Contents

[Namespaces 5](#_Toc14692419)

[R will follow a certain order when searching for a function in these *namespaces*. 5](#_Toc14692420)

[You can see the order by typing: search() 5](#_Toc14692421)

[You can force the use of a specific name space by using double colons (::) like this: 5](#_Toc14692422)

[dplyr::filter 5](#_Toc14692423)

[Packages 6](#_Toc14692424)

[The purrr package 6](#_Toc14692425)

[Data frame 7](#_Toc14692426)

[Manipulating data frames 8](#_Toc14692427)

[mutate : Adding a column with mutate 8](#_Toc14692428)

[filter : Subsetting with filter 8](#_Toc14692429)

[select : Selecting columns with select 8](#_Toc14692430)

[e.g: 9](#_Toc14692431)

[murders <- **mutate**(murders, population\_in\_millions = population **/** 10**^**6) 9](#_Toc14692432)

[**select**(murders, state, population) **%>% head**() 9](#_Toc14692433)

[no\_florida <- **filter**(murders, state **!=** "Florida") 9](#_Toc14692434)

[**filter**(murders, state **%in% c**("New York", "Texas")) 9](#_Toc14692435)

[**filter**(murders, population **<** 5000000 **&** region **==** "Northeast") 9](#_Toc14692436)

[heights %>% filter(grepl("^male", sex, ignore.case=TRUE)) //case insensitive 9](#_Toc14692437)

[Sorting Data Frame 9](#_Toc14692438)

[Nested Sorting Data Frame 9](#_Toc14692439)

[The top *n* 9](#_Toc14692440)

[tibble 10](#_Toc14692441)

[Tibble vs data frame 10](#_Toc14692442)

[Tibbles display better 10](#_Toc14692443)

[Subsets of tibbles are tibbles 10](#_Toc14692444)

[Tibbles can have complex entries 10](#_Toc14692445)

[Tibbles can be grouped 10](#_Toc14692446)

[do 10](#_Toc14692447)

[The do function understands grouped tibbles and always returns a data frame. 10](#_Toc14692448)

[Vectors: numerics, characters, and logical 11](#_Toc14692449)

[: operator 11](#_Toc14692450)

[Creating a vector using seq() function 11](#_Toc14692451)

[Sometimes it is useful to name the entries of a vector. 11](#_Toc14692452)

[We can also assign names using the names functions: 11](#_Toc14692453)

[Access Elements of a Vector 12](#_Toc14692454)

[Using integer vector as index 12](#_Toc14692455)

[Using logical vector as index 12](#_Toc14692456)

[Using character vector as index 12](#_Toc14692457)

[Modify a vector 12](#_Toc14692458)

[Delete a Vector 13](#_Toc14692459)

[Sequences 14](#_Toc14692460)

[Coercion 14](#_Toc14692461)

[Not availables (NA) 14](#_Toc14692462)

[Factors 15](#_Toc14692463)

[Create 15](#_Toc14692464)

[c() 15](#_Toc14692465)

[Arguments 15](#_Toc14692466)

[use.names 15](#_Toc14692467)

[Access 15](#_Toc14692468)

[modify 15](#_Toc14692469)

[levels() 16](#_Toc14692470)

[reorder() 16](#_Toc14692471)

[Lists 17](#_Toc14692472)

[Matrices 18](#_Toc14692473)

[Create 18](#_Toc14692474)

[Access 18](#_Toc14692475)

[Notice that,the above 2 returns a vector, not a matrix. 18](#_Toc14692476)

[We can convert matrices into data frames using the function as.data.frame: 18](#_Toc14692477)

[Sorting 20](#_Toc14692478)

[sort 20](#_Toc14692479)

[order 20](#_Toc14692480)

[max and which.max 20](#_Toc14692481)

[rank 20](#_Toc14692482)

[Rescaling a vector 21](#_Toc14692483)

[which 22](#_Toc14692484)

[match 22](#_Toc14692485)

[%in% 22](#_Toc14692486)

[Basic Plot 22](#_Toc14692487)

[Plot 22](#_Toc14692488)

[hist 22](#_Toc14692489)

[boxplot 22](#_Toc14692490)

[image 23](#_Toc14692491)

[Conditional expressions 23](#_Toc14692492)

[If 23](#_Toc14692493)

[ifelse 23](#_Toc14692494)

[any and all 24](#_Toc14692495)

[Functions 24](#_Toc14692496)

[Defining functions 24](#_Toc14692497)

[In general, functions are objects, so we assign them to variable names with <-. 24](#_Toc14692498)

[ForLoop 25](#_Toc14692499)

[Vectorization and functionals 26](#_Toc14692500)

[*vectorized* function is a function that will apply the same operation on each of the vectors. 26](#_Toc14692501)

[*Functionals* are functions that help us apply the same function to each entry in a vector, matrix, data frame or list. 26](#_Toc14692502)

[sapply 26](#_Toc14692503)

[The function sapply permits us to perform element-wise operations on any function. 26](#_Toc14692504)

[The tidyverse 27](#_Toc14692505)

[Tidy data 27](#_Toc14692506)

