R: lo básico

1. El espacio de trabajo (Workspace)

El *espacio de trabajo* es el ambiente actual de trabajo en `R`. Incluye todos los objetos definidos por el usuario (vectores, matrices, funciones, dataframes, listas).

Una sesión de R inicia cuando abres la consola. Al terminar el trabajo se puede guardar la imagen del espacio de trabajo tal cual está, de manera que sea posible continuar *desde donde te quedaste* (**kabacoff2015r**).

1.1. Directorio de trabajo

El directorio de trabajo (*working directory*) es el directorio en tu computadora en el que estás trabajando en ese momento. Cuando se le pide a R que abra un archivo o guarde ciertos datos, R lo hará a partir del directorio de trabajo que le hayas fijado.

Para saber en qué directorio te encuentras, se usa el comando `getwd()`.

> Usa la mnemotécnica del inglés: *get working directory* ≡ *getwd*. Notarás como muchas funciones tienen un nombre que acorta lo que hacen.

```
getwd()
```

```
## [1] "/home/animalito/study/aprendeR/01_programacion_basica"
```

Para especificar el directorio de trabajo, se utiliza el comando `setwd()` (*set working directory*) en la consola. Y volvemos a

```
setwd("/home/animalito/study/")
getwd()
```

Ejercicio

1. Abre tu consola de `R` y escribe *setwd(“/*.
2. Utiliza la tecla `tab` ⇥ para autocompletar las posibles rutas desde donde quiera que estes.
3. Escoge alguna (nuevamente usando la tecla tab para moverte entre las opciones). Si esto no funciona, teclea textualmente alguna de las rutas que ves.
4. Cierra la doble comilla y el paréntesis.
5. Teclea enter.
6. Debes encontrarte en la ruta elegida cuando tecleas `getwd()`.

Con lo que acabamos de hacer, R buscará archivos o guardará archivos en el folder `/home/animalito/study/`. En R también es posible navegar a partir de el directorio de trabajo. Como siempre,

- “../un_archivo.R” le indica a R que busque un folder arriba del actual directorio de trabajo por el archivo *un_archivo.R*.
- “datos/otro_archivo.R” hace que se busque en el directorio de trabajo, dentro del folder *datos* por el archivo *otro_archivo.R*

> **Rutas relativas vs. Rutas absolutas**
>
> El resultado que se muestra aquí al usar el comando `getwd()` depende de la computadora en la que se esta trabajando debido a que es una *ruta absoluta*. Nota como es diferente la ruta que obtienes al

correr el comando en tu consola de `R`. Eso es porque se trata de una ruta absoluta, es decir, es tal que da la ruta (*path*) completo al directorio en cuestión. Puedes accesar todos los directorios o archivos usando su ruta absoluta.

En investigación reproducible (*reproducible research*), en investigación colaborativa o incluso cuando trabajas en varias computadoras es una buena idea usar rutas relativas en lugar de absolutas. Esto hace que el código sea menos dependiente de una estructura de archivos o computadora en particular (**gandrud2013**).

En general, es *buena práctica* configurar el código de un proyecto con rutas relativas. En `R` en particular, cuando guardas un `Rmarkdown` y lo corres desde la línea de comandos (o lo *tejes* desde `RStudio`), la ruta que está fija -como si hubieras usado el comando `setwd()` es en donde *vive* ese archivo, es decir, el directorio en donde está guardado el mismo.

Desde cualquier `script` puedes llamar a otros usando este tipo de ruta como en el ejemplo anterior.

