Trying opening your graphics device with different values for **height** and **width**. Fonts that look too big might be better proportioned in a larger graphics window.

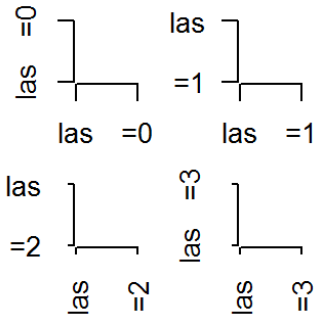
3. Use the **cex** argument to increase or decrease the relative size of your fonts.

If your axes don't fit

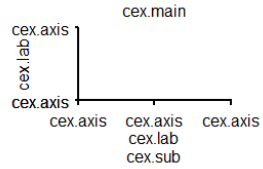
There are several possible solutions.

1. You can assign wider margins using the **mar** or **mai** argument in **par**.

2. You can change the orientation of the axis labels with **las**. Choose among
   a. **las=0** both axis labels parallel
   b. **las=1** both axis labels horizontal
   c. **las=2** both axis labels perpendicular
   d. **las=3** both axis labels vertical.



3. change the relative size of the font
   a. **cex.axis** for the tick mark labels.
   b. **cex.lab** for **xlab** and **ylab**.
   c. **cex.main** for the main title
   d. **cex.sub** for the subtitle.



Basic Regular Expressions in R
Cheat Sheet

Character Classes

[[:digit:]] or \\d	Digits; `[0-9]`
\\D	Non-digits; `[^0-9]`
[[:lower:]]	Lower-case letters; `[a-z]`
[[:upper:]]	Upper-case letters; `[A-Z]`
[[:alpha:]]	Alphabetic characters; `[A-z]`
[[:alnum:]]	Alphanumeric characters `[A-z0-9]`
\\w	Word characters; `[A-z0-9_]`
\\W	Non-word characters
[[:xdigit:]] or \\x	Hexadec. digits; `[0-9A-Fa-f]`
[[:blank:]]	Space and tab
[[:space:]] or \\s	Space, tab, vertical tab, newline, form feed, carriage return
\\S	Not space; `[^[:space:]]`
[[:punct:]]	Punctuation characters; `!"#$%&'()*+,-./:;<=>?@[]^_`{|}~`
[[:graph:]]	Graphical char.; `[[:alnum:][:punct:]]`
[[:print:]]	Printable characters; `[[:alnum:][:punct:]\\s]`
[[:cntrl:]] or \\c	Control characters; `\n, \r` etc.

Special Metacharacters

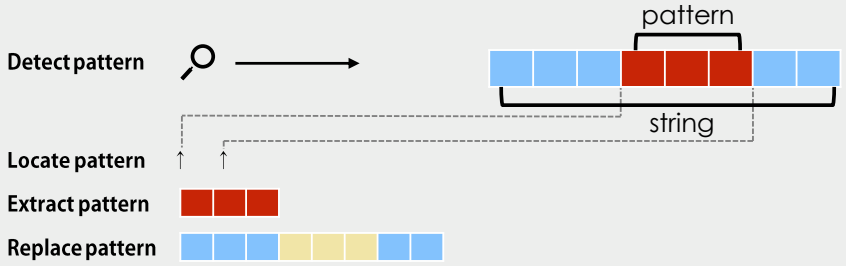
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

Lookaraounds and Conditionals*

(?=)	Lookahead (requires PERL = TRUE), e.g. `(?=yx)`: position followed by `'xy'`
(?!)	Negative lookahead (PERL = TRUE); position NOT followed by pattern
(?<=)	Lookbehind (PERL = TRUE), e.g. `(?<=yx)`: position following `'xy'`
(?<!)	Negative lookbehind (PERL = TRUE); position NOT following pattern
?(if)then	If-then-condition (PERL = TRUE); use lookaheads, optional char. etc in if-clause
?(if)then|else	If-then-else-condition (PERL = TRUE)

*see, e.g. http://www.regular-expressions.info/lookaround.html
http://www.regular-expressions.info/conditional.html

Functions for Pattern Matching




```
> string <- c("Hiphopopotamus", "Rhymenoceros", "time for bottomless lyrics")
> pattern <- "t.m"
```

Detect Patterns

grep(pattern, string)
```
[1] 1 3
```
grep(pattern, string, value = TRUE)
```
[1] "Hiphopopotamus"
[2] "time for bottomless lyrics"
```
grepl(pattern, string)
```
[1]  TRUE FALSE  TRUE
```
stringr::str_detect(string, pattern)
```
[1]  TRUE FALSE  TRUE
```

Locate Patterns

regexpr(pattern, string)
find starting position and length of first match

gregexpr(pattern, string)
find starting position and length of all matches

stringr::**str_locate(string, pattern)**
find starting and end position of first match

stringr::**str_locate_all(string, pattern)**
find starting and end position of all matches