[We say that a data table is in *tidy* format if each row represents one observation and columns represent the different variables available for each of these observations. 27](#_Toc14692507)

[The pipe: %>% 27](#_Toc14692508)

[Summarizing data 28](#_Toc14692509)

[Summarize 28](#_Toc14692510)

[Pull 28](#_Toc14692511)

[group\_by 28](#_Toc14692512)

[Misc Points 29](#_Toc14692513)

[a <- 1 29](#_Toc14692514)

[a OR print(a) 29](#_Toc14692515)

[ls() 29](#_Toc14692516)

[ls 29](#_Toc14692517)

[help("log") OR ?log 29](#_Toc14692518)

[& vs && in vector 29](#_Toc14692519)

[& operates on each element (i.e, vectorized) and return as many results as elements 29](#_Toc14692520)

[&& is not vectorized and if provided vectors, returns result for only first element 29](#_Toc14692521)

[args(log) 29](#_Toc14692522)

[log(8, base = 2) 29](#_Toc14692523)

[log(x = 8, base = 2) 29](#_Toc14692524)

[log(base = 2, x = 8) 29](#_Toc14692525)

[To specify arguments, we must use =, and cannot use <- 29](#_Toc14692526)

[There are some exceptions to the rule that functions need the parentheses to be evaluated. Among these, the most commonly used are the arithmetic and relational operators. For example: 2 ^ 3 30](#_Toc14692527)

[data() 30](#_Toc14692528)

[Saving/loading workspace : 30](#_Toc14692529)

[Comments 30](#_Toc14692530)

[seq(1, n) 30](#_Toc14692531)

[sum(x) 30](#_Toc14692532)

[class(a) 30](#_Toc14692533)

[Bydefault R stores Strings as Factors. And can be deactivated using stringsAsFactors = FALSE 30](#_Toc14692534)

[apply ? 31](#_Toc14692535)

[lapply ? 31](#_Toc14692536)

[tapply ? 31](#_Toc14692537)

[mapply ? 31](#_Toc14692538)

[vapply ? 31](#_Toc14692539)

[replicate ? 31](#_Toc14692540)

[Standard Deviation 31](#_Toc14692541)

[Bessel's Correction 31](#_Toc14692542)

# Namespaces

It is likely that two packages use the same name for two different functions. And often these functions do completely different things.

In fact, you have already encountered this becuase both **dplyr** and the R-base **stats** package define a filter function. There are five other examples in **dplyr**. We know this becasue when we first load **dplyr** we see the following message:

The following objects are masked from ‘package:stats’:

filter, lag

The following objects are masked from ‘package:base’:

intersect, setdiff, setequal, union

These function live in different *namespaces*.

## R will follow a certain order when searching for a function in these *namespaces*.

### You can see the order by typing: search()

### You can force the use of a specific name space by using double colons (::) like this:

stats::filter

If we want to be absolutely sure we use the dplyr filter we can use

### dplyr::filter

if we want to use a function in a package without loading the entire package, we can use the

double colon as well.

# Packages

## The purrr package

# Debugging

## traceback()

### Shows the calls

## debug(myfunc)

### Enters to debugging mode for function name provided. Each time function will be called, it enters to debugging mode.

## Commands while debugging:

### n : next line

### c :continue for execution

### Q :Quits/ comes out of the Debugging

### <Variablename> : Type variable name to get current value

## undebug(myfunc)

### Stops debugging for the function name provided

## debugonce(myfunc)

### Enters to debugging mode only for first time the myfunc will be called

## browser()

### It acts like debug point, and enters to debug mode when executed. Put this function in code where you want to stop for debugging.

# Data frame

The most common way of storing a dataset in R is in a data frame. Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns. Data frames are particularly useful for datasets because we can combine different data types into one object.

str(murders) //The function str is useful for finding out more about the structure of an object

-- Create

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Print the data frame.

print(emp.data)

-- Can provide the names explicitly

names(emp.data) <- c("Emp\_id", "Emp\_name", "Salary", "Start\_Date")

OR

emp.data <- data.frame(Emp\_id=emp\_id, Emp\_name=emp\_name, Salary=salary, Start\_Date=start\_date) //In case objects are present outside or created as vectors

-- Extract Data from Data Frame

result <- data.frame(emp.data$emp\_name,emp.data$salary)

# Extract 3rd and 5th row with 2nd and 4th column.

result <- emp.data[c(3,5),c(2,4)]

-- Expand Data Frame

A data frame can be expanded by adding columns and rows.

-- Add Column

Just add the column vector using a new column name.

# Add the "dept" coulmn.

emp.data$dept <- c("IT","Operations","IT","HR","Finance")

-- Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the rbind() function.

