Apuntes Intermediate R

Pilar Amat Rodrigo

9/1/2018

Contents

Relational Operators

Equality == & Inequality !=

```
32==31
```

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

```
"apple"!="pear"
```

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

```
TRUE==TRUE
```

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

> < / >= <=

- When comparing strings R tales an alphabetical order, meaning a smallest and z biggest

```
"Apple"< "Pear"
```

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

- When comparing logicals, TRUE coerces to 1 and FALSE coerces to 0, so:

```
TRUE > FALSE
```

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

- We can compare also Vectors and Matrices

```
my_blog_visits<- c(12,0,3,4,6)
my_IG_visits <-c(11,16,25,12,4)
mean(my_IG_visits)
```

```
## [1] 13.6
```

```
my_blog_visits >= mean(my_IG_visits)
```

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

Relational Operators

AND operator (&)

Only TRUE if both arguments are TRUE, FALSE otherwise, see some examples

```
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
x<- 10
x>4 & x<14
```

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

```r
x>4 & x>14
```

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

OR operator (|)

TRUE is at least ONE of the arguments is TRUE, FALSE only if all arguments are FALSE

```r
TRUE | TRUE
```

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

```r
TRUE | FALSE
```

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

```r
x<- 10
x>4 | x<14
```

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

```r
x<4 | x>14
```

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

NOT Operator (!)

Changes the logical value result.

```r
!TRUE
```

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

```r
!FALSE
```

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

- DOUBLE SIGNES &&, ||: Operates ONLY with the FIRST element of each vector, while the single operator does work in all the elements of the vector.

```r
c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)
```

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

```r
c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FALSE)
```

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

Conditional Statements

IF statement

When the condition is TRUE, the expression gets executed. If the expression is FALSE will not get executed and the printout will NOT occur.

```
if(condition){
  expression
}
```

```r
x<- 45
if(x>0) {
  print("x is a positive number")
}
```

```
## [1] "x is a positive number"
```

```r
x<- -49
if(x>0) {
  print("x is a positive number")
}
```

This second expression is false so there is NO printout!

ELSE statement

Gets executed when the condition DOESN'T get satisfied.

```
    if(condition){
      expression 1
    } else {
      expression 2
    }
```

```r
x<- -49
if(x>0) {
  print("x is a positive number")
}else {
  print("x is a negative number or zero")
}
```

```
## [1] "x is a negative number or zero"
```

ELSE IF

Comes in between the IF and ELSE statement, so we can add more conditions. ELSE IF will be repeated as many times as we need and we just put ELSE (without IF) for the not decrived conditions.

```
  if (condition1) {
    expression 1
  } else if (condition 2) {
    expression 2
  } else if (condition 3) {
    expression 3
  } else {
    expression 4
  }
```

In this example try x<-0 "x is zero" and also other numbers positive or negative to see the answers.

```r
x<- 0

if(x>0) {
  print("x is a positive number")
}else  if (x==0){
  print ("x is zero")
} else {
  print("x is a negative number or zero")
}
```

```
## [1] "x is zero"
```

IMPORTANT: The formula stops once the condition IS TRUE, so if the arguments aren't exclusive, there can be problems.

**Example joining concepts seen so far:

```
# Variables related to your last day of recordings
ig <- 15
fb <- 9

# Code the control-flow construct
if (ig>=15 & fb>=15) {
  sms <- 2 * (ig + fb)
} else if (ig<10 & fb<10) {
  sms <- 0.5 * (ig + fb)
} else {
  sms <- ig+fb
}

# Print the resulting sms to the console
sms
```

```
## [1] 24
```

Loops

WHILE Loop

Executes the code inside if as long the condition is TRUE. When the conditions fails, meaning is FALSE, the R abandons the While loop. ALWAYS MAKE SURE THAT THE LOOP ENDS AT SOME POINT.

```
  while(condition) {
    expression
  }
```

In this example, R executes the loop while the condition is TRUE and adds one number (z<- z+1) in each run.If we miss this part, the loop will never end!

```
z<- 1
while (z<=5) {
  print (paste("z is set to", z))
  z<- z+1
}
```

```
## [1] "z is set to 1"
## [1] "z is set to 2"
## [1] "z is set to 3"
## [1] "z is set to 4"
## [1] "z is set to 5"
```

BREAK STATEMENT: It breaks out the WHILE loop, when R finds it, it abandons the currently active loop.