1.2. Ejemplos básicos

La consola permite hacer operaciones sobre números o caracteres (cuando tiene sentido).

```
# Potencias, sumas, multiplicaciones
2^3 + 67 * 4 - (45 + 5)
```

```
## [1] 226
```

```
# Comparaciones
56 > 78
```

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

```
34 <= 34
```

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

```
234 < 345
```

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

```
"hola" == "hola"
```

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

```
"buu" != "yay"
```

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

```
# módulo
10 %% 4
```

```
## [1] 2
```

Estas operaciones también pueden ser realizadas entre vectores[1].

```
# Creamos un vector con entradas del -1 al 12 y lo asignamos a la variable x
x <- -1:12
# Lo vemos
x
```

[1] Revisaremos más adelante con detalle la definición de vectores en la sección 6.

```
##  [1] -1  0  1  2  3  4  5  6  7  8  9 10 11 12
```

```
# Le sumamos 1 a todas las entradas
x + 1
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13
```

```
# Multiplicamos por 2 cada entrada y le sumamos 3
2 * x + 3
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27
```

```
# Sacamos el módulo de cada entrada
x %% 5
```

```
##  [1] 4 0 1 2 3 4 0 1 2 3 4 0 1 2
```

1.3. Comandos útiles

Para enlistar los objetos que están en el espacio de trabajo

```
ls()
```

```
## [1] "x"
```

Para eliminar todos los objetos en un workspace

```
rm(list = ls()) # se puede borrar solo uno, por ejemplo, nombrándolo
ls()
```

```
## character(0)
```

También se puede utilizar/guardar la historia de comandos utilizados

```
history()
history(max.show = 5)
history(max.show = Inf) # Muestra toda la historia

# Se puede salvar la historia de comandos a un archivo
savehistory(file = "mihistoria") # Por default, R ya hace esto
# en un archivo ".Rhistory"

# Cargar al espacio de trabajo actual (current workspace) una
# historia de comandos en particular
loadhistory(file = "mihistoria")
```

Es posible también guardar el workspace -en forma completa- en un archivo con el comando `save.image()` a un archivo con extensión *.RData*. Puedes guardar una lista de objetos específica a un archivo *.RData*. Por ejemplo:

```
x <- 1:12
y <- 3:45
save(x, y, file = "ejemplo.RData") #la extensión puede ser arbitraria.
```

Después puedo cargar ese archivo. Prueba hacer:

```
rm(list = ls()) # limpiamos workspace
load(file = "ejemplo.RData") #la extensión puede ser arbitraria.
ls()
```

Nota como los objetos preservan el nombre con el que fueron guardados.

2. Librerías

R puede hacer muchos análisis estadísticos y de datos. Las diferentes capacidades están organizadas en paquetes o librerías. Con la instalación estándar se instalan también los paquetes más comunes. Para obtener una lista de todos los paquetes instalados se puede utilizar el comando `library()` en la consola o en un script.

Existen una gran cantidad de paquetes disponibles además de los incluidos por omisión (*default*).

2.1. CRAN

Comprehensive R Archive Network [@cran] es una colección de sitios que contienen exactamente el mismo material, es decir, son espejos (*mirrors*) de las distribuciones de R, las extensiones, la documentación y los binarios. El master de CRAN está en Wirtschaftsuniversität Wien en Austria. Éste se "espeja" (*mirrors*) en forma diaria a muchos sitios alrededor del mundo. En la lista de espejos se puede ver que para México están disponibles el espejo del ITAM, del Colegio de Postgraduados (Texcoco) y Jellyfish Foundation [@cran].

Los espejos son importantes pues, cada vez que busquen instalar paquetes, se les preguntará qué espejo quieren utilizar para la sesión en cuestión. Del espejo que selecciones, será del cuál R *bajará* el binario y la documentación.

Del CRAN es que se obtiene la última versión oficial de R. Diario se actualizan los espejos. Para más detalles consultar el FAQ.

Para contribuir un paquete en CRAN se deben seguir las instrucciones aquí.

2.2. Github

Git es un controlador de versiones muy popular para desarrollar software. Cuando se combina con GitHub se puede compartir el código con el resto de la comunidad. Éste controlador de versiones es el más popular entre los que contribuyen a R. Muchos problemas a los que uno se enfrenta alguien ya los desarrolló y no necesariamente publicó el paquete en CRAN. Para instalar algún paquete desde GitHub, se pueden seguir las instrucciones siguientes

```
install.packages("devtools")
devtools::install_github("username/packagename")
```

Donde `username` es el usuario de Github y `packagename` es el nombre del repositorio que contiene el paquete. Cuidado, no todo repositorio en GitHub es un paquete. Para más información ver el capítulo Git and GitHub en **wickham2015r**

2.3. Otras fuentes

Otros lugares en donde es común que se publiquen paquetes es en Bioconductor un projecto de software para la comprensión de datos del genoma humano.