# Create the first data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

dept = c("IT","Operations","IT","HR","Finance"),

stringsAsFactors = FALSE

)

# Create the second data frame

emp.newdata <- data.frame(

emp\_id = c (6:8),

emp\_name = c("Rasmi","Pranab","Tusar"),

salary = c(578.0,722.5,632.8),

start\_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),

dept = c("IT","Operations","Fianance"),

stringsAsFactors = FALSE

)

# Bind the two data frames.

emp.finaldata <- rbind(emp.data,emp.newdata)

We can show the first six lines using the function head

head(murders)

The accessor: $

For our analysis, we will need to access the different variables represented by columns included in this data frame. To do this, we use the accessor operator $ as :

murders$population //can say as murders's population

We can quickly access the variable names using:

names(murders)

## Manipulating data frames

The **dplyr** package from the **tidyverse** introduces functions that perform some of the most common operations when working with data frames and uses names for these functions that are relatively easy to remember. For instance, to change the data table by adding a new column, we use mutate. To filter the data table to a subset of rows, we use filter. Finally, to subset the data by selecting specific columns, we use select.

### mutate : Adding a column with mutate

**library**(dslabs)

**data**("murders")

murders <- **mutate**(murders, rate = total **/** population **\*** 100000)

### filter : Subsetting with filter

**filter**(murders, rate **<=** 0.71)

### select : Selecting columns with select

new\_table <- **select**(murders, state, region, rate)

**filter**(new\_table, rate **<=** 0.71)

### e.g:

### murders <- **mutate**(murders, population\_in\_millions = population **/** 10**^**6)

### **select**(murders, state, population) **%>% head**()

### no\_florida <- **filter**(murders, state **!=** "Florida")

### **filter**(murders, state **%in% c**("New York", "Texas"))

### **filter**(murders, population **<** 5000000 **&** region **==** "Northeast")

### heights %>% filter(grepl("^male", sex, ignore.case=TRUE)) //case insensitive

## Sorting Data Frame

murders **%>% arrange**(population) **%>% head**()

murders **%>% arrange**(**desc**(rate)) **%>% head**()

## Nested Sorting Data Frame

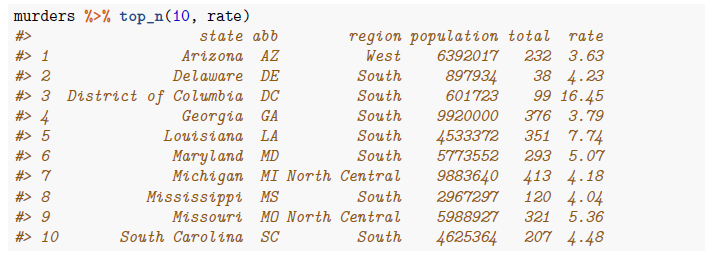
If we are ordering by a column with ties, we can use a second column to break the tie. Similarly, a third column can be used to break ties between first and second and so on. Here we order by region, then within region we order by murder rate:

murders **%>% arrange**(region, rate) **%>% head**()

## The top *n*

we have used the function head to avoid having the page fill up with the entire dataset.

If we want to see a larger proportion, we can use the top\_n function. This function takes a data frame as it’s first argument, the number of rows to show in the second, and the variable to filter by in the third. Here is an example of how to see the top 10 rows:



Note that rows are not sorted by rate, only filtered. If want to sort, we need to use arrange. Note that if the third argument is left blank, top\_n, filters by the last column.

murders **%>%** **arrange**(desc(total)) **%>% top**\_n(10, total)

# tibble

The tbl, pronounced tibble, is a special kind of data frame.

The functions group\_by and summarize always return this type of data frame.

the \_\_dplyr\_\_manipulation verbs (select, filter, mutate, and arrange) preserve the class of the input: if they receive a regular data frame they return a regular data frame, while if they receive a tibble they return a tibble. But tibbles are the preferred format in the tidyverse and as a result tidyverse functions that produce a data frame from scratch return a tibble.

## Tibble vs data frame

Tibbles are very similar to data frames. In fact, you can think of them as a modern version of data frames.

Nonetheless there are three important differences which we describe in the next.

### Tibbles display better

as\_tibble(murders)

### Subsets of tibbles are tibbles

If you subset the columns of a data frame, you may get back an object that is not a data frame, such as a vector or scalar. For example:

**class**(murders[,4])

*#> [1] "numeric"*

is not a data frame. With tibbles this does not happen:

**class**(**as\_tibble**(murders)[,4])

*#> [1] "tbl\_df" "tbl" "data.frame"*

This is useful in the tidyverse since functions require data frames as input.

### Tibbles can have complex entries

While data frame columns need to be vectors of numbers, strings or logical values, tibbles can have more complex objects, such as lists or functions. Also, we can create tibbles with functions:

**tibble**(id = **c**(1, 2, 3), func = **c**(mean, median, sd))

### Tibbles can be grouped

The function group\_by returns a special kind of tibble: a grouped tibble. This class stores information that lets you know which rows are in which groups. The tidyverse functions, in particular the summarize function, are aware of the group information.

## do

### The do function understands grouped tibbles and always returns a data frame.

The tidyverse functions know how to interpret grouped tibbles. Furthermore, to facilitate stringing commands through the pipe %>%, tidyverse functions consistently return data frames, since this assures that the output of a function is accepted as the input of another. But most R functions do not recognize grouped tibbles nor do they return data frames. The quantile function is an example. The do functions serves as a bridge between R functions such as quantile and the tidyverse.