Split a String using a Pattern

strsplit(string, pattern) or stringr::**str_split(string, pattern)**

Extract Patterns

regmatches(string, regexpr(pattern, string))
extract first match
```
[1] "tam" "tim"
```
regmatches(string, gregexpr(pattern, string))
extracts all matches, outputs a list
```
[[1]] "tam" [[2]] character(0) [[3]] "tim" "tom"
```
stringr::**str_extract(string, pattern)**
extract first match
```
[1] "tam" NA  "tim"
```
stringr::**str_extract_all(string, pattern)**
extract all matches, outputs a list

stringr::**str_extract_all(string, pattern, simplify = TRUE)**
extract all matches, outputs a matrix

stringr::**str_match(string, pattern)**
extract first match + individual character groups

stringr::**str_match_all(string, pattern)**
extract all matches + individual character groups

Replace Patterns

sub(pattern, replacement, string)
replace first match

gsub(pattern, replacement, string)
replace all matches

stringr::**str_replace(string, pattern, replacement)**
replace first match

stringr::**str_replace_all(string, pattern, replacement)**
replace all matches

Character Classes and Groups

.	Any character except \n
|	Or, e.g. `(a|b)`
[…]	List permitted characters, e.g. `[abc]`
[a-z]	Specify character ranges
[^…]	List excluded characters
(…)	Grouping, enables back referencing using `\\N` where `N` is an integer

Anchors

^	Start of the string
$	End of the string
\\b	Empty string at either edge of a word
\\B	NOT the edge of a word
\\<	Beginning of a word
\\>	End of a word

Quantifiers

*	Matches at least 0 times
+	Matches at least 1 time
?	Matches at most 1 time; optional string
{n}	Matches exactly n times
{n,}	Matches at least n times
{,n}	Matches at most n times
{n,m}	Matches between n and m times

General Modes

By default R uses *POSIX extended regular expressions*. You can switch to *PCRE regular expressions* using `PERL = TRUE` for base or by wrapping patterns with `perl()` for stringr.

All functions can be used with literal searches using `fixed = TRUE` for base or by wrapping patterns with `fixed()` for stringr.

All base functions can be made case insensitive by specifying `ignore.cases = TRUE`.

Escaping Characters

Metacharacters (`. * +` etc.) can be used as literal characters by escaping them. Characters can be escaped using `\\` or by enclosing them in `\\Q...\\E`.

Case Conversions

Regular expressions can be made case insensitive using `(?i)`. In backreferences, the strings can be converted to lower or upper case using `\\L` or `\\U` (e.g. `\\L\\1`). This requires `PERL = TRUE`.

Greedy Matching

By default the asterisk * is greedy, i.e. it always matches the longest possible string. It can be used in lazy mode by adding `?`, i.e. `*?`.

Greedy mode can be turned off using `(?U)`. This switches the syntax, so that `(?U)a*` is lazy and `(?U)a*?` is greedy.

Note

Regular expressions can conveniently be created using rex::**rex()**.

Updated: 09/16

R For Data Science *Cheat Sheet*

xts

Learn R for data science Interactively at www.DataCamp.com

xts

eXtensible Time Series (xts) is a powerful package that provides an extensible time series class, enabling uniform handling of many R time series classes by extending zoo.

Load the package as follows:

```
> library(xts)
```

xts Objects

xts objects have three main components:

- **coredata**: always a matrix for xts objects, while it could also be a vector for zoo objects
- **index**: vector of any `Date`, `POSIXct`, `chron`, `yearmon`, `yearqtr`, or `DateTime` classes
- **xtsAttributes**: arbitrary attributes

Creating xts Objects

```
> xts1 <- xts(x=1:10, order.by=Sys.Date()-1:10)
> data <- rnorm(5)
> dates <- seq(as.Date("2017-05-01"),length=5,by="days")
> xts2 <- xts(x=data, order.by=dates)
> xts3 <- xts(x=rnorm(10),
              order.by=as.POSIXct(Sys.Date()+1:10),
              born=as.POSIXct("1899-05-08"))
> xts4 <- xts(x=1:10, order.by=Sys.Date()+1:10)
```

Convert To And From xts

```
> data(AirPassengers)
> xts5 <- as.xts(AirPassengers)
```

Import From Files

```
> dat <- read.csv(tmp_file)
> xts(dat, order.by=as.Date(rownames(dat),"%m/%d/%Y"))
> dat_zoo <- read.zoo(tmp_file,
                      index.column=0,
                      sep=",",
                      format="%m/%d/%Y")
> dat_zoo <- read.zoo(tmp,sep=",",FUN=as.yearmon)
> dat_xts <- as.xts(dat_zoo)
```

Export xts Objects

```
> data_xts <- as.xts(matrix)
> tmp <- tempfile()
> write.zoo(data_xts,sep=",",file=tmp)
```

Replace & Update

```
> xts2[dates] <- 0
> xts5["1961"] <- NA
> xts2["2016-05-02"] <- NA
```

Replace values in `xts2` on `dates` with `0`
Replace dates from `1961` with `NA`
Replace the value at 1 specific index with `NA`