3. Paquetes recomendados

Hay muchísimas librerías y lo recomendable es, dado un problema y un modelo para resolverlo, revisar si alguien ya implementó el método en algunas de las fuentes de paquetes mencionadas antes.

Para mantener orden en los paquetes descargados puede ser útil utilizar el **pacman** pues provee de herramientas para instalar paquetes en una forma un poco más sencilla que usando la función `install.packages`.

En particular, la función `p_load` permite instalar, cargar y actualizar uno o varios paquetes.

Si queremos instalar varios paquetes usando las herramientas del R básico (*base*) [@rbase] haríamos algo como (**pacman**):

```
packs <- c("XML", "devtools", "RCurl", "fakePackage", "SPSSemulate")
success <- suppressWarnings(sapply(packs, require, character.only = TRUE))
install.packages(names(success)[!success])
sapply(names(success)[!success], require, character.only = TRUE)
```

Con `pacman::p_load` la tarea se reduce a:

```
pacman::p_load(XML, devtools, RCurl, fakePackage, SPSSemulate)
```

> Nota como se puede llamar a una función por su nombre `p_load` una vez que ya cargamos el paquete en el cuál esa función está guardada con el comando `library(pacman)` o podemos llamarla directamente utilizando la convención `paquete::funcion`, en este caso, `pacman::p_load`.

Para instalar `pacman` escribe:

```
install.packages("pacman")
```

Algunos paquetes se encuentran en desarrollo. En particular, si se encuentran en `github` pueden descargarse usando la función `pacman::p_install_gh('usuario/repositorio')`.

A continuación, hay una lista de paquetes que se recomienda descargar para tener a la mano herramientas diversas útiles para el trabajo del científico de datos. La lista no es comprensiva pues hay un gran número de paquetes útiles.

```
# Para cargar datos al ambiente de trabajo (data load)
pacman::p_load(RODBC, RMySQL, RPostgreSQL, RSQLite, foreign, Rpostgres, haven
               , readr)
pacman::p_install_gh("hadley/readxl")
pacman::p_install_gh("rstats-db/RPostgres")

# Para manipular datos (data manipulation)
pacman::p_load(plyr, dplyr, data.table, tidyr, stringr, lubridate, gsubfn)

# Para visualizar datos (data visualization)
pacman::p_load(ggplot2, graphics, ggvis)
pacman::p_install_gh("RcppCore/Rcpp")
pacman::p_install_gh("rstats-db/DBI")
pacman::p_install_gh('ramnathv/htmlwidgets')
pacman::p_install_gh('rstudio/leaflet')
pacman::p_install_gh('bwlewis/rthreejs')
pacman::p_install_gh('htmlwidgets/sparkline')
pacman::p_load(dygraphs, DT, DiagrammeR, networkD3, googleVis)

# Para modelar (data modelling)
pacman::p_load(car, mgcv, lme4, nlme, randomForest, multcomp, vcd, glmnet, survival, caret)

# Para generar reportes (reports)
pacman::p_load(shiny, xtable, knitr, rmarkdown)

# Para trabajar con datos espaciales (spatial data)
pacman::p_load(sp, maptools, maps, ggmap, rgdal)
```

```
# Para trabajo con series de tiempo (time series)
pacman::p_load(zoo, quantmod)

# Para escribir código de alto rendimiento en R (High performance R code)
pacman::p_load(Rcpp, parallel)

# Trabajar con la web
pacman::p_load(XML, jsonlite, httr)

# Para escribir paquetes en R
pacman::p_load(devtools, testthat, roxygen2)
```

4. Scripting

R es un intérprete. Utiliza un ambiente basado en línea de comandos. Por ende, es necesario escribir la secuencia de comandos que se desea realizar a diferencia de otras herramientas en donde es posible utlizar el mouse o menús.