# Vectors: numerics, characters, and logical

Vector is a basic data structure in R. It contains element of the same type. The data types can be logical, integer, double, character, complex or raw. Vectors are generally created using the c() function. Since, a vector must have elements of the same type, this function will try and coerce elements to the same type, if they are different. Coercion is from lower to higher types from logical to integer to double to character.

The object murders$population is not one number but several. We call these types of objects vectors. A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries. The function length tells you how many entries are in the vector

pop <- murders$population

length(pop)

This particular vector is numeric since population sizes are numbers:

class(pop)

# [1] "numeric"

In a numeric vector, every entry must be a number.

class(1) : returns numeric. You can turn them into class integer with the as.integer() function or by adding an L like this: 1L. Note the class by typing: class(1L)

## : operator

If we want to create a vector of consecutive numbers, the : operator is very helpful.

x <- 1:7; x

[1] 1 2 3 4 5 6 7

y <- 2:-2; y

[1] 2 1 0 -1 -2

## Creating a vector using seq() function

More complex sequences can be created using the seq() function, like defining number of points in an interval, or the step size.

seq(1, 3, by=0.2) # specify step size

[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0

seq(1, 5, length.out=4) # specify length of the vector

[1] 1.000000 2.333333 3.666667 5.000000

## Sometimes it is useful to name the entries of a vector.

For example, when defining a vector of country codes, we can use the names to connect the two:

codes <- c(italy = 380, canada = 124, egypt = 818)

codes

# italy canada egypt

# 380 124 818

## We can also assign names using the names functions:

codes <- c(380, 124, 818)

country <- c("italy","canada","egypt")

names(codes) <- country

codes

# italy canada egypt

# 380 124 818

## Access Elements of a Vector

Elements of a vector can be accessed using vector indexing. The vector used for indexing can be logical, integer or character vector.

Vector index in R starts from 1, unlike most programming languages where index start from 0.

### Using integer vector as index

We can use a vector of integers as index to access specific elements.

We can also use negative integers to return all elements except that those specified.

But we cannot mix positive and negative integers while indexing and real numbers, if used, are truncated to integers.

x

[1] 0 2 4 6 8 10

x[3] # access 3rd element

[1] 4

x[c(2, 4)] # access 2nd and 4th element

[1] 2 6

x[-1] # access all but 1st element

[1] 2 4 6 8 10

x[c(2, -4)] # cannot mix positive and negative integers

Error in x[c(2, -4)] : only 0's may be mixed with negative subscripts

x[c(2.4, 3.54)] # real numbers are truncated to integers

[1] 2 4

### Using logical vector as index

When we use a logical vector for indexing, the position where the logical vector is TRUE is returned.

x[c(TRUE, FALSE, FALSE, TRUE)]

[1] -3 3

x[x < 0] # filtering vectors based on conditions

[1] -3 -1

x[x > 0]

[1] 3

In the above example, the expression x>0 will yield a logical vector (FALSE, FALSE, FALSE, TRUE) which is then used for indexing.

### Using character vector as index

This type of indexing is useful when dealing with named vectors. We can name each elements of a vector.

x <- c("first"=3, "second"=0, "third"=9)

names(x)

[1] "first" "second" "third"

x["second"]

second

0

x[c("first", "third")]

first third

3 9

## Modify a vector

We can modify a vector using the assignment operator.

We can use the techniques discussed above to access specific elements and modify them.

If we want to truncate the elements, we can use reassignments.

x

[1] -3 -2 -1 0 1 2

x[2] <- 0; x # modify 2nd element

[1] -3 0 -1 0 1 2

x[x<0] <- 5; x # modify elements less than 0

[1] 5 0 5 0 1 2

x <- x[1:4]; x # truncate x to first 4 elements

[1] 5 0 5 0

## Delete a Vector

We can delete a vector by simply assigning a NULL to it.

x

[1] -3 -2 -1 0 1 2

x <- NULL

x

NULL

x[4]

NULL

(https://www.datamentor.io/r-programming/vector/)

# Sequences

Another useful function for creating vectors generates sequences:

seq(1, 10)

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

When we use these functions, R produces integers, not numerics, because they are typically used to index something:

class(1:10)

# [1] "integer"

However, if we create a sequence including non-integers, the class changes:

class(seq(1, 10, 0.5))

# [1] "numeric"

# Coercion

In general, coercion is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion.

We said that vectors must be all of the same type. So if we try to combine, say, numbers and characters, you might expect an error:

x <- c(1, "canada", 3)

But we don’t get one, not even a warning! What happened? Look at x and its class:

x

# [1] "1" "canada" "3"

class(x)

# [1] "character"

R coerced the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings "1" and “3”. The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to change from one type to another. For example, you can turn numbers into characters with:

x <- 1:5

y <- as.character(x)

y

# [1] "1" "2" "3" "4" "5"

You can turn it back with as.numeric:

as.numeric(y)

# [1] 1 2 3 4 5

# Not availables (NA)

When a function tries to coerce one type to another and encounters an impossible case, it usually gives us a warning and turns the entry into a special value called an NA for “not available”. For example:

x <- c("1", "b", "3")

as.numeric(x)

# Warning: NAs introduced by coercion

# [1] 1 NA 3

# Factors

Factors are useful for storing categorical data. Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data). For example: a data field such as marital status may contain only values from single, married, separated, divorced, or widowed.