Applying Functions

```
> ep1 <- endpoints(xts4,on="weeks",k=2)
[1]  0  5 10
> ep2 <- endpoints(xts5,on="years")
[1]   0  12  24  36  48  60  72  84  96 108 120 132 144
> period.apply(xts5,INDEX=ep2,FUN=mean)
> xts5_yearly <- split(xts5,f="years")
> lapply(xts5_yearly,FUN=mean)
> do.call(rbind,
          lapply(split(xts5,"years"),
          function(w) last(w,n="1 month")))
> do.call(rbind,
          lapply(split(xts5,"years"),
          cumsum))
> rollapply(xts5, 3, sd)
```

Take index values by time

Calculate the yearly mean
Split `xts5` by year
Create a list of yearly means
Find the last observation in each year in `xts5`

Calculate cumulative annual passengers

Apply sd to rolling margins of `xts5`

Selecting, Subsetting & Indexing

Select

```
> mar55 <- xts5["1955-03"]
```

Get value for March 1955

Subset

```
> xts5_1954 <- xts5["1954"]
> xts5_janmarch <- xts5["1954/1954-03"]
> xts5_janmarch <- xts5["/1954-03"]
> xts4[ep1]
```

Get all data from 1954
Extract data from Jan to March '54
Get all data until March '54
Subset `xts4` using `ep2`

`first()` and `last()`

```
> first(xts4,'1 week')
> first(last(xts4,'1 week'),'3 days')
```

Extract first 1 week
Get first 3 days of the last week of data

Indexing

```
> xts2[index(xts3)]
> days <- c("2017-05-03","2017-05-23")
> xts3[days]
> xts2[as.POSIXct(days,tz="UTC")]
> index <- which(.indexwday(xts1)==0|.indexwday(xts1)==6)
> xts1[index]
```

Extract rows with the index of `xts3`

Extract rows using the vector `days`
Extract rows using `days` as POSIXct
Index of weekend days
Extract weekend days of `xts1`

Missing Values

```
> na.omit(xts5)
> xts_last <- na.locf(xts2)

> xts_last <- na.locf(xts2,
                      fromLast=TRUE)
> na.approx(xts2)
```

Omit NA values in `xts5`
Fill missing values in `xts2` using last observation
Fill missing values in `xts2` using next observation
Interpolate NAs using linear approximation

Arithmetic Operations

`coredata()` or `as.numeric()`

```
> xts3 + as.numeric(xts2)
> xts3 * as.numeric(xts4)
> coredata(xts4) - xts3
> coredata(xts4) / xts3
```

Addition
Multiplication
Subtraction
Division

Shifting Index Values

```
> xts5 - lag(xts5)
> diff(xts5,lag=12,differences=1)
```

Period-over-period differences
Lagged differences

Reindexing

```
> xts1 + merge(xts2,index(xts1),fill=0)
                   e1
2017-05-04 5.231538
2017-05-05 5.829257
2017-05-06 4.000000
2017-05-07 3.000000
2017-05-08 2.000000
2017-05-09 1.000000
> xts1 - merge(xts2,index(xts1),fill=na.locf)
                   e1
2017-05-04 5.231538
2017-05-05 5.829257
2017-05-06 4.829257
2017-05-07 3.829257
2017-05-08 2.829257
2017-05-09 1.829257
```

Addition

Subtraction

Merging

```
> merge(xts2,xts1,join='inner')
                   xts2 xts1
2017-05-05 -0.8382068   10

> merge(xts2,xts1,join='left',fill=0)
                    xts2 xts1
2017-05-01  1.7482704    0
2017-05-02 -0.2314678    0
2017-05-03  0.1685517    0
2017-05-04  1.1685649    0
2017-05-05 -0.8382068   10

> rbind(xts1, xts4)
```

Inner join of `xts2` and `xts1`

Left join of `xts2` and `xts1`, fill empty spots with `0`

Combine `xts1` and `xts4` by rows

Inspect Your Data

```
> core_data <- coredata(xts2)
> index(xts1)
```

Extract core data of objects
Extract index of objects

Class Attributes

```
> indexClass(xts2)
> indexClass(convertIndex(xts,'POSIXct'))
> indexTZ(xts5)
> indexFormat(xts5) <- "%Y-%m-%d"
```

Get index class
Replacing index class
Get index class
Change format of time display

Time Zones

```
> tzone(xts1) <- "Asia/Hong_Kong"
> tzone(xts1)
```

Change the time zone
Extract the current time zone

Periods, Periodicity & Timestamps

```
> periodicity(xts5)
> to.yearly(xts5)
> to.monthly(xts3)
> to.quarterly(xts5)
> to.period(xts5,period="quarters")
> to.period(xts5,period="years")
> nmonths(xts5)
> nquarters(xts5)
> nyears(xts5)
> make.index.unique(xts3,eps=1e-4)
> make.index.unique(xts3,drop=TRUE)
> align.time(xts3,n=3600)
```