Aunque los comandos pueden ser ejecutados directamente en consola una única vez, también es posible guardarlos en archivos conocidos como *scripts*. Típicamente, utilizamos la extensión **.R** o **.r**. En RStudio, `CTRL + SHIFT + N` abre inmediatamente un nuevo editor en el panel superior izquierdo.

Se puede *ir editando* el script y corriendo los comandos línea por línea con `CTRL + ENTER`. Esto también aplica para *correr* una selección del texto editable.

Es posible también correr todo el script

```
source("foo.R")
```

O con el atajo `CTRL + SHIFT + S` en RStudio.

Para enlistar algunos shortcuts comunes en RStudio presiona `ALT + SHIFT + K`. De la misma manera, si utilizas Emacs + ESS, existen múltiples atajos de teclado para realizar todo mucho más eficientemente. Estudiarlos no es tiempo perdido.

5. Ayuda y documentación

R tiene mucha documentación. Desde la consola se puede accesar a la misma.

Para ayuda general,

```
help.start()
```

Para la ayuda de una función en especifico, por ejemplo, si se quiere graficar algo y sabemos que existe la funcion `plot` podemos consultar fácilmente la ayuda.

```
help(plot)
# o tecleando directamente
?plot
```

El segundo ejemplo se puede extender para buscar esa función en todos los paquetes que tengo instalados en mi ambiente al escribir `??plot`.

La documetnación normalmente se acompaña de ejemplos. Para *correr* los ejemplos sin necesidad de copiar y pegar, prueba

```r
example(plot)
```

Para búsquedas más comprensivas, se puede buscar de otras maneras:

```r
apropos("foo") # Enlista todas las funciones que contengan la cadena "foo"
RSiteSearch("foo") # Busca por la cadena "foo" en todos los manuales de ayuda
# y listas de distribución.
```

6. Estructuras de datos

R tiene diferentes tipos y estructuras de datos que permiten al usuario aprovechar el lenguaje. La manipulación de estos objetos es algo que se hace diario y entender cómo operarlos o cómo convertir de una a otra es muy útil.

En R

- Todo lo que existe es un objeto.
- Todo lo que sucede es una llamada a una función.

6.1. Clases atómicas (atomic classes)

R tiene 6 clases atómicas.

- character (*caracter*)
- numeric (números reales o decimales)
- integer (números enteros)
- logical (booleanos, i.e. falsos-verdaderos)
- complex (números complejos)

Tipo	Ejemplo
Caracter	"hola", "x"
Numérico	67, 45.5
Integer	2L, 67L
Lógico	TRUE, FALSE, T, F
Complejo	1 + 4i

Cuadro 1: Clases atómicas.

Algunos comandos importantes para las clases atómicas son su tipo `typeof()`, su tamaño `length()` y sus atributos `attributes()`, es decir, sus metadatos.

```r
############ Ejemplo 1

x <- "una cadena"
typeof(x)
```

```
## [1] "character"
```

```r
length(x) # tamaño: cuántas cadenas son?
```

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

```r
nchar(x) # Número de caracteres
```

```
## [1] 10
```

```r
attributes(x) # Le pusimos metadatos?
```

```
## NULL
```

```r
############ Ejemplo 2

y <- 1:10
typeof(y)
```

```
## [1] "integer"
```

```r
length(y)
```

```
## [1] 10
```

```r
attributes(y)
```

```
## NULL
```

```r
############ Ejemplo 3

z <- c(1L, 2L, 3L) # Nota como para denotar enteros debes incluir una L al final
typeof(z)
```

```
## [1] "integer"
```

```r
length(z)
```

```
## [1] 3
```

6.2. Vectores

Los vectores son la estructura de datos más común y básica de R. Hay dos tipos de vectores: vectores atómicos y listas.

Típicamente -en libros, blogs, manuales, cuando se mencionan vectores se refieren a los atómicos y no a las listas.