In such case, we know the possible values beforehand and these predefined, distinct values are called levels.

Warning: Factors can be a source of confusion since sometimes they behave like characters and sometimes they do not. As a result, confusing factors and characters are a common source of bugs.

Bydefault R stores Strings as Factors. And can be deactivated using stringsAsFactors = FALSE

## Create

We can create a factor using the function factor(). Levels of a factor are inferred from the data if not provided.

x <- factor(c("single", "married", "married", "single")) //In this case levels are inferred

x <- factor(c("single", "married", "married", "single"), levels = c("single", "married", "divorced"))

//In this case levels are provided at creation time.

### c()

Combine Values Into A Vector Or List. This is a generic function which combines its arguments.

The default method combines its arguments to form a vector. All arguments are coerced to a common type which is the type of the returned value, and all attributes except names are removed.

## S3 Generic function

c(…)

# S3 method for default

c(…, recursive = FALSE, use.names = TRUE)

### Arguments

… : objects to be concatenated.

recursive : logical. If recursive = TRUE, the function recursively descends through lists (and pairlists) combining all their elements into a vector.

use.names : logical indicating if names should be preserved.

## Access

x # Shows all

x[3] # Access 3rd element

x[c(2, 4)] # access 2nd and 4th element

x[-1] # access all but 1st element

x[c(TRUE, FALSE, FALSE, TRUE)] # using logical vector

## modify

Components of a factor can be modified using simple assignments. However, we cannot choose values outside of its predefined levels.

x

[1] single married married single

Levels: single married divorced

x[2] <- "divorced" # modify second element

x[3] <- "widowed" # cannot assign values outside levels

Warning message:

In `[<-.factor`(`\*tmp\*`, 3, value = "widowed") :

invalid factor level, NA generated

A workaround to this is to add the value to the level first.

levels(x) <- c(levels(x), "widowed") # add new level

x[3] <- "widowed"

## levels()

We can see that there are only 4 regions by using the levels function: levels(murders$region)

The default is for the levels to follow alphabetical order

## reorder()

The function reorder lets us change the order of the levels of a factor variable based on a summary computed on a numeric vector.

region <- reorder(region, value, FUN = sum)

# Lists

Data frames are a special case of lists. They are useful because you can store any combination of different types.

As with data frames, you can extract the components of a list with the accessor $. In fact, data frames are a type of list.

# Matrices

Matrices are similar to data frames in that they are two-dimensional: they have rows and columns.

However, like numeric, character and logical vectors, entries in matrices have to be all the same type. For this reason data frames are much more useful for storing data, since we can have characters, factors and numbers in them.

Yet matrices have a major advantage over data frames: we can perform a matrix algebra operations, a powerful type of mathematical technique.

## Create

We can define a matrix using the matrix function. We need to specify the number of rows and columns.

mat <- matrix(1:12, 4, 3)

mat

# [,1] [,2] [,3]

# [1,] 1 5 9

# [2,] 2 6 10

# [3,] 3 7 11

# [4,] 4 8 12

## Access

You can access specific entries in a matrix using square brackets ([). If you want the second row, third column, you use:

mat[2, 3]

# [1] 10

If you want the entire second row, you leave the column spot empty:

mat[2, ]

# [1] 2 6 10

Similarly, if you want the entire third column, you leave the row spot empty:

mat[, 3]

# [1] 9 10 11 12

### Notice that,the above 2 returns a vector, not a matrix.

You can access more than one column or more than one row if you like. This will give you a new matrix.

mat[, 2:3]

# [,1] [,2]

# [1,] 5 9

# [2,] 6 10

# [3,] 7 11

# [4,] 8 12

You can subset both rows and columns:

mat[1:2, 2:3]

# [,1] [,2]

# [1,] 5 9

# [2,] 6 10

### We can convert matrices into data frames using the function as.data.frame:

as.data.frame(mat)

# V1 V2 V3

# 1 1 5 9

# 2 2 6 10

# 3 3 7 11

# 4 4 8 12

You can also use single square brackets ([) to access rows and columns of a data frame:

data("murders")

murders[25, 1]

# [1] "Mississippi"

murders[2:3, ]

# state abb region population total

# 2 Alaska AK West 710231 19

# 3 Arizona AZ West 6392017 232

# Sorting

## sort

**sort**(murders$total)

## order

x <- **c**(31, 4, 15, 92, 65)

**sort**(x)

*#> [1] 4 15 31 65 92*

Rather than sort the input vector, the function order returns the index that sorts input vector:

index <- **order**(x)

x[index]

*#> [1] 4 15 31 65 92*

## max and which.max

If we are only interested in the entry with the largest value, we can use max for the value:

**max**(murders$total)

*#> [1] 1257*

and which.max for the index of the largest value:

i\_max <- **which.max**(murders$total)

murders$state[i\_max]

*#> [1] "California"*

For the minimum, we can use min and which.min in the same way.

