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 Libraries

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

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.

Matrixes

`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 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** library.

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

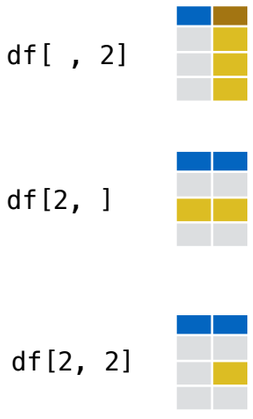
List subsetting

`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, ]`

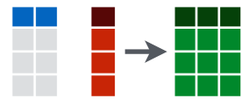
`df[2, 2]`

`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** library.

`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 but 'cutting' into sections.

Statistics

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

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

`summary`
Get more detailed information out a model.

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

`pairwise.t.test`
Preform 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`
Poison	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** library.

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

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

`hist(x)`
Histogram of x.

Dates

See the **lubridate** library.

DATA ANALYSIS THE DATA.TABLE WAY

The official Cheat Sheet for the DataCamp course

General form: `DT[i, j, by]` → "Take `DT`, subset rows using `i`, then calculate `j` grouped by `by`"

CREATE A DATA TABLE

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

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

```
> DT
   V1 V2     V3 V4
1:  1  A -1.1727  1
2:  2  B -1.3825  2
3:  1  C -1.0604  3
4:  2  A  0.6651  4
5:  1  B -1.1727  5
6:  2  C -0.3825  6
7:  1  A -1.0604  7
8:  2  B  0.6651  8
9:  1  C -1.1727  9
10: 2  A -0.3825 10
11: 1  B -1.0604 11
12: 2  C  0.6651 12
```

SUBSETTING ROWS USING `i`

What?	*Example*	*Notes*	*Output*

Subsetting rows by numbers.	`DT[3:5,]` #or `DT[3:5]`	Selects third to fifth row.	`V1 V2 V3 V4`
			`1: 1 C -1.0604 3`
			`2: 2 A 0.6651 4`
			`3: 1 B -1.1727 5`

Use column names to select rows in `i` based on a condition using fast automatic indexing. Or for selecting on multiple values: `DT[column %in% c("value1","value2")]`, which selects all rows that have **value1** or **value2** in **column**.	`DT[ V2 == "A"]`	Selects all rows that have value **A** in column **V2**.	`V1 V2 V3 V4`
			`1: 1 A -1.1727 1`
			`2: 2 A 0.6651 4`
			`3: 1 A -1.0604 7`
			`4: 2 A -0.3825 10`

	`DT[ V2 %in% c("A","C")]`	Select all rows that have the value **A** or **C** in column **V2**.	`V1 V2 V3 V4`
			`1: 1 A -1.1727 1`
			`2: 1 C -1.0604 3`
			`...`
			`7: 2 A -0.3825 10`
			`8: 2 C 0.6651 12`

MANIPULATING ON COLUMNS IN `j`

What?	*Example*	*Notes*	*Output*
Select 1 column in `j`.	`DT[,V2]`	Column **V2** is returned as a vector.	`[1] "A" "B" "C" "A" "B" "C" ...`

Select several columns in `j`.	`DT[,.(V2,V3)]`	Columns **V2** and **V3** are returned as a `data.table`.	`V2 V3`
			`1: A -1.1727`
			`2: B -0.3825`
			`3: C -1.0604`
			`...`

`.()` is an alias to `list()`. If `.()` is used, the returned value is a `data.table`. If `.()` is not used, the result is a vector.

Call functions in `j`.	`DT[,sum(V1)]`	Returns the sum of all elements of column **V1** in a vector.	`[1] 18`
Computing on several columns.	`DT[,.(sum(V1),sd(V3))]`	Returns the sum of all elements of column **V1** and the standard deviation of **V3** in a `data.table`.	`V1 V2` `1: 18 0.7634655`
Assigning column names to computed columns.	`DT[,.(Aggregate = sum(V1), Sd.V3 = sd(V3))]`	The same as above, but with new names.	`Aggregate Sd.V3` `1: 18 0.7634655`

Columns get recycled if different length.	`DT[,.(V1, Sd.V3 = sd(V3))]`	Selects column **V1**, and compute std. dev. of **V3**, which returns a single value and gets recycled.	`V1 Sd.V3`
			`1: 1 0.7634655`
			`2: 2 0.7634655`
			`...`
			`11: 1 0.7634655`
			`12: 2 0.7634655`