6.2.1. Vectores atómicos

Un vector es un conjunto de elementos con alguna de las clases atómicas, es decir, `character`, `logical`, `integer`, `numeric`. Se puede crear un vector vacío con el comando `vector()` así como especificar su tamaño y su clase.

```r
v <- vector()
v
```

```
## logical(0)
```

```r
## Especifico clase y longitud
vector("character", length = 10)
```

```
##  [1] "" "" "" "" "" "" "" "" "" ""
```

```r
## Lo mismo pero usando un wrapper
character(10)
```

```
##  [1] "" "" "" "" "" "" "" "" "" ""
```

```
## Numerico de tamaño 5
numeric(5)
```

```
## [1] 0 0 0 0 0
```

```
## Lógico tamaño 5
logical(5)
```

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

Realiza los siguientes ejemplos en la consola de R.

```
x <- rep(1, 5)
x
typeof(x)

xi <- c(1L, 3L, 56L, 4L)
xi
typeof(xi)

y <- c(T, F, T, F, F, T)

z <- c("a", "aba", "andrea", "b", "bueno")
class(z)
str(z)
```

Operaciones con vectores

Accesar partes del vector.

```
a <- c(1:5)
a
```

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

```
a[1]
```

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

```
a[2]
```

```
## [1] 2
```

```
a[4:5]
```

```
## [1] 4 5
```

Aritmética: por default, se realizan componente a componente.

```
b <- a + 10
b
```

```
## [1] 11 12 13 14 15
```

```
c <- sqrt(b) # square root = raíz
c
```

```
## [1] 3.316625 3.464102 3.605551 3.741657 3.872983
```

```
a + c
```

```
## [1] 4.316625 5.464102 6.605551 7.741657 8.872983
```

```
10 * (a + c)
```

```
## [1] 43.16625 54.64102 66.05551 77.41657 88.72983
```

```
a^2
```

```
## [1]  1  4  9 16 25
```

```
a * c
```

```
## [1]  3.316625  6.928203 10.816654 14.966630 19.364917
```

Agregar elementos aun vector ya creado

```
a <- c(a, 7)
a
```

```
## [1] 1 2 3 4 5 7
```

Para construir datos rápido, podemos usar comandos como `rep`, `seq` o distintas distribuciones, e.g., la normal `rnorm`, uniformes `runif` o cualquiera en esta lista.

Prueba lo siguiente:

```
# Dame un vector donde el minimo sea 0, maximo 1 en intervalos de 0.25
seq(0, 1, 0.25)
# Vector con 10 unos
rep(1, 10)
# 5 realizaciones de una normal(0,1)
rnorm(5)
# De una normal(10, 5)
rnorm(5, mean = 10, sd = sqrt(5))
# De una uniforme(0,1)
runif(5)
# De una uniforme(5, 15)
runif(5, min = 5, max = 15)
```

6.2.1.1. Otros objetos importantes

`Inf` es como R denomina al infinito. En el mundo de R se permite también positivo o negativo.

```
1/0
```

```
## [1] Inf
```

```
1/Inf
```

```
## [1] 0
```

`NaN` es como R denota a algo que no es un número (literal: *not a number*).

```
0/0
```

```
## [1] NaN
```

Cada objeto tiene atributos. Hay atributos específicos para vectores que, sin importar su clase, tienen en común. Ya revisamos algunos: tamaño (`length`), clase (`class`). También son importantes atributos como los nombres

```
calificaciones <- c(6, 5, 8, 9, 10)
names(calificaciones) <- c("Maria", "Jorge", "Miguel", "Raúl", "Carla")
attributes(calificaciones)
```

```
## $names
## [1] "Maria"  "Jorge"  "Miguel" "Raúl"   "Carla"
```

```
# O llamamos directo a los nombres
names(calificaciones)
```

```
## [1] "Maria"  "Jorge"  "Miguel" "Raúl"   "Carla"
```

Mezclar tipos no es una buena idea

```
c(1.7, "a")
```

```
## [1] "1.7" "a"
```

```
c(TRUE, 2)
```

```
## [1] 1 2
```

```
c("a", TRUE)
```

```
## [1] "a"    "TRUE"
```