Estimate frequency of observations
Convert `xts5` to yearly OHLC
Convert `xts3` to monthly OHLC
Convert `xts5` to quarterly OHLC
Convert to quarterly OHLC
Convert to yearly OHLC
Count the months in `xts5`
Count the quarters in `xts5`
Count the years in `xts5`
Make index unique
Remove duplicate times
Round index time to the next `n` seconds

Other Useful Functions

```
> .index(xts4)
> .indexwday(xts3)

> .indexhour(xts3)
> start(xts3)
> end(xts4)
> str(xts3)
> time(xts1)
> head(xts2)
> tail(xts2)
```

Extract raw numeric index of `xts1`
Value of week(day), starting on Sunday, in index of `xts3`
Value of hour in index of `xts3`
Extract first observation of `xts3`
Extract last observation of `xts4`
Display structure of `xts3`
Extract raw numeric index of `xts1`
First part of `xts2`
Last part of `xts2`

The *eurostat* package

R tools to access open data from Eurostat database

Search and download

Data in the Eurostat database is stored in tables. Each table has an identifier, a short table_code, and a description (e.g. *tsdtr420* - People killed in road accidents).
Key eurostat functions allow to find the table_code, download the eurostat table and polish labels in the table.

Find the table code

The **search_eurostat(pattern, ...)** function scans the directory of Eurostat tables and returns codes and descriptions of tables that match pattern.

```
library("eurostat")
query <- search_eurostat("road", type = "table")
query[1:3,1:2]
##                                 title     code
## 1           Goods transport by road ttr00005
## 2   People killed in road accidents tsdtr420
## 3 Enterprises with broadband access tin00090
```

Download the table

The **get_eurostat(id, time_format = "date", filters = "none", type = "code", cache = TRUE, ...)** function downloads the requested table from the *Eurostat bulk download facility* or from *The Eurostat Web Services JSON API* (if **filters** are defined). Downloaded data is cached (if **cache=TRUE**). Additional arguments define how to read the time column (**time_format**) and if table dimensions shall be kept as codes or converted to labels (**type**).

```
dat <- get_eurostat(id="tsdtr420", time_format="num")
head(dat)
##     unit    sex    geo  time values
## 1     NR      T     AT  1999   1079
## 2     NR      T     BE  1999   1397
## 3     NR      T     CZ  1999   1455
## 4     NR      T     DK  1999    514
## 5     NR      T     EL  1999   2116
## 6     NR      T     ES  1999   5738
```

Add labels

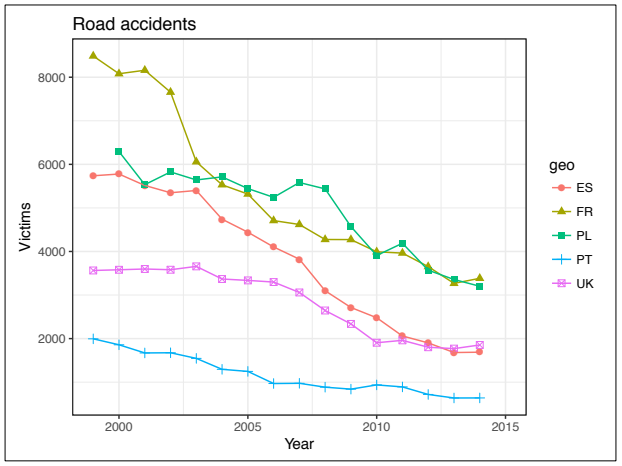
The **label_eurostat(x, lang = "en", ...)** gets definitions for Eurostat codes and replace them with labels in given language ("en", "fr" or "de").

```
dat <- label_eurostat(dat)
head(dat)
##     unit    sex            geo  time values
## 1 Number  Total        Austria  1999   1079
## 2 Number  Total        Belgium  1999   1397
## 3 Number  Total Czech Republic  1999   1455
## 4 Number  Total        Denmark  1999    514
## 5 Number  Total         Greece  1999   2116
## 6 Number  Total          Spain  1999   5738
```