## rank

Although not as frequently used as order and sort, the function rank is also related to order and can be useful. For any given vector it returns a vector with the rank of the first entry, second entry, etc., of the input vector. Here is a simple example:

x <- **c**(31, 4, 15, 92, 65)

**rank**(x)

*#> [1] 3 1 2 5 4*

To summarize, let’s look at the results of the three functions we have introduced:

|  |  |  |  |
| --- | --- | --- | --- |
| **original** | **sort** | **order** | **rank** |
| 31 | 4 | 2 | 3 |
| 4 | 15 | 3 | 1 |
| 15 | 31 | 1 | 2 |
| 92 | 65 | 5 | 5 |
| 65 | 92 | 4 | 4 |

# Rescaling a vector

In R, arithmetic operations on vectors occur element-wise.

For a quick example, suppose we have height in inches:

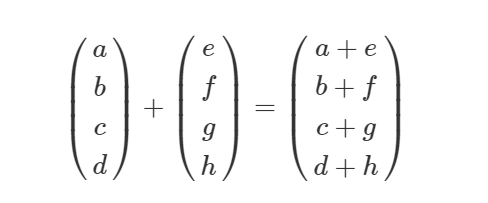
inches <- **c**(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)

and want to convert to centimeters. Notice what happens when we multiply inches by 2.54:

inches \* 2.54

*#> [1] 175 157 168 178 178 185 170 185 170 178*

If we have two vectors of the same length, and we sum them in R, they will be added entry by entry as follows:



The same holds for other mathematical operations, such as -, \* and /.

This implies that to compute the murder rates we can simply type:

murder\_rate <- murders$total / murders$population \* 100000

# which

The function which tells us which entries of a logical vector are TRUE.

ind <- **which**(murders**$**state **==** "California") //which index fulfills the condition

murder\_rate[ind]

*#> [1] 3.37*

# match

This function tells us which indexes of a second vector match each of the entries of a first vector

ind <- **match**(**c**("New York", "Florida", "Texas"), murders**$**state)

ind

*#> [1] 33 10 44*

Now we can look at the murder rates:

murder\_rate[ind]

*#> [1] 2.67 3.40 3.20*

# %in%

If rather than an index we want a logical that tells us whether or not each element of a first vector is in a second, we can use the function %in%.

**c**("Boston", "Dakota", "Washington") **%in%** murders**$**state // is vector in source vector

*#> [1] FALSE FALSE TRUE*

**Advanced**: There is a connection between match and %in% through which. To see this, notice that the

following two lines produce the same index (although in different order):

**match**(**c**("New York", "Florida", "Texas"), murders**$**state)

*#> [1] 33 10 44*

**which**(murders**$**state**%in%c**("New York", "Florida", "Texas"))

*#> [1] 10 33 44*

# Basic Plot

## Plot

x <- murders**$**population **/** 10**^**6

y <- murders**$**total

**plot**(x, y)

For a quick plot that avoids accessing variables twice, we can use the with function:

**with**(murders, **plot**(population, total))

## hist

x <- **with**(murders, total **/** population **\*** 100000)

**hist**(x)

We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme

case with a murder rate of more than 15:

murders**$**state[**which.max**(x)]

*#> [1] "District of Columbia"*

## boxplot

murders**$**rate <- **with**(murders, total **/** population **\*** 100000)

**boxplot**(rate**~**region, data = murders)

## image

The image function displays the values in a matrix using color. Here is a quick example:

x <- **matrix**(1**:**120, 12, 10)

**image**(x)

# Conditional expressions

## If

a <- 0

**if**(a**!=**0){

**print**(1**/**a)

} **else**{

**print**("No reciprocal for 0.")

}

*#> [1] "No reciprocal for 0."*

ind <- **which.min**(murder\_rate)

**if**(murder\_rate[ind] **<** 0.5){

**print**(murders**$**state[ind])

} **else**{

**print**("No state has murder rate that low")

}

*#> [1] "Vermont"*

If we try it again with a rate of 0.25, we get a different answer:

**if**(murder\_rate[ind] **<** 0.25){

**print**(murders**$**state[ind])

} **else**{

**print**("No state has a murder rate that low.")

}

*#> [1] "No state has a murder rate that low."*

## ifelse

a <- 0

**ifelse**(a **>** 0, 1**/**a, NA)

*#> [1] NA*

The function is particularly useful because it works on vectors. It examines each entry of the logical vector

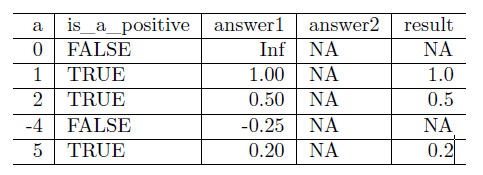
and returns elements from the vector provided in the second argument, if the entry is TRUE, or elements from

the vector provided in the third argument, if the entry is FALSE.

a <- **c**(0,1,2,**-**4,5)

result <- **ifelse**(a **>** 0, 1**/**a, NA)