R realiza una coerción implícita entre los objetos y "decide" cuál es la clase del vector. También hay coercioń explícita (*explicit coercion*) utilizando `as.<nombre_clase>`.

```
as.numeric()
as.character()
as.integer()
as.logical()
```

Muchos problemas suceden cuando le permites a R decidir por ti (o cuando no sabes cuál decisión tomará R por *default*).

```
x <- 0:5

identical(x, as.numeric(x))
```

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

En este ejemplo, cuando declaramos x no especificamos su clase y R decidió que era entero. Al coercionar al objeto para que fuese numérico, R no considera a los dos objetos iguales. R te protege -no te permite hacer o te advierte- de algunas cosas

```
1 < "2"
```

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

Pero en otras, hace lo mejor que puede con lo que le das (cosa que a veces no tiene sentido)

```
x <- c("a", "b", "c")
as.numeric(x)
```

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

```
as.logical(x)
```

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

6.2.2. Matrices

Las matrices son un tipo especial de vectores. Son un vector atómico con dimensión pues tienen filas y columnas.

```r
m <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Como puedes notar, las matrices se llenan siguiendo las columnas. Podemos simplemente "agregarle" una dimensión a un vector para construir una matriz.

```r
m <- 1:10
m
```

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

```r
dim(m) <- c(2, 5)
m
```

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

También podemos pegar vectores de la misma longitud como si fueran columnas de una matriz `cbind` o como si fueran filas `rbind` (r = row, c = column).

```r
x <- runif(4)
y <- rnorm(4)
cbind(x, y)
```

```
##               x          y
## [1,] 0.18681273 -1.1640959
## [2,] 0.02267067 -2.3134259
## [3,] 0.91348432  0.3100017
## [4,] 0.99191958  0.2299862
```

```r
rbind(x, y)
```

```
##         [,1]        [,2]      [,3]      [,4]
## x  0.1868127  0.02267067 0.9134843 0.9919196
## y -1.1640959 -2.31342586 0.3100017 0.2299862
```

Le agregamos atributos para accesar más fácilmente a los objetos.

```r
m <- matrix(c(x, y), nrow = 4, ncol = 2, byrow = T,
            dimnames = list(paste0("row", 1:4),
                            paste0("col", 1:2)))
m
```

```
##            col1        col2
## row1  0.1868127  0.02267067
## row2  0.9134843  0.99191958
## row3 -1.1640959 -2.31342586
## row4  0.3100017  0.22998623
```

```r
m[1, 1] == m["row1", "col1"]
```

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

```r
dimnames(m)
```

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

```
## [1] "row1" "row2" "row3" "row4"
##
## [[2]]
## [1] "col1" "col2"
```

6.2.3. Listas

Es un tipo de vector en el cuál cada elemento puede ser de un tipo distinto. Mas aun, es posible incluir una lista como un elemento de otra lista y por eso también se les conoce como vectores recursivos (*recursive vectors*).

Para crear una lista vacía utilizas `list()` y para coercionar un objeto a una lista usa `as.list()`.

```
x <- list(3L, 3.56, 1 + 4i, TRUE, "hola", list("genial", 1))
x
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 3.56
##
## [[3]]
## [1] 1+4i
##
## [[4]]
## [1] TRUE
##
## [[5]]
## [1] "hola"
##
## [[6]]
## [[6]][[1]]
## [1] "genial"
##
## [[6]][[2]]
## [1] 1
```

```
length(x)
```

```
## [1] 6
```

```
class(x)
```

```
## [1] "list"
```

```
class(x[1])
```

```
## [1] "list"
```

```
class(x[[1]])
```

```
## [1] "integer"
```

```
y <- as.list(1:10)
length(y)
```

```
## [1] 10
```

Nota como muchas propiedades que tenían los vectores atómicos los tienen también las listas. Por su propiedad recursiva, se navega diferente. Si pides `x[1]` te devuelve una lista con lo que hayas puesto en ese contenedor.

Para extraer el objeto (con la clase de ese objeto y no simplemente otra lista) necesitas usar `x[[1]]`, es decir, el integer 3.

Las listas también pueden tener nombres

```
# Lista vacia
lista <- list()
lista[["numeros"]] <- c(1, 34, 45.5, 34)
lista[["datos"]] <- head(iris)

lista
```

```
## $numeros
## [1]  1.0 34.0 45.5 34.0
##
## $datos
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

> R tiene muchos datos de ejemplo que son utilizados en muchos paquetes, blogs y libros. Utiliza **help(iris)** para saber más del dataset usado arriba.