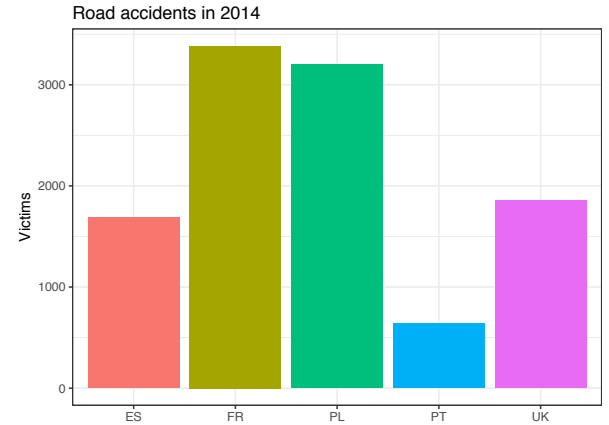
eurostat and plots

The **get_eurostat()** function returns tibbles in the long format. Packages **dplyr** and **tidyr** are well suited to transform these objects. The **ggplot2** package is well suited to plot these objects.

```
t1 <- get_eurostat("tsdtr420", filters =
            list(geo = c("UK", "FR", "PL", "ES", "PT")))

library("ggplot2")
ggplot(t1, aes(x = time, y = values, color = geo,
               group = geo, shape = geo)) +
  geom_point(size = 2) +
  geom_line() + theme_bw() +
  labs(title="Road accidents", x = "Year", y = "Victims")
```




```
library("dplyr")
t2 <- t1 %>% filter(time == "2014-01-01")
ggplot(t2, aes(geo, values, fill=geo)) +
  geom_bar(stat = "identity") + theme_bw() +
  theme(legend.position = "none")+
  labs(title="Road accidents in 2014", x="", y="Victims")
```



eurostat and maps

Fetch and process data

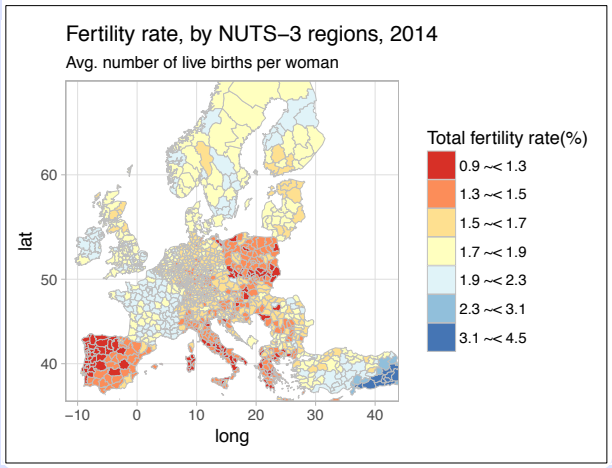
There are three function to work with geospatial data from GISCO. The **get_eurostat_geospatial()** returns preprocessed spatial data as sp-objects or as data frames. The **merge_eurostat_geospatial()** both downloads and merges the geospatial data with a preloaded tabular data. The **cut_to_classes()** is a wrapper for **cut()** - function and is used for categorizing data for maps with tidy labels.

```
library("eurostat")
library("dplyr")

fertility <- get_eurostat("demo_r_frate3")  %>%
  filter(time == "2014-01-01") %>%
  mutate(cat = cut_to_classes(values, n=7, decimals=1))

mapdata <-  merge_eurostat_geodata(fertility,
              resolution = "20")

head(select(mapdata,geo,values,cat,long,lat,order,id))
##     geo values         cat     long      lat order id
## 1 AT124   1.39 1.3 ~< 1.5 15.54245 48.90770   214 10
## 2 AT124   1.39 1.3 ~< 1.5 15.75363 48.85218   215 10
## 3 AT124   1.39 1.3 ~< 1.5 15.88763 48.78511   216 10
## 4 AT124   1.39 1.3 ~< 1.5 15.81535 48.69270   217 10
## 5 AT124   1.39 1.3 ~< 1.5 15.94094 48.67173   218 10
## 6 AT124   1.39 1.3 ~< 1.5 15.90833 48.59815   219 10
```

Draw a cartogram

The object returned by **merge_eurostat_geospatial()** are ready to be plotted with ggplot2 package. The **coord_map()** function is useful to set the projection while **labs()** adds annotations o the plot.

```
library("ggplot2")
ggplot(mapdata, aes(x = long, y = lat, group = group))+
  geom_polygon(aes(fill=cat), color="grey", size = .1)+
  scale_fill_brewer(palette = "RdYlBu") +
  labs(title="Fertility rate, by NUTS-3 regions, 2014",
       subtitle="Avg. number of live births per woman",
       fill="Total fertility rate(%)") + theme_light()+
  coord_map(xlim=c(-12,44), ylim=c(35,67))
```



This onepager presents the **eurostat** package
Leo Lahti, Janne Huovari, Markus Kainu, Przemyslaw Biecek 2014-2017 package version 2.2.43 URL: *https://github.com/rOpenGov/eurostat*

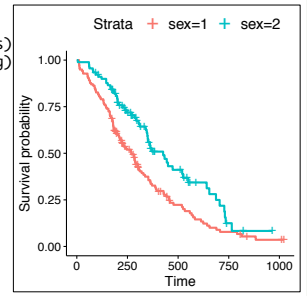
Creating Survival Plots

Informative and Elegant with *survminer*

Survival Curves

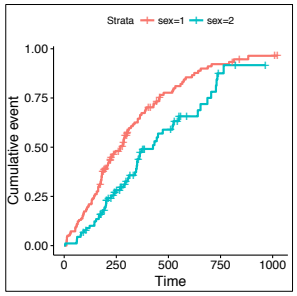
The **ggsurvplot()** function creates **ggplot2** plots from **survfit** objects.