**Example we say to break the loop when we get a number that divided by 5 gives a remainded that equals 0.

```
z<- 1
while (z<=100) {
  if(z %% 5 == 0) {
    break
  }
  print (paste("z is set to", z))
  z<- z+1
}
```

```
## [1] "z is set to 1"
## [1] "z is set to 2"
## [1] "z is set to 3"
## [1] "z is set to 4"
```

** Example 2

```
# Initialize the speed variable
speed <- 88

while (speed > 30) {
  print(paste("Your speed is", speed))

  # Break the while loop when speed exceeds 80
  if (speed>80 ) {
    break
  }

  if (speed > 48) {
    print("Slow down big time!")
    speed <- speed - 11
  } else {
    print("Slow down!")
    speed <- speed - 6
  }
}
```

```
## [1] "Your speed is 88"
```

FOR Loop

It uses as variables the ones that are in a vector, list, matrix…

```
  for(variable in a sequence) {
    expression
  }
```

FOOR Loop WHITH POSITION

Gives acces to the looping index.

```
  for (i in 1:length(vector)) {
```

```
  expression(vector[i])
  }
```

**Examples comparing two FOR Loop methods, with exact results:

```r
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
# Loop version 1
for (views in linkedin) {
  print(views)
}
```

```
## [1] 16
## [1] 9
## [1] 13
## [1] 5
## [1] 2
## [1] 17
## [1] 14
```

```r
# The nyc list is already specified
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"),
            capital = FALSE)
# Loop version 1
for (element in nyc){
  print (element)
}
```

```
## [1] 8405837
## [1] "Manhattan"     "Bronx"         "Brooklyn"      "Queens"
## [5] "Staten Island"
## [1] FALSE
```

```r
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
# Loop version 2
for (i in 1:length(linkedin)){
  print(linkedin[i])
}
```

```
## [1] 16
## [1] 9
## [1] 13
## [1] 5
## [1] 2
## [1] 17
## [1] 14
```

```r
# The nyc list is already specified
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"),
            capital = FALSE)
# Loop version 2
for(i in 1:length(nyc)){
  print(nyc[[i]])
}
```

```
## [1] 8405837
## [1] "Manhattan"     "Bronx"         "Brooklyn"      "Queens"
## [5] "Staten Island"
## [1] FALSE
```

** Example with nested FOR Loop

```r
# The tic-tac-toe matrix ttt has already been defined for you
ttt<-matrix(c(c("O", "NA","X"),c("NA","O","O"),c("X","NA","X")),byrow=TRUE, nrow=3)
ttt
```

```
##      [,1] [,2] [,3]
## [1,] "O"  "NA" "X"
## [2,] "NA" "O"  "O"
## [3,] "X"  "NA" "X"
```

```r
# define the double for loop
for (i in 1:nrow(ttt)) {
  for (j in 1:ncol(ttt)) {
    print(paste("On row", i, "and column", j, "the board contains", ttt[i,j]))
  }
}
```

```
## [1] "On row 1 and column 1 the board contains O"
## [1] "On row 1 and column 2 the board contains NA"
## [1] "On row 1 and column 3 the board contains X"
## [1] "On row 2 and column 1 the board contains NA"
## [1] "On row 2 and column 2 the board contains O"
## [1] "On row 2 and column 3 the board contains O"
## [1] "On row 3 and column 1 the board contains X"
## [1] "On row 3 and column 2 the board contains NA"
## [1] "On row 3 and column 3 the board contains X"
```

** Exemple with conditionals and FOR Loop

```r
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Code the for loop with conditionals
for (li in linkedin) {
  if (li>10 ) {
    print("You're popular!")
  } else {
    print("Be more visible!")
  }
  print(li)
}
```

```
## [1] "You're popular!"
## [1] 16
## [1] "Be more visible!"
## [1] 9
## [1] "You're popular!"
## [1] 13
## [1] "Be more visible!"
## [1] 5
## [1] "Be more visible!"
## [1] 2
## [1] "You're popular!"
## [1] 17
## [1] "You're popular!"
## [1] 14
```

FOR Loops works with important 2 statements: BREAK and NEXT.