6.2.4. Factores (factor)

Otro tipo de vector pero que ayuda a representar datos del tipo categórico u ordinal. Es muy importante decirle a R que algo debe ser tratado como factor cuando se empieza a modelar o incluso para que los métodos de gráficos funcionen de manera apropiada. Sin embargo, hay que entender bien cómo tratarlos porque mal usados hacen que pasen muchas cosas muy raras que dan resultados que estan *mal, mal, mal*.

Los factores son enteros pero con etiquetas encima.

```
x <- factor(c("no", "si", "si", "no"))
x
```

```
## [1] no si si no
## Levels: no si
```

Lo que te deja utilizar métodos para factores como tablas de frecuencias

```
table(x)
```

```
## x
## no si
##  2  2
```

Los factores se van a ver *como si fueran* vectores tipo caracter. A veces se comportan como character vectors pero *debemos* recordar que por abajo son integers y tenemos que ser cuidadosos si los tratamos como caracteres. Algunos métodos que están hechos para caracteres coersionan un factor a caracter mientras que otros arrojan un error. Si usas métodos de caracteres, lo mejor es "castear" a caracter tu factor `as.character(mifactor)`. Pierdes algunas cosas pero te aseguras que las cosas funcionen como deben.

```
summary(x)
```

```
## no si
##  2  2
```

```
summary(as.character(x))
```

```
##    Length     Class      Mode
##         4 character character
```

Los factores pueden contener únicamente valores predefinidos. Por eso la "unión" de factores puede tronar.

```
y <- factor(c("si", "no", "tal vez"))
c(x, y)
```

```
## [1] 1 2 2 1 2 1 3
```

```
class(c(x, y))
```

```
## [1] "integer"
```

¿Cómo recuperas el valor de las etiquetas? R hizo lo que pudo y está mal. Para hacerlo bien, debemos

```
factor(c(as.character(x), as.character(y)))
```

```
## [1] no      si      si      no      si      no      tal vez
## Levels: no si tal vez
```

Para datos ordinales como las respuestas en una pregunta de encuesta con escala likert, podemos usar

```
set.seed(2887)
respuestas <- sample(x = c(1:5), size = 5, replace =  T)
respuestas
```

```
## [1] 4 1 4 2 1
```

```
y <- factor(
  x = respuestas,
  levels = c("1", "2", "3", "4", "5"),
  labels = c("muy en contra", "en contra", "indiferente", "a favor", "muy a favor"),
  ordered = T)
y
```

```
## [1] a favor       muy en contra a favor       en contra     muy en contra
## 5 Levels: muy en contra < en contra < indiferente < ... < muy a favor
```

Nota como aunque no tengamos todos las respuestas, nuestro factor sabe que las no ocurrencias son factibles (los niveles y las etiquetas las incluyen).

```
table(y)
```

```
## y
## muy en contra     en contra   indiferente       a favor   muy a favor
##             2             1             0             2             0
```

Nota

En R muchas cosas se reducen a utlizar la estructura de datos apropiada y darle todos los metadatos necesarios al objeto para que R no haga tonterias.

6.3. Data frames

Los dataframes son uno de los objetos más importantes en R. Tanto así que muchos no dejarían R porque implica abandonar este objeto. En python, se intenta replicar este objeto con la librería `pandas`.

Este objeto es tan importante porque muchos de los modelos estadísticos que se utliizan necesitan una estructura de datos tabular.