This table helps us see what happened:



Here is an example of how this function can be readily used to replace all the missing values in a vector with

zeros:

**data**(na\_example)

no\_nas <- **ifelse**(**is.na**(na\_example), 0, na\_example)

**sum**(**is.na**(no\_nas))

*#> [1] 0*

## any and all

Two other useful functions are any and all. The any function takes a vector of logicals and returns TRUE if

any of the entries is TRUE. The all function takes a vector of logicals and returns TRUE if all of the entries

are TRUE. Here is an example:

z <- **c**(TRUE, TRUE, FALSE)

**any**(z)

*#> [1] TRUE*

**all**(z)

*#> [1] FALSE*

# Functions

## Defining functions

avg <- **function**(x){

s <- **sum**(x)

n <- **length**(x)

s**/**n

}

Now avg is a function that computes the mean:

x <- 1**:**100

**identical**(**mean**(x), **avg**(x))

*#> [1] TRUE*

### In general, functions are objects, so we assign them to variable names with <-.

The function function tells R you are about to define a function. The general form of a function definition looks like this:

my\_function <- **function**(VARIABLE\_NAME){

perform operations on VARIABLE\_NAME and calculate VALUE

VALUE

}

The functions you define can have multiple arguments as well as default values. For example, we can define

a function that computes either the arithmetic or geometric average depending on a user defined variable

like this:

avg <- **function**(x, arithmetic = TRUE){

n <- **length**(x)

**ifelse**(arithmetic, **sum**(x)**/**n, **prod**(x)**^**(1**/**n))

}

# ForLoop

**for**(i **in** 1**:**5){

**print**(i)

}

we can create one that computes the *Sn*:

compute\_s\_n <- **function**(n){

x <- 1**:**n

**sum**(x)

}

How can we compute Sn for various values of n, say n = 1, . . . , 25?

Here is the for-loop we would write for our *Sn* example:

m <- 25

s\_n <- **vector**(length = m) *# create an empty vector*

**for**(n **in** 1**:**m){

s\_n[n] <- **compute\_s\_n**(n)

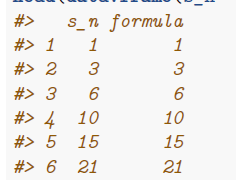
}

Now we can create a plot to search for a pattern:

n <- 1**:**m

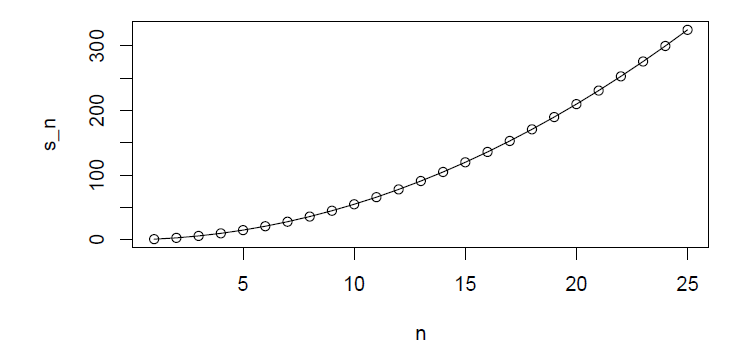
**plot**(n, s\_n)

**head**(**data.frame**(s\_n = s\_n, formula = n**\***(n**+**1)**/**2))



**plot**(n, s\_n)

**lines**(n, n**\***(n**+**1)**/**2)



# Vectorization and functionals

## *vectorized* function is a function that will apply the same operation on each of the vectors.

x <- 1**:**10

**sqrt**(x)

*#> [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16*

y <- 1**:**10

x**\***y

*#> [1] 1 4 9 16 25 36 49 64 81 100*

However, not all functions work this way. For instance, the function we just wrote, compute\_s\_n, does not work element-wise since it is expecting a scalar.

## *Functionals* are functions that help us apply the same function to each entry in a vector, matrix, data frame or list.

## sapply

### The function sapply permits us to perform element-wise operations on any function.

Each element of x is passed on to the function sqrt and the result is returned. These results are concatenated.

In this case, the result is a vector of the same length as the original x. This implies that the for-loop above can be written as follows:

n <- 1**:**25

s\_n <- **sapply**(n, compute\_s\_n)

**plot**(n, s\_n)

# The tidyverse

**install.packages("tidyverse")**

**library**(tidyverse)

## Tidy data

### We say that a data table is in *tidy* format if each row represents one observation and columns represent the different variables available for each of these observations.

# The pipe: %>%

With **dplyr** we can perform a series of operations, for example select and then filter, by sending the results of one function to another using what is called the *pipe operator*: %>%.

e.g. :

murders **%>% select**(state, region, rate) **%>% filter**(rate **<=** 0.71)

#### In general, the pipe sends the result of the left side of the pipe to be the first argument of the function on the right side of the pipe.

Here is a very simple example:

#### 16 %>% sqrt()

*#> [1] 4*

We can continue to pipe values along:

#### 16 %>% sqrt() %>% log2()

*#> [1] 2*

The above statement is equivalent to log2(sqrt(16)).