BREAK Statement: abandons the active loop: the remaining code in the loop is skipped and the loop is not iterated over anymore.

```
# Pre-defined variables.
#Can you write code that counts the number of r's that come before the first u in rquote?
rquote <- "r's internals are irrefutably intriguing"
chars <- strsplit(rquote, split = "")[[1]]

# Initialize rcount
rcount <- 0

# Finish the for loop
for (char in chars) {
  if(char=="r"){
  rcount<- rcount+1
  }else if (char=="u"){
    break
  }
}
# Print out rcount
rcount
```

```
## [1] 5
```

NEXT Statement: Jumps the variable that makes the condition FALSE and proceds to the next variable. Skips the remainder of the code in the loop, but continues the iteration. **Example with BREAK and NEXT

```
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Adapt/extend the for loop
for (li in linkedin) {
  if (li > 10) {
    print("You're popular!")
  } else {
    print("Be more visible!")
  }
  # Add if statement with break
  if (li > 16) {
    print("This is ridiculous, I'm outta here!")
    break
  }
  # Add if statement with next
  if (li < 5) {
    print("This is too embarrassing!")
    next
  }
  print(li)
}
```

```
## [1] "You're popular!"
## [1] 16
## [1] "Be more visible!"
## [1] 9
## [1] "You're popular!"
## [1] 13
## [1] "Be more visible!"
## [1] 5
## [1] "Be more visible!"
## [1] "This is too embarrassing!"
## [1] "You're popular!"
## [1] "This is ridiculous, I'm outta here!"
```

Functions

Function Documentation

To consult the documentation use also this link:www.rdocumentation.org A quick hack to see the arguments of the function is the args() function.

```
help(name of the formula)
?name of the formula
args(name of the formula)
```

An optional argument like na.rm that specifies whether or not to remove missing values from the input vector

```
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, NA, 17, 14)
facebook <- c(17, NA, 5, 16, 8, 13, 14)

# Basic average of linkedin
mean(linkedin)
```

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

```
# Advanced average of linkedin
mean(linkedin,na.rm=TRUE)
```

```
## [1] 12.33333
```

¿Cómo sé que argumentos son opcionales? Ver funcion read.table

Writting Functions

```
  my_function<- function(argument1,argument2){
    body
  }
```

- We can add a default value for an argument by setting its value when it is not specified. Using =.

  my_function<- function(argument1,argument2=SOME VALUE){ body }

- Function scoping:It implies that variables that are defined inside a function are not accessible outside that function. Try running the following code and see if you understand the results:

```
pow_two <- function(x) {
  y <- x ^ 2
  return(y)
}
pow_two(4)
```

```
## [1] 16
```

y was defined inside the pow_two() function and therefore it is not accessible outside of that function. This is also true for the function's arguments of course - x in this case.

- R passes arguments by value: t means that an R function cannot change the variable that you input to that function.

```
increment <- function(x, inc = 1) {
  x <- x + inc
  x
}
count <- 5
a <- increment(count, 2)
b <- increment(count)
count <- increment(count, 2)
```

Given that R passes arguments by value and not by reference, the value of count is not changed after the first two calls of increment(). Only in the final expression, where count is re-assigned explicitly, does the value of count change.

**Ejercicio x x x x x x x x

x

R Packages

- There are 7 by defalult
- Install it: >install.packages("ggvis")
- Load packages: > library("ggvis")
- If we want to check that the pack is set run > search() and it has to appear in the result
- Another way to call a new package is by using the function: >require("thenameofthepack") if we combine it with the function > result <- require("thenameofthepack") and it turns FALSE means the package failed.