Multiple expressions can be wrapped in curly braces.	`DT[,{print(V2) plot(V3) NULL}]`	`Print` column **V2** and plot **V3**.	`[1] "A" "B" "C" "A" "B" "C" ...` `#And a plot`

DOING `j` BY `BY` GROUP

What?	*Example*	*Notes*	*Output*
Doing `j` by group.	`DT[,.(V4.Sum = sum(V4)),by=V1]`	Calculates the sum of **V4**, for every group in **V1**.	`V1 V4.Sum` `1: 1 36`

Doing `j` by several groups using `.()`.	`DT[,.(V4.Sum = sum(V4)),by=.(V1,V2)]`	The same as above, but for every group in **V1** and **V2**.	`V1 V2 V4.Sum`
			`1: 1 A 10`
			`2: 2 B 10`
			`3: 1 C 16`
			`4: 2 A 14`
			`5: 1 B 10`
			`6: 2 C 18`

Call functions in by.	`DT[,.(V4.Sum = sum(V4)),by=sign(V1-1)]`	Calculates the sum of **V4**, for every group in **sign (V1-1)**.	`sign V4.Sum`
			`1: 0 36`
			`2: 1 42`

Assigning new column names in by.	`DT[,.(V4.Sum = sum(V4)), by=.(V1.01 = sign(V1-1))]`	Same as above, but with a new name for the variable we are grouping by.	`V1.01 V4.Sum`
			`1: 0 36`
			`2: 1 42`

Grouping only on a subset by specifying `i`.	`DT[1:5,.(V4.Sum = sum(V4)),by=V1]`	Calculates the sum of **V4**, for every group in **V1**, after subsetting on the first five rows.	`V1 V4.Sum`
			`1: 1 9`
			`2: 2 6`

Using `.N` to get the total number of observations of each group.	`DT[,.N,by=V1]`	Count the number of rows for every group in **V1**.	`V1 N`
			`1: 1 6`
			`2: 2 6`

ADDING/UPDATING COLUMNS BY REFERENCE IN `j` USING `:=`

What?	*Example*	*Notes*	*Output*
Adding/updating a column by reference using `:=` in one line. Watch out: extra assignment (`DT <- DT[...]`) is redundant.	`DT[, V1 := round(exp(V1), 2)]`	Column **V1** is updated by what is after `:=`.	Returns the result invisibly. Column **V1** went from: `[1] 1 2 1 2 1 2` to `[1] 2.72 7.39 2.72 7.39 ...`
Adding/updating several columns by reference using `:=`.	`DT[, c("V1","V2") := list(round(exp(V1),2), LETTERS[1:12])]`	Column **V1** and **V2** are updated by what is after `:=`.	Returns the result invisibly. Column **V1** changed as above. Column **V2** went from: `[1] "A" "B" "C" "A" "B" "C" ...` to: `[1] "A" "B" "C" "D" "E" "F" ...`
Using functional `:=`.	`DT[, `:=`( V1 = round(exp(V1),2), V2 = LETTERS[4:15]) ]`	Another way to write the same line as above this one, but easier to write comments side-by-side. Also, when `[]` is added the result is printed to the screen.	Same changes as line above this one, but the result is printed to the screen because of the `[]` at the end of the statement.
Remove a column instantly using `:=`.	`DT[, V1 := NULL]`	Removes column **V1**.	Returns the result invisibly. Column **V1** became `NULL`.
Remove several columns instantly using `:=`.	`DT[, c("V1","V2") := NULL]`	Removes columns **V1** and **V2**.	Returns the result invisibly. Columns **V1** and **V2** became `NULL`.
Wrap the name of a variable which contains column names in parenthesis to pass the contents of that variable to be deleted.	`Cols.chosen = c("A","B")` `DT[, Cols.chosen := NULL]`	Watch out: this deletes the column with column name `Cols.chosen`.	Returns the result invisibly. Column with name `Cols.chosen` became `NULL`.
	`DT[, (Cols.chosen) := NULL]`	Deletes the columns specified in the variable `Cols.chosen` (**V1** and **V2**).	Returns the result invisibly. Columns **V1** and **V2** became `NULL`.

INDEXING AND KEYS

What?	*Example*	*Notes*	*Output*
Use `setkey()` to set a key on a `DT`. The data is sorted on the column we specified by reference.	`setkey(DT,V2)`	A key is set on column **V2**.	Returns results invisibly.