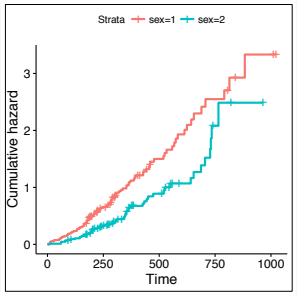
```
library("survival")
fit <- survfit(Surv(time,status)
               ~ sex, data = lung)

class(fit)
## [1] "survfit"

library("survminer")
ggsurvplot(fit, data = lung)
```



Use the **fun** argument to set the transformation of the survival curve. E.g. **"event"** for cumulative events, **"cumhaz"** for the cumulative hazard function or **"pct"** for survival probability in percentage.

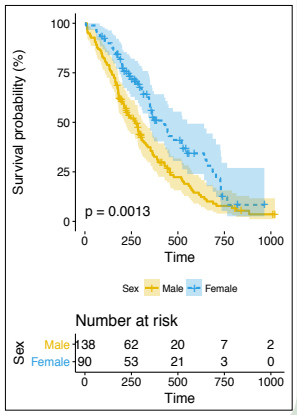
```
ggsurvplot(fit, data = lung, fun = "event")
ggsurvplot(fit, data = lung, fun = "cumhaz")
```





With lots of graphical parameters you have full control over look and feel of the survival plots; position and content of the legend; additional annotations like p-value, title, subtitle.

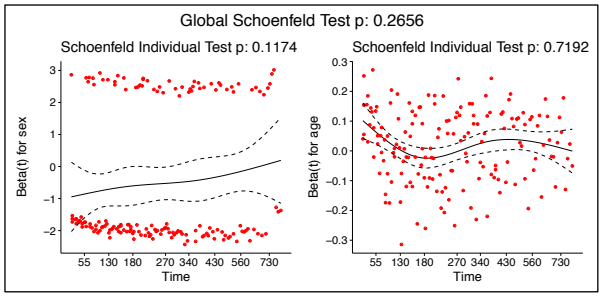
```
ggsurvplot(fit, data = lung,
    conf.int = TRUE,
    pval = TRUE,
    fun = "pct",
    risk.table = TRUE,
    size = 1,
    linetype = "strata",
    palette = c("#E7B800",
                "#2E9FDF"),
    legend = "bottom",
    legend.title = "Sex",
    legend.labs = c("Male",
                    "Female"))
```



Diagnostics of Cox Model

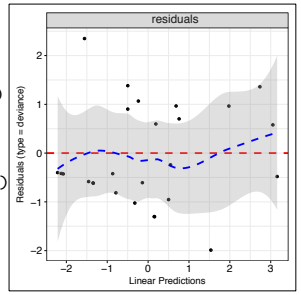
The function **cox.zph()** from **survival** package may be used to test the proportional hazards assumption for a Cox regression model fit. The graphical verification of this assumption may be performed with the function **ggcoxzph()** from the **survminer** package. For each covariate it produces plots with scaled Schoenfeld residuals against the time.

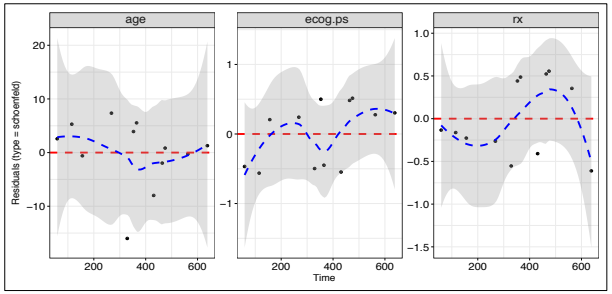
```
library("survival")
fit <- coxph(Surv(time, status) ~ sex + age, data = lung)
ftest <- cox.zph(fit)
ftest
##            rho chisq     p
## sex     0.1236 2.452 0.117
## age    -0.0275 0.129 0.719
## GLOBAL      NA 2.651 0.266
library("survminer")
ggcoxzph(ftest)
```



The function **ggcoxdiagnostics()** plots different types of residuals as a function of time, linear predictor or observation id. The type of residual is selected with **type** argument. Possible values are "martingale", "deviance", "score", "schoenfeld", "dfbeta", "dfbetas", and "scaledsch".
The **ox.scale** argument defines what shall be plotted on the OX axis. Possible values are "linear.predictions", "observation.id", "time".
Logical arguments **hline** and **sline** may be used to add horizontal line or smooth line to the plot.

```
library("survival")
library("survminer")
fit <- coxph(Surv(time, status) ~
             sex + age, data = lung)

ggcoxdiagnostics(fit,
     type = "deviance",
     ox.scale = "linear.predictions")

ggcoxdiagnostics(fit,
     type = "schoenfeld",
     ox.scale = "time")
```