The Apply family

lapply

lapply takes a vector or list X, and applies the function FUN to each of its members. If FUN requires additional arguments, you pass them after you've specified X and FUN (. . .). The output of lapply() is a list, the same length as X, where each element is the result of applying FUN on the corresponding element of X.

```
  lapply(X, FUN, ...)
```

It makes easy some for loop and functions. Iy applies the same change to all the elements on the list/vector. It always returns a list so to see the result as a vector use the function > unlist() **Example 1

```
#We want to see what kind of elements "class()" contain the list "nyc"
nyc<- list(pop=8405837, boroughs=c("Manhattan","Bronx","Brooklyn","Queens","Staten Island"),capital=FAL

#We run a for loop that prints every element at a time.
for(info in nyc){
  print(class(info))
}
```

```
## [1] "numeric"
## [1] "character"
## [1] "logical"
```

```
#There is a shortcut using lapply. Which will return a list keeping the names of the elements.
lapply(nyc, class)
```

```
## $pop
## [1] "numeric"
##
## $boroughs
## [1] "character"
##
## $capital
## [1] "logical"
```

**Example 2

```
#We want to see how mamy characters has each element inside the vector cities "nchar()"
cities<-c("New York", "Paris", "London", "Tokyo", "Rio de Janeiro","Cape Town")
#We run a for loop
num_chars<- c()
for (i in 1:length(cities)) {
  num_chars[i]<-nchar(cities[i])
}
num_chars
```

```
## [1]  8  5  6  5 14  9
```

```
#We can bild a lapply simplier, but we don't want the result as a list so we will use unlist() so we ge
unlist(lapply(cities, nchar))
```

```
## [1]  8  5  6  5 14  9
```

**Example 3

```
#We want to multiply by 3 the prices of the oil list. First lets create our own funcion "multiply"
oil_prices<-list(2.37,2.49,2.18,2.22,2.47,2.32)
multiply<- function(x,factor){
  x*factor
}
#Now we give the lapply function a number to get the multiply function done.And we want the result in a
times3<-lapply(oil_prices,multiply,factor=3)
unlist(times3)
```

```
## [1] 7.11 7.47 6.54 6.66 7.41 6.96
```

```
#We can set the factor for more results
times4<-lapply(oil_prices,multiply,factor=4)
unlist(times4)
```

```
## [1] 9.48 9.96 8.72 8.88 9.88 9.28
```

**Example 4

```
# The vector pioneers has already been created for you
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

# Split names from birth year
split_math <- strsplit(pioneers, split = ":")
split_math
```

```
## [[1]]
## [1] "GAUSS" "1777"
##
## [[2]]
## [1] "BAYES" "1702"
##
## [[3]]
## [1] "PASCAL" "1623"
##
## [[4]]
## [1] "PEARSON" "1857"
```

```
# Convert to lowercase strings: split_low
split_low<-lapply(split_math,tolower)
split_low
```

```
## [[1]]
## [1] "gauss" "1777"
##
## [[2]]
## [1] "bayes" "1702"
##
## [[3]]
## [1] "pascal" "1623"
##
## [[4]]
## [1] "pearson" "1857"
```

```
# Take a look at the structure of split_low
str(split_low)
```

```
## List of 4
##  $ : chr [1:2] "gauss" "1777"
##  $ : chr [1:2] "bayes" "1702"
##  $ : chr [1:2] "pascal" "1623"
##  $ : chr [1:2] "pearson" "1857"
```

**Example 5

```
# Code from previous exercise:
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")
split <- strsplit(pioneers, split = ":")
split_low <- lapply(split, tolower)

# Write function select_first()
select_first <- function(x) {
  x[1]
```

```r
}

# Apply select_first() over split_low: names
names<- lapply(split_low,select_first)
names
```

```
## [[1]]
## [1] "gauss"
##
## [[2]]
## [1] "bayes"
##
## [[3]]
## [1] "pascal"
##
## [[4]]
## [1] "pearson"
```

```r
# Write function select_second()
select_second<- function(x){
  x[2]
  }

# Apply select_second() over split_low: years
years<-lapply(split_low,select_second)
years
```

```
## [[1]]
## [1] "1777"
##
## [[2]]
## [1] "1702"
##
## [[3]]
## [1] "1623"
##
## [[4]]
## [1] "1857"
```

lapply and anonymous functions

Functions in R are objects in their own right. This means that they aren't automatically bound to a name. When you create a function, you can use the assignment operator to give the function a name. It's perfectly possible, however, to not give the function a name. This is called an anonymous function:

```r
# Named function
triple <- function(x) { 3 * x }

# Anonymous function with same implementation
function(x) { 3 * x }
```

```
## function(x) { 3 * x }
```

```r
# Use anonymous function inside lapply()
lapply(list(1,2,3), function(x) { 3 * x })
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 9
```