Use keys like supercharged rownames to select rows.	`DT["A"]`	Returns all the rows where the key column (set to column **V2** in the line above) has the value **A**.	`V1 V2 V3 V4`
			`1: 1 A -1.1727 1`
			`2: 2 A 0.6651 4`
			`3: 1 A -1.0604 7`
			`4: 2 A -0.3825 10`

	`DT[c("A","C")]`	Returns all the rows where the key column (**V2**) has the value **A** or **C**.	`V1 V2 V3 V4`
			`1: 1 A -1.1727 1`
			`2: 2 A 0.6651 4`
			`...`
			`5: 1 C -1.1727 9`
			`6: 2 C 0.6651 12`

The `mult` argument is used to control which row that `i` matches to is returned, default is all.	`DT["A", mult ="first"]`	Returns first row of all rows that match the value **A** in the key column (**V2**).	`V1 V2 V3 V4` `1: 1 A -1.1727 1`
	`DT["A", mult = "last"]`	Returns last row of all rows that match the value **A** in the key column (**V2**).	`V1 V2 V3 V4` `1: 2 A -0.3825 10`

The `nomatch` argument is used to control what happens when a value specified in `i` has no match in the rows of the `DT`. Default is **NA**, but can be changed to **0**. **0** means no rows will be returned for that non-matched row of `i`.	`DT[c("A","D"), nomatch = 0]`	Returns all the rows where the key column (**V2**) has the value **A** or **D**. Value **D** is not found and not returned because of the `nomatch` argument.	`V1 V2 V3 V4`
			`1: 1 A -1.1727 1`
			`2: 2 A 0.6651 4`
			`3: 1 A -1.0604 7`
			`4: 2 A -0.3825 10`

`by=.EACH!` allows to group by each subset of known groups in `i`. A key needs to be set to use `by=.EACH!`.	`DT[c("A","C"), sum(V4)]`	Returns one total sum of column **V4**, for the rows of the key column (**V2**) that have values **A** or **C**.	`[1] 52`

	`DT[c("A","C"), sum(V4), by=.EACH!]`	Returns one sum of column **V4** for the rows of column **V2** that have value **A**, and another sum for the rows of column **V2** that have value **C**.	`V2 V1`
			`1: A 72`
			`2: C 30`

Any number of columns can be set as key using `setkey()`. This way rows can be selected on 2 keys which is an equijoin.	`setkey(DT,V1,V2)`	Sorts by column **V1** and then by column **V2** within each group of column **V1**.	Returns results invisibly.

	`DT[.(1,"C")]`	Selects the rows that have the value **2** for the first key (column **V1**) and the value **C** for the second key (column **V2**).	`V1 V2 V3 V4`
			`1: 2 C -0.3825 6`
			`2: 2 C 0.6651 12`

	`DT[.(1, c("A","C"))]`	Selects the rows that have the value **2** for the first key (column **V1**) and within those rows the value **A** or **C** for the second key (column **V2**).	`V1 V2 V3 V4`
			`1: 2 A 0.6651 4`
			`2: 2 A -0.3825 10`
			`3: 2 C -0.3825 6`
			`4: 2 C 0.6651 12`

ADVANCED DATA TABLE OPERATIONS

What?	*Example*	*Notes*	*Output*
`.N` contains the number of rows or the last row.	Usable in `i`: `DT[.N-1]`	Returns the penultimate row of the `data.table`.	`V1 V2 V3 V4` `1: 1 B -1.0604 11`
	Usable in `j`: `DT[,.N]`	Returns the number of rows.	`[1] 12`

`.()` is an alias to `list()` and means the same. The `.()` notation is not needed when there is only one item in `by` or `j`.	Usable in `j`: `DT[,.(V2,V3)]` #or `DT[,list(V2,V3)]`	Columns **V2** and **V3** are returned as a `data.table`.	`V2 V3`
			`1: A -1.1727`
			`2: B -0.3825`
			`3: C -1.0604`
			`...`

	Usable in `by`: `DT[, .(mean(V3)), by=.(V1,V2)]`	Returns the result of `j`, grouped by all possible combinations of groups specified in `by`.	`V1 V2 V1`
			`1: 1 A -1.11655`
			`2: 2 B 0.14130`
			`3: 1 C -1.11655`
			`4: 2 A 0.14130`
			`5: 1 B -1.11655`
			`6: 2 C 0.14130`