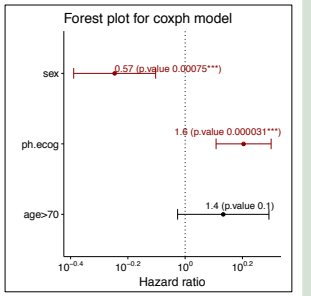




Summary of Cox Model

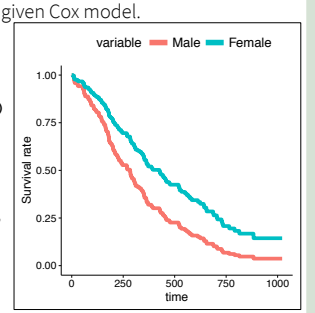
The function **ggforest()** from the **survminer** package creates a forest plot for a Cox regression model fit. Hazard ratio estimates along with confidence intervals and p-values are plotter for each variable.

```
library("survival")
library("survminer")
lung$age <- ifelse(lung$age > 70, ">70","<= 70")
fit <- coxph( Surv(time, status) ~ sex + ph.ecog + age, data = lung)
fit

## Call:
## coxph(formula = Surv(time, status) ~ sex+ph.ecog+age, data=lung)
##
##            coef exp(coef) se(coef)     z       p
## sex     -0.567     0.567    0.168 -3.37 0.00075
## ph.ecog  0.470     1.600    0.113  4.16 3.1e-05
## age>70   0.307     1.359    0.187  1.64 0.10175
##
## Likelihood ratio test=31.6  on
## n= 227, number of events= 164

ggforest(fit)
```



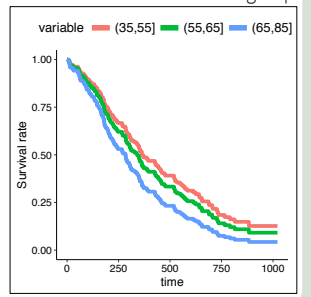
The function **ggcoxadjustedcurves()** from the **survminer** package plots Adjusted Survival Curves for Cox Proportional Hazards Model. Adjusted Survival Curves show how a selected factor influences survival estimated from a Cox model.
Note that these curves differ from Kaplan Meier estimates since they present expected ssurvival based on given Cox model.

```
library("survival")
library("survminer")

lung$sex <- ifelse(lung$sex == 1,
                   "Male", "Female")

fit <- coxph(Surv(time, status) ~
             sex + ph.ecog + age,
             data = lung)

ggcoxadjustedcurves(fit, data=lung,
                    variable=lung$sex)
```



Note that it is not necessary to include the grouping factor in the Cox model. Survival curves are estimated from Cox model for each group defined by the factor independently.

```
lung$age3 <- cut(lung$age,
                 c(35,55,65,85))

ggcoxadjustedcurves(fit, data=lung,
                    variable=lung$age3)
```



This onepager presents the **survminer** package [Alboukadel Kassambara, Marcin Kosinski 2017] in version 0.3.
See *https://github.com/kassambara/survminer/* for more details.

Leaflet Cheat Sheet

Leaflet for R

an open-source JavaScript library for mobile-friendly interactive maps

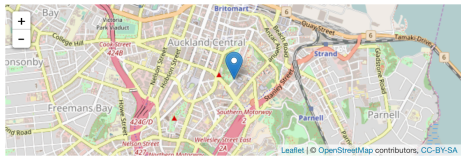
Quick Start

Installation

Use **install.packages("leaflet")** to installize the package or directly from Github *devtools::install_github("rstudio/leaflet")*.

First Map

```
m <- leaflet() %>%        # leaflet works with the pipe operator
  addTiles() %>%          # setup the default OpenStreetMap map tiles
  addMarkers(lng = 174.768, lat = -36.852, popup = "The birthplace of R")
                          # add a single point layer
m
```



Map Widget

Initialization

m <- leaflet(options = leafletOptions(...))

center	Initial geographic center of the map
zoom	Initial map zoom level
minZoom	Minimum zoom level of the map
maxZoom	Maximum zoom level of the map

Map Methods

m %>% setView(lng, lat, zoom, options = list())
Set the view of the map (center and zoom level)

m %>% fitBounds(lng1, lat1, lng2, lat2)
Fit the view into the rectangle [lng1, lat1] - [lng2, lat2]

m %>% clearBounds()
Clear the bound, automatically determine from the map elements

Data Object

Both **leaflet()** and the **map** layers have an optional data parameter that is designed to receive spatial data with the following formats:

Base R — *The arguments of all layers take normal R objects:*

```
df <- data.frame(lat = ..., lng = ... )
leaflet(df) %>% addTiles() %>% addCircles()
```

sp *package* — library(sp) *Useful functions:*
SpatialPoints, SpatialLines, SpatialPolygons, ...

maps *package* — library(maps) *Build a map of states with colors:*

```
mapStates <- map("state", fill = TRUE, plot = FALSE)
leaflet(mapStates) %>% addTiles() %>%
  addPolygons(fillColor = topo.colors(10, alpha =
  NULL), stroke = FALSE)
```

Markers

Use markers to call out points, express locations with latitude/longitude coordinates, appear as icons or as circles.
Data come from vectors or assigned data frame, or **sp** package objects.