lapply with additional arguments

lapply() provides a way to handle functions that require more than one argument, such as the multiply() function:

```
multiply <- function(x, factor) {
  x * factor
}
lapply(list(1,2,3), multiply, factor = 3)
```

```
# Definition of split_low
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")
split <- strsplit(pioneers, split = ":")
split_low <- lapply(split, tolower)

# Generic select function
select_el <- function(x, index) {
  x[index]
}

# Use lapply() twice on split_low: names and years
names<-lapply(split_low,select_el,index=1)
years<-lapply(split_low,select_el,index=2)
names
```

```
## [[1]]
## [1] "gauss"
##
## [[2]]
## [1] "bayes"
##
## [[3]]
## [1] "pascal"
##
## [[4]]
## [1] "pearson"
```

```
years
```

```
## [[1]]
## [1] "1777"
##
## [[2]]
## [1] "1702"
##
## [[3]]
## [1] "1623"
```

```
##
## [[4]]
## [1] "1857"
```

sapply

We can use sapply() similar to how we used lapply(). The first argument of sapply() is the list or vector X over which you want to apply a function, FUN. Potential additional arguments to this function are specified afterwards (…):

```
sapply(X, FUN, ...)
```

lapply() returns a list, while sapply() returns a vector that is a simplified version of this list. -If R is not able to simplify, both functions will have same result. -If R finds NULLS won't create a vector with NULLS it will remain as a list, in this case lapply=sapply.

**Example: see how sapply helps to simplify.

```
temp<- list(c(3,7,9,6,-1),c(6,9,12,13,5),c(4,8,3,-1,-3),c(1,4,7,2,-2),c(5,7,9,4,2),c(-3,5,8,9,4),c(3,6,
# Create a function that returns min and max of a vector: extremes
extremes <- function(x) {
  c(min = min(x), max = max(x))
}

# Apply extremes() over temp with sapply()
sapply(temp, extremes)
```

```
##     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## min   -1    5   -3   -2    2   -3    1
## max    9   13    8    7    9    9    9
```

```
# Apply extremes() over temp with lapply()
lapply(temp, extremes)
```

```
## [[1]]
## min max
##  -1   9
##
## [[2]]
## min max
##   5  13
##
## [[3]]
## min max
##  -3   8
##
## [[4]]
## min max
##  -2   7
##
## [[5]]
## min max
##   2   9
##
## [[6]]
## min max
##  -3   9
##
## [[7]]
## min max
##   1   9
```

vapply

The main difference is that we have to give the structure of the result. vapply() can be considered a more robust version of sapply(), because you explicitly restrict the output of the function you want to apply.

```
  vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

**Example1: Here we had to specify R the kind of FUN.VALUE=numeric and the number of that argument =3 refered to (min, mean, max)

```
# temp is already available in the workspace
temp
```

```
## [[1]]
## [1]  3  7  9  6 -1
##
## [[2]]
## [1]  6  9 12 13  5
##
## [[3]]
## [1]  4  8  3 -1 -3
##
## [[4]]
## [1]  1  4  7  2 -2
##
## [[5]]
## [1] 5 7 9 4 2
##
## [[6]]
## [1] -3  5  8  9  4
##
## [[7]]
## [1] 3 6 9 4 1
```

```
# Definition of basics()
basics <- function(x) {
  c(min = min(x), mean = mean(x), max = max(x))
}

# Apply basics() over temp using vapply()
vapply(temp, basics, numeric(3))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## min  -1.0    5 -3.0 -2.0  2.0 -3.0  1.0
## mean  4.8    9  2.2  2.4  5.4  4.6  4.6
## max   9.0   13  8.0  7.0  9.0  9.0  9.0
```

Recap Apply Family

- lapply() Apply function over a list or vector. output = list
- sapply() Apply function over a list or vector. Try to simplify list to array
- vapply() Apply function over a list or vector. Explicity specify output format

Useful Functions

-sort() with a decreasing argument set FALSE for default.

-str() inspect the structures.

-print()

-identical()

-mean()

-sum()

-abs() returs the absolute values.

-round() Round the values to 0 decimal places by default. Try out ?round for variations.

-seq() Generate sequences, by specifying the from, to, and by arguments.