#### Remember that the pipe sends values to the first argument, so we can define other arguments as if the first

argument is already defined:

#### 16 **%>% sqrt**() **%>% log**(base = 2)

*#> [1] 2*

Therefore, when using the pipe with data frames and **dplyr**, we no longer need to specify the required first argument since the **dplyr** functions we have described all take the data as the first argument. In the code *we wrote:*

#### murders **%>% select**(state, region, rate) **%>% filter**(rate **<=** 0.71)

#### Note that the pipe works well with functions where the first argument is the input data. Functions in **tidyverse** packages like **dplyr** have this format and can be used easily with the pipe.

# Summarizing data

## Summarize

since, as most **dplyr** functions, summarize always returns a data frame.

## Pull

since, as most **dplyr** functions, summarize always returns a data frame.

This might be problematic if we want to use this result with functions that require a numeric value. Here we show a useful trick for accessing values stored in data when using pipes: when a data object is piped that object and its columns can be accessed using the pull function. To understand what we mean take a look at this line of code:

us\_murder\_rate **%>% pull**(rate)

*#> [1] 3.03*

This returns the value in the rate column of us\_murder\_rate making it equivalent to us\_murder\_rate$rate.

To get a number from the original data table with one line of code we can type:

us\_murder\_rate <- murders **%>% summarize**(rate = **sum**(total) **/ sum**(population) **\*** 100000) **%>% pull**(rate)

us\_murder\_rate

*#> [1] 3.03*

which is now a numeric:

**class**(us\_murder\_rate)

*#> [1] "numeric"*

## group\_by

This converts the data frame to a special data frame called a *grouped data frame* and **dplyr** functions, in particular summarize, will behave differently when acting on this object. Conceptually, you can think of this table as many tables, with the same columns but not necessarily the same number of rows, stacked together in one object.

class(heights)

[1] "data.frame"

grouped\_heights <- heights %>% group\_by(sex)

class(grouped\_heights)

[1] "grouped\_df" "tbl\_df" "tbl" "data.frame"

heights **%>% group\_by**(sex) **%>% m msummarize**(average = **mean**(height), standard\_deviation = **sd**(height))

*#> # A tibble: 2 x 3*

*#> sex average standard\_deviation*

*#> <fct> <dbl> <dbl>*

*#> 1 Female 64.9 3.76*

*#> 2 Male 69.3 3.61*

The summarize function applies the summarization to each group separately.

# Misc Points

## a <- 1

We use <- to assign values to the variables. We can also assign values using = instead of <-, but we recommend against using = to avoid confusion.

## a OR print(a)

we simply ask R to evaluate a and it shows the stored value:

## ls()

You can see all the variables saved in your workspace by typing ls()

## ls

If you type ls, the function is not evaluated and instead R shows you the code that defines the function.

## help("log") OR ?log

You can find out what the function expects and what it does by reviewing the very useful manuals included in R.

## & vs && in vector

### & operates on each element (i.e, vectorized) and return as many results as elements

### && is not vectorized and if provided vectors, returns result for only first element

> m <- 1:3

> m

[1] 1 2 3

> n <- 6:4

> n

[1] 6 5 4

> (m<2) & (n>5)

[1] TRUE FALSE FALSE

> (m<2) && (n>5)

[1] TRUE

## args(log)

If you want a quick look at the arguments without opening the help system

## log(8, base = 2)

You can change the default values by simply assigning another object:

## log(x = 8, base = 2)

This code works, but we can save ourselves some typing: if no argument name is used, R assumes you are entering arguments in the order shown in the help file or by args. So by not using the names, it assumes the arguments are x followed by base

## log(base = 2, x = 8)

If using the arguments’ names, then we can include them in whatever order we want:

## To specify arguments, we must use =, and cannot use <-

## There are some exceptions to the rule that functions need the parentheses to be evaluated. Among these, the most commonly used are the arithmetic and relational operators. For example: 2 ^ 3

## data()

There are several datasets that are included for users to practice and test out functions. You can see all the available datasets by typing data()

## Saving/loading workspace :

You can do this by using the function save or save.image

To load, use the function load

When saving a workspace, we recommend the suffix rda or RData. In RStudio, you can also do this by navigating to the Session tab and choosing Save Workspace as.

## Comments

If a line of R code starts with the symbol #, it is not evaluated.

## seq(1, n)

creates sequence of 1 to n

## sum(x)

adds sequence inside x

Below is a program for sum of 1 to 1000

n <- 1000

x <- seq(1, n)

sum(x)

OR

sum(seq(1,n))

## class(a)

The function class helps us determine what type of object we have

## Bydefault R stores Strings as Factors. And can be deactivated using stringsAsFactors = FALSE

# apply ?

# lapply ?

# tapply ?

# mapply ?

# vapply ?

# replicate ?

Logistics , split, cut, do.call and Reduce

# Standard Deviation

<https://simple.wikipedia.org/wiki/Standard_deviation>

## Bessel's Correction

To find sd in case only part of group is known. In this we use (n-1) instead of n as the denominator of formula.