Icon Markers

Regular Icons: default and simple

addMarkers(lng, lat, popup, label) *add basic icon markers*

makeIcon/icons (iconUrl, iconWidth, iconHeight, iconAnchorX, iconAnchorY, shadowUrl, shadowWidth, shadowHeight, ...) *customize marker icons*

iconList() *create a list of icons*

Awesome Icons: customizable with colors and icons

addAwesomeMarkers, makeAwesomeIcon, awesomeIcons, awesomeIconList

Marker Clusters: option of addMarters()

clusterOptions = markerClusterOptions()

freezeAtZoom Freeze the cluster at assigned zoom level

Circle Markers

addCircleMarkers(color, radius, stroke, opacity, ...)
Customize their color, radius, stroke, opacity

Popups and Labels

addPopups(lng, lat, ...content..., options) *Add standalone popups*
options = popupOptions(closeButton=FALSE)

addMarkers(..., popup, ...) *Show popups with markers or shapes*

addMarkers(..., label, labelOptions...) *Show labels with markers or shapes*
labelOptions = labelOptions(noHide, textOnly, textsize, direction, style)

addLabelOnlyMarkers() *Add labels without markers*

Lines and Shapes

Polygons and Polylines

addPolygons(color, weight=1, smoothFactor=0.5, opacity=1.0, fillOpacity=0.5, fillColor= ~colorQuantile("YlOrRd", ALAND)(ALAND), highlightOptions, …)

highlightOptions(color, weight=2, bringToFront=TRUE) *highlight shapes*

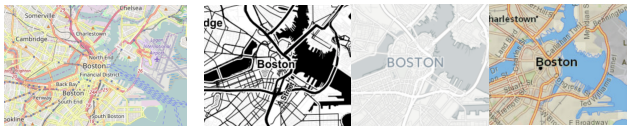
Use **rmapshaper::ms_simplify** to simplify complex shapes

Circles **addCircles**(lng, lat, weight=1, radius, ...)

Rectangles **addRectangles**(lng1, lat1, lng2, lat2, fillColor="transparent", ...)

Basemaps

addTiles() providers$Stamen.Toner, CartoDB.Positron, Esri.NatGeoWorldMap



Default Tiles — Third-Party Tiles **addProviderTiles()**

Use **addTiles()** to add a custom map tile URL template, use **addWMSTiles()** to add WMS (Web Map Service) tiles

GeoJSON and TopoJSON

There are two options to use the GeoJSON/TopoJSON data.

* To read into **sp** objects with the **geojsonio** or **rgdal** package:
geojsonio::geojson_read(...,what="sp") *rgdal::readOGR(...,"OGRGeoJSON")*
* Or to use the **addGeoJSON()** and **addTopoJSON()** functions:
addTopoJSON/addGeoJSON(... weight, color, fill, opacity, fillOpacity...)
Styles can also be tuned separately with a *style: {...}* object.

Other packages including **RJSONIO** and **jsonlite** can help fast parse or generate the data needed.

Shiny Integration

To integrate a Leaflet map into an app:
* In the UI, call **leafletOutput**("name")
* On the server side, assign a **renderLeaflet**(...) call to the output
* Inside the **renderLeaflet** expression, return a Leaflet map object

Modification

To modify an existing map or add incremental changes to the map, you can use **leafletProxy()**. This should be performed in an observer on the server side.

Other useful functions to edit your map:

fitBounds(0, 0, 11, 11) *similar to setView*
fit the view to within these bounds

addCircles(1:10, 1:10, layerId = LETTERS[1:10])
create circles with layerIds of "A", "B", "C"...

removeShape(c("B", "F")) *remove some of the circles*

clearShapes() *clear all circles (and other shapes)*

Inputs/Events

Object Events

Object event names generally use this pattern:
input$*MAPID_OBJCATEGORY_EVENTNAME*.
Triger an event changes the value of the Shiny input at this variable.
Valid values for *OBJCATEGORY* are *marker*, *shape*, *geojson* and *topojson*.
Valid values for *EVENTNAME* are *click*, *mouseover* and *mouseout*.

All of these events are set to either *NULL* if the event has never happened, or a *list()* that includes:
* lat The latitude of the object, if available; otherwise, the mouse cursor
* lng The longitude of the object, if available; otherwise, the mouse cursor
* id The layerId, if any

GeoJSON events also include additional properties:
* featureId The feature ID, if any
* properties The feature properties

Map Events

input$*MAPID*_click *when the map background or basemap is clicked*
value -- a list with lat and lng

input$*MAPID*_bounds *provide the lat/lng bounds of the visible map area*
value -- a list with north, east, south and west

input$*MAPID*_zoom *an integer indicates the zoom level*

Kejia Shi @ Data Science Institute, Columbia University in the City of New York, Kejia.Shi@Columbia.edu

Adapted from Rstudio materials https://rstudio.github.io/leaflet/ 2017/05/01