```r
vector1<-seq(8,2,by=-2)
```

-rep() replicates its imput as many times as indicated. Each can be used too.

```r
vector1<-seq(8,2,by=-2)
rep(vector1, times=2)
```

```
## [1] 8 6 4 2 8 6 4 2
```

```r
rep(vector1, each=2)
```

```
## [1] 8 8 6 6 4 4 2 2
```

-is.*() gives a logical answer

```r
#we ask if c() is a list, it has to return FALSE
is.list(c(1,2,3))
```

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

-as.*() Convert an R object from one class to another.

```r
aaa<- as.list(c(1,2,3))
#if we ask now the fuction is.* to aaa it will be true since we have just convert it
is.list(aaa)
```

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

-unlist() Flatten (possibly embedded) lists to produce a vector.

-append() add elements to a vector or list.Merge vectors or lists.

-rev() reverses the elements.

———————— jump this chapter————————

Regular Expressions (for text)

grepl(), grep()

- grepl(), which returns TRUE when a pattern is found in the corresponding character string.
- grep(), which returns a vector of indices of the character strings that contains the pattern.

Both functions need a pattern and an x argument, where pattern is the regular expression you want to match for, and the x argument is the character vector from which matches should be sought.

You can use the caret, ˆ, and the dollar sign, $ to match the content located in the start and end of a string, respectively. This could take us one step closer to a correct pattern for matching only the ".edu" email addresses from our list of emails. But there's more that can be added to make the pattern more robust:

- @, because a valid email must contain an at-sign.
- ., *which matches any character (.) zero or more times ()*. Both the dot and the asterisk are metacharacters. You can use them to match any character between the at-sign and the ".edu" portion of an email address.
- \.edu$, to match the ".edu" part of the email at the end of the string. The \ part escapes the dot: it tells R that you want to use the . as an actual character.

```r
# The emails vector has already been defined for you
emails <- c("john.doe@ivyleague.edu", "education@world.gov", "dalai.lama@peace.org",
            "invalid.edu", "quant@bigdatacollege.edu", "cookie.monster@sesame.tv")

# Use grepl() to match for .edu addresses more robustly
grepl("@.*\\.edu$",emails)
```

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

```r
# Use grep() to match for .edu addresses more robustly, save result to hits
hits<-grep("@.*\\.edu$",emails)

# Subset emails using hits
emails[hits]
```

```
## [1] "john.doe@ivyleague.edu"   "quant@bigdatacollege.edu"
```

sub() gsub()

You can specify a replacement argument. If inside the character vector x, the regular expression pattern is found, the matching element(s) will be replaced with replacement.sub() only replaces the first match, whereas gsub() replaces all matches.

```r
# The emails vector has already been defined for you
emails <- c("john.doe@ivyleague.edu", "education@world.gov", "global@peace.org",
            "invalid.edu", "quant@bigdatacollege.edu", "cookie.monster@sesame.tv")

# Use sub() to convert the email domains to datacamp.edu
sub("@.*\\.edu$","@datacamp.edu",emails)
```

```
## [1] "john.doe@datacamp.edu"    "education@world.gov"
## [3] "global@peace.org"         "invalid.edu"
## [5] "quant@datacamp.edu"       "cookie.monster@sesame.tv"
```

Times and dates

To create a Date object from a simple character string in R, you can use the as.Date() function. The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 13 January, 1982):

```
%Y: 4-digit year (1982)
%y: 2-digit year (82)
%m: 2-digit month (01)
%d: 2-digit day of the month (13)
%A: weekday (Wednesday)
%a: abbreviated weekday (Wed)
%B: month (January)
%b: abbreviated month (Jan)
```

The following R commands will all create the same Date object for the 13th day in January of 1982:

as.Date("1982-01-13") as.Date("Jan-13-82", format = "%b-%d-%y") as.Date("13 January, 1982", format = "%d %B, %Y")

Notice that the first line here did not need a format argument, because by default R matches your character string to the formats "%Y-%m-%d" or "%Y/%m/%d".

In addition to creating dates, you can also convert dates to character strings that use a different date notation. For this, you use the format() function. Try the following lines of code:

today <- Sys.Date() format(Sys.Date(), format = "%d %B, %Y") format(Sys.Date(), format = "Today is a %A!")

——- EXTRA EXAMPLES———