# vectors have variables of \_one\_ type

c(1, 2, "three")

# shorter arguments are recycled

(1:3) \* 2

(1:4) \* c(1, 2)

# warning! (why?)

(1:4) \* (1:3)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Each element of a vector can be given a name.

x <- c("red", "green", "blue")

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Obviously the second version is much more suggestive of its meaning. The names of a vector

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_need not be unique, but in most applications you'll want unique names (if any).

capColor = c(huey = "red", duey = "blue", louie = "green")

capColor

capColor["louie"]

#-------------- To get the variable name with the specified value

names(capColor)[capColor == "blue"]

#---------------vector created with few data

x <- c(4, 7, 6, 5, 2, 8)

#---------------creating two conditions

I <- x < 6

J <- x > 6

#---------------Printing which all values satisfies this condition.

x[I | J]

x[I]

Below

#?????????????????????????????????????????????????????????????????

x[c(TRUE, FALSE)]

x[c(-1, -2)]

x[c(TRUE)]

#-------------------Replacing values in vectors can be done in the same way

x <- 1:10

--------------every other value of x is replaced with 1 whchi satisfy the condition mentioned

x[c(TRUE, FALSE)] <- 1

#------------------A list is generalization of a vector that contains vector of different types even it may include other lists too

L <- list(x = c(1:5), y = c("a", "b", "c"), z = capColor)

# below are the way how we can fetch data from list

# below command just mentioning the second colum name to see the values

L[[2]]

# another way is to mentiond the variable name in the list preseeded by $

L$y

# another way is to mentiond the row number and column number() this can be shown as combinbations of column numbers too

# below syntax is to pull column 2 and 3

L[c(2,3)]

# we can also pull this by mentioning the column names in combination too

L[c("x", "y")]

L[["z"]]

#--------------A data.frame is not much more than a list of vectors, possibly of different types, but with

# every vector (now columns) of the same length. Since data.frames are a type of list, indexing

# them with a single index returns a sub-data.frame; that is, a data.frame with less columns

#-----------------VVI main thing to remember list gives horoizantal values and data frame return vertical valiues like Matrix

d <- data.frame(x = 1:10, y = letters[1:10], z = LETTERS[1:10])

d[1] #----------pull a column no

d[, 2] #---------Pull [all row,column]

d[, "x", drop = TRUE]

d[c("x", "z")]

d[d$x > 3, "y", drop = FALSE] #--------Pull all values of x where >3 and corosponding value of Y

d[2, ] #--------Pull all column but 2nd row

#------------Special values Like most programming languages, R has a number of Special values that are exceptions to the

# normal values of a type. These are NA, NULL, ±Inf and NaN

NA + 1

sum(c(NA, 1, 2))

median(c(NA, 1, 2, 3), na.rm = T) #------ Median will b e calculated if na.rm=TRUE else it will be NA

length(c(NA, 2, 3, 4))

3 == NA

NA == NA

TRUE | NA

#---------------------The function is.na can be used to detect NA's.

length(c(1, 2, NULL, 4))

sum(c(1, 2, NULL, 4), na.rm = T)

x <- NULL

c(x, 2)

#----------------------The function is.null can be used to detect NULL variables. is.null is a primitive function

is.null(L)

is.null(integer(0))

is.null(logical(0))

as.null(list(a = 1, b = "c"))

#------------------------------Below example we are just assigning same matrix (m) to m1,m2,m3,m4

m <- matrix(round(100 \* rnorm(6)), 2,3); m1 <- m2 <- m3 <- m4 <- m

dimnames(m1) <- list(NULL, NULL)

dimnames(m2) <- list(NULL, character())

dimnames(m3) <- rev(dimnames(m2))

dimnames(m4) <- rep(list(character()),2)

m4 ## prints absolutely identically to m or not by using stopifnot() function

stopifnot(m == m1, m1 == d, m2 == m3, m3 == m4,

identical(capture.output(m) -> cm,

capture.output(m1)),

identical(cm, capture.output(m2)),

identical(cm, capture.output(m3)),

identical(cm, capture.output(m4)))

??stopifnot()

#--------------------------------Usage of stopifnot() function

stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # -----here all.equal function and stopifnot function is explained

m <- matrix(c(1,3,3,1), 2, 2)

stopifnot(m == t(m), diag(m) == rep(1, 2)) # all(.) |=> TRUE

op <- options(error = expression(NULL))

# "disable stop(.)" << Use with CARE! >>

stopifnot(all.equal(pi, 3.141593), 2 < 2, all(1:10 < 12), "a" < "b") #-----Program will stop at the point of mismatch

stopifnot(all.equal(pi, 3.1415927), 2 < 2, all(1:10 < 12), "a" < "b")

options(op) # revert to previous error handler

op

#---------------Inf Stands for infinity and only applies to vectors of class numeric. A vector of class integer can