`.SD` is a `data.table` and holds all the values of all columns, except the one specified in `by`. It reduces programming time but keeps readability. `.SD` is only accessible in `j`.	`DT[, print(.SD), by=V2]`	To look at what `.SD` contains.	`#All of .SD (output too long to display here)`

	`DT[,.SD[c(1,.N)], by=V2]`	Selects the first and last row grouped by column **V2**.	`V2 V1 V3 V4`
			`1: A 1 -1.1727 1`
			`2: A 2 -0.3825 10`
			`3: B 1 -1.0604 11`
			`4: B 2 0.6651 8`
			`5: C 1 -1.0604 3`
			`6: C 2 0.6651 12`

	`DT[, lapply(.SD, sum), by=V2]`	Calculates the sum of all columns in `.SD` grouped by **V2**.	`V2 V1 V3 V4`
			`1: A 4 -1.9505 22`
			`2: B 4 -1.9505 24`
			`3: C 4 -1.9505 30`

`.SDcols` is used together with `.SD`, to specify a subset of the columns of `.SD` to be used in `j`.	`DT[, lapply(.SD,sum), by=V2, .SDcols = c("V3","V4")]`	Same as above, but only for columns **V3** and **V4** of `.SD`.	`V2 V3 V4`
			`1: A -1.9505 22`
			`2: B -1.9505 24`
			`3: C -1.9505 30`

`.SDcols` can be the result of a function call.	`DT[, lapply(.SD,sum), by=V2, .SDcols = paste0("V",3:4)]`	Same result as the line above.	

CHAINING HELPS TACK EXPRESSIONS TOGETHER AND AVOID (UNNECESSARY) INTERMEDIATE ASSIGNMENTS

What?	*Example*	*Notes*	*Output*

Do `Z` (or more) sets of statements at once by chaining them in one statement. This corresponds to *having* in SQL.	`DT[,.(V4.Sum = sum(V4)),by=V1][V4.Sum > 40] #no chaining`
	`DT[, .(V4.Sum = sum(V4)), by=V1][V4.Sum > 40 ]`
		First calculates sum of **V4**, grouped by **V1**. Then selects that group of which the sum is > 40 without chaining.
		Same as above, but with chaining.
			`V1 V4.Sum`
			`1: 2 42`
			`V1 V4.Sum`
			`1: 2 42`

Order the results by chaining.	`DT[, .(V4.Sum = sum(V4)), by=V1][order(-V1)]`	Calculates sum of **V4**, grouped by **V1**, and then `order`s the result on **V1**.	`V1 V4.Sum`
			`1: 2 42`
			`2: 1 36`

USING THE `set()`-FAMILY

What?	*Example*	*Notes*	*Output*

`set()` is used to repeatedly update rows and columns by reference. `Set()` is a loopable low overhead version of `:=`. Watch out: It can not handle grouping operations.	Syntax of `set()`: `for (i in from:to) set(DT, row, column, new value)`.
	`rows = list(1:5,5:1)`
	`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**.	Returns the result invisibly.
			`> DT`
			`V1 V2 V3 V4`
			`1: 1 A -1.1727 1`
			`2: 2 B -0.3825 2`
			`3: NA C -1.0604 3`
			`4: NA A 0.6651 4`
			`5: 1 NA -1.1727 5`
			`6: 2 NA -0.3825 6`
			`7: 1 A -1.0604 7`
			`8: 2 B 0.6651 8`

`setnames()` is used to create or update column names by reference.	Syntax of `setnames()`: `setnames(DT,"oldi","new")` [1]
	`setnames(DT,"V2","Rating")`
	`setnames(DT,c("V2","V3"), c("V2.rating","V3.DataCamp"))`
		Changes (set) the name of column **oldi** to **new**. Also, when `[]` is added at the end of any set`()` function the result is printed to the screen.
		Sets the name of column **V2** to **Rating**.
		Changes two column names.
			Returns the result invisibly.
			Returns the result invisibly.
			Returns the result invisibly.

`setcolorder()` is used to reorder columns by reference.	`setcolorder(DT, "neworder")` `setcolorder(DT, c("V2","V1","V4","V3"))`	**neworder** is a character vector of the new column name ordering. Changes the column ordering to the contents of the vector.	Returns the result invisibly. The new column order is now `[1] "V2" "V1" "V4" "V3"`