# never be Inf. This is because the Inf in R is directly derived from the international standard

# for floating point arithmetic 1. Technically, Inf is a valid numeric that results from

# calculations like division of a number by zero

pi/0

2 \* Inf

Inf - 1e+10

Inf + Inf

3 < -Inf

Inf == Inf

#-----------------NaN Stands for not a number. This is generally the result of a calculation of which the result is

# unknown, but it is surely not a number. In particular operations like 0/0, Inf-Inf and

# Inf/Inf result in NaN. Technically, NaN is of class numeric, which may seem odd since it is

# used to indicate that something is not numeric

Inf-Inf

NaN + 1

exp(NaN)

exp(-Inf)

#--------------------------------READING A FILE INTO R ENVIRONMENT

# read.csv for comma separated values with period as decimal separator.

# read.csv2 for semicolon separated values with comma as decimal separator.

# read.delim tab-delimited files with period as decimal separator.

# read.delim2 tab-delimited files with comma as decimal separator.

# read.fwf data with a predetermined number of bytes per column.

#Argument description

#header Does the first line contain column names?

#col.names character vector with column names.

#na.string Which strings should be considered NA?

#colClasses character vector with the types of columns.

Will coerce the columns to the specified types.

#stringsAsFactors If TRUE, converts all character vectors into factor vectors.

#sep Field separator.

Used only internally by read.fwf

getwd()

setwd("F:/Practice R")

#-------- during th file import how to mention the column names if the column names (header) does not exist already

# person <- read.csv( file = "pp.txt" , header = FALSE , col.names = c("age","height") )

person

#------ if we don't give the column names then the first line by default will e considered as column names

str(person) #--- this is to check the variable type and some sample values

#---------------if a column contains Na or any other character then R will change the type of variable to factor

# We can change the same with below code with stringsAsFactor() and as.numeric()

dat <- read.csv(

file = "pp.txt"

, header = FALSE

, col.names = c("age","height")

, stringsAsFactors=FALSE)

dat$height <- as.numeric(dat$height)

str(dat)

#-----------------------------HOW TO READ SELECTIVE LINES FROM A TXT FILE

# Selecting lines containing data using grep function.

(txt <- readLines("pg.txt"))

# detect lines starting with a percentage sign..

I <- grepl("^%", txt)

# and throw them out

(dat <- txt[!I])

## [1] "Gratt,1861,1892" "Bob,1892" "1871,Emmet,1937"

# Table 1: Steps to take when converting lines in a raw text file to a data.frame with correctly typed columns.

# Step result

# 1 Read the data with readLines character

# 2 Select lines containing data character

# 3 Split lines into separate fields list of character vectors

# 4 Standardize rows list of equivalent vectors

# 5 Transform to data.frame data.frame

# 6 Normalize and coerce to correct type data.frame

# --------------Split lines into separate fields. This can be done with strsplit. This function accepts

# a character vector and a split argument which tells strsplit how to split a string into

# substrings. The result is a list of character vectors.

(fieldList <- strsplit(dat, split = ",")) #this will split the sentense to words. here "," was set as delemeter

## [[1]]

## [1] "Gratt" "1861" "1892"

##

## [[2]]

## [1] "Bob" "1892"

##

## [[3]]

## [1] "1871" "Emmet" "1937"

#-------------Step 4. Standardize rows. The goal of this step is to make sure that 1) every row has the same

# number of fields and 2) the fields are in the right order

#---------------------creating a macro for

assignFields <- function(x){

out <- character(3)

# get names

i <- grepl("[[:alpha:]]",x)

out[1] <- x[i]

# get birth date (if any)

i <- which(as.numeric(x) < 1890)

out[2] <- ifelse(length(i)>0, x[i], NA)

# get death date (if any)

i <- which(as.numeric(x) > 1890)

out[3] <- ifelse(length(i)>0, x[i], NA)

out

}

standardFields <- lapply(datasetname, assignFields) #---- lapply function is called on fieldlist data set which is strsplit data

standardFields

# ---------------------------------PARALLEL PROCESSING

# -----------------------------Below code will do parallel processing when there is fair amount of processing required

# Also see the code from Krishnendu.... this is not working

install.packages("parallel")

library(parallel)

cluster <- makeCluster(4)

standardFields <- parLapply(cl=cluster, fieldList, assignFields)

stopCluster(cl)

#------------------------Transform to data.frame. There are several ways to transform a list to a data.frame

# object. Here, first all elements are copied into a matrix which is then coerced into a

# data.frame.

(M <- matrix(

unlist(standardFields)

, nrow=length(standardFields)

, byrow=TRUE))

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

## [1,] "Gratt" "1861" "1892"

## [2,] "Bob" NA "1892"

## [3,] "Emmet" "1871" "1937"

colnames(M) <- c("name","birth","death")

(daltons <- as.data.frame(M, stringsAsFactors=FALSE))

M <- M[-1,] #-------------------------------------------as the first and last low did not ha ve good values capturedhence deleted them

#------------------------------------Step 6. Normalize and coerce to correct types.

daltons$birth <- as.numeric(daltons$birth)

daltons$death <- as.numeric(daltons$death)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

daltons = transform( daltons

, birth = as.numeric(birth)

, death = as.numeric(death)

)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_TYPE CONVERSION\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#

# as.numeric as.logical

# as.integer as.factor

# as.character as.ordered

as.numeric(c("7", "7\*", "7.0", "7,0"))

class(c("abc", "def"))

## [1] "character"

class(1:10)

## [1] "integer"

class(c(pi, exp(1)))

## [1] "numeric"

class(factor(c("abc","def")))

sapply(dat, class)

# In R, the value of categorical variables is stored in factor variables. A factor is an integer

# vector endowed with a table specifying what integer value corresponds to what level. The

# values in this translation table can be requested with the levels function.

f <- factor(c("a", "b", "a", "a", "c"))

levels(f)

## [1] "a" "b" "c"

# example:

gender <- c(2, 1, 1, 2, 0, 1, 1)

# recoding table, stored in a simple vector

recode <- c(male = 1, female = 2)

(gender <- factor(gender, levels = recode, labels = names(recode))) # this shows how to recode variable values from 2,1 to Male and Female

## [1] female male male female <NA> male male

## Levels: male female

#---------------The relevel function allows you to determine which level comes first

(gender <- relevel(gender, ref = "female"))

# Levels can also be reordered, depending on the mean value of another variable, for example

age <- c(27, 52, 65, 34, 89, 45, 68)

(gender <- reorder(gender, age))

# Levels can also be reordered, depending on the mean value of another variable,

age <- c(27, 52, 65, 34, 89, 45, 68)

(gender <- reorder(gender, age))

#Here, the means are added as a named vector attribute to gender. It can be removed by setting that attribute to NULL.

gender

attr(gender, "scores") <- NULL #---- this is removing only the average score from Gender variable

gender

#-------------------------The base R installation has three types of objects to store a time instance: Date, POSIXlt and

# POSIXct. The Date object can only be used to store dates, the other two store date and/or time

current\_time <- Sys.time() #----------this gives the current system time

class(current\_time)

## [1] "POSIXct" "POSIXt"

current\_time

## [1] "2013-10-28 11:12:50 CET"

# ---------------The lubridate package13 contains a number of functions facilitating the conversion of text to

# POSIXct dates.

library(lubridate)

dates <- c("15/02/2013", "15 Feb 13", "It happened on 15 02 '13")

q<-dmy(dates) #------------------this means Date month Year of dates variable

q

#NOTE:------------

#---------------------Here, the function dmy assumes that dates are denoted in the order day-month-year and tries to

# extract valid dates. Note that the code above will only work properly in locale settings where

# the name of the second month is abbreviated to Feb. This holds for English or Dutch locales, but

# fails for example in a French locale (Février)

dmy myd ydm

mdy dym ymd

dmy("01 01 68")

# Code description Example

# %a Abbreviated weekday name in the current locale. Mon

# %A Full weekday name in the current locale. Monday

# %b Abbreviated month name in the current locale. Sep

# %B Full month name in the current locale. September

# %m Month number (01-12) 09

# %d Day of the month as decimal number (01-31). 28

# %y Year without century (00-99) 13

# %Y Year including century. 2013

mybirth <- dmy("28 Sep 1976")

format(mybirth, format = "I was born on %B %d, %Y")

## [1] "I was born on September 28, 1976"

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_String normalization\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Extra white spaces at the beginning or end of a string can be removed using str\_trim.

library(stringr)

str\_trim(" hello world ")

## [1] "hello world"

str\_trim(" hello world ", side = "left")

## [1] "hello world "

str\_trim(" hello world ", side = "right")

## [1] " hello world"

str\_pad(112, width = 9, side = "left", pad = "X") # this includes the space in string

## [1] "xxxxxx112"

toupper("Hello world") # to make something in upper case

## [1] "HELLO WORLD"

tolower("Hello World") # to make something in lower case

## [1] "hello world"

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Approximate string matching\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_