Los dataframes tienen atributos adicionales a los que tienen los vectores:

- `rownames()`
- `colnames()`
- `names()`
- `head()` te enseña las primeras 6 lineas.
- `tail()` te enseña las últimas 6 líneas.
- `nrow()` te da el número de filas
- `ncol()` te da el número de columnas
- `str()` te dice el tipo de cada columna y te muestra ejemplos

Podemos ver a los dataframes como un tipo de lista restringido a que todos los elementos de ésta tienen la misma longitud o tamaño.

Los dataframes se pueden crear utilizando comandos como `read.table()` (que tiene como caso particular `read.csv()`. Para convertir un dataframe a una matriz se utiliza `data.matrix()`. La coerción es forzada y no necesariamente da lo que uno espera.

Se pueden crear data.frames con la función `data.frame()`.

```
df <- data.frame(
  x = rnorm(10),
  y = runif(10),
  n = LETTERS[1:10],
  stringsAsFactors = F
)

head(df)
```

```
##            x         y n
## 1  0.4923136 0.7117542 A
## 2  1.2949079 0.5276390 B
## 3 -0.2432564 0.4198618 C
## 4  1.1128253 0.6586744 D
## 5 -2.2455891 0.9040571 E
## 6 -1.2421756 0.2724684 F
```

```
dim(df)
```

```
## [1] 10  3
```

```
str(df)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ x: num  0.492 1.295 -0.243 1.113 -2.246 ...
##  $ y: num  0.712 0.528 0.42 0.659 0.904 ...
##  $ n: chr  "A" "B" "C" "D" ...
```

> ¿Por qué usar la opción "stringsAsFactors = F"?

Podemos "pegarle" columnas o filas:

```r
df <- cbind(df, data.frame(z = rexp(10)))
df <- rbind(df, c(rnorm(1), runif(1), "K", rexp(1)))
dim(df)
```

```
## [1] 11  4
```

7. Valores perdidos (missing values)

En la página 10 se habló de otros objetos en R. De particular importancia es `NA` para valores perdidos en general y `NaN` para operaciones matemáticas no definidas. Lógicamente, podemos preguntar a R si un objeto es de este tipo

```r
is.na()
is.nan()
```

Los valores `NA` tienen una clase particular. Puede haber valores perdidos enteros `NA_integer_` o caracteres `NA_character_`. `NaN` es un `NA` pero no al revés.

```r
x <- c(1, 4, 6, NA, NaN, 45)
is.nan(x)
```

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

```r
is.na(x)
```

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

Cuando tenemos un dataframe que tiene valores perdidos y lo queremos incorporar, por ejemplo, a un modelo de regresión, lo primero que hará el método es excluir todos los renglones que tengan *algún* valor perdido usando `na.exclude(datos)`.

8. Estructuras

Las estructuras de control permiten controlar la ejecución. Pueden ser utilizadas en un script o dentro de funciones. Entre las más comunes se encuentran:

- if, else
- for
- while
- repeat
- break
- next
- return

8.1. If

```r
if ( condicion ) {
  # Cuando se cumple la condicion, ejecuta esto
} else {
  # Para todo lo que no se cumple la condicion, ejecuta esto
}
```

Ejemplo,

```r
x <- 1:20
if ( sample(x, 1) <= 10 ) {
  print("x es menor o igual que 10")
} else {
  print("x es mayor que 10")
}
```

```
## [1] "x es menor o igual que 10"
```

O lo que es lo mismo pero un poco mas eficiente (vectorizado)

```r
ifelse(sample(x, 1) <= 10, "x es menor o igual que 10", "x es mayor que 10")
```

```
## [1] "x es mayor que 10"
```

También es posible asignar variables a través condicionando a algo.

```r
if ( sample(x, 1) <= 10 ){
  y <- 0
} else {
  y <- 1
}

# o

y <- if ( sample(x, 1) <= 10 ){
    0
  } else {
    1
  }
```

8.2. For

Un ciclo `for` itera una variable y va realizando, para cada iteración, la secuencia de comandos que se especifica dentro del mismo.

```r
for (i in 1:3 ){
  print(paste0("i vale: ", i))
}
```

```
## [1] "i vale: 1"
## [1] "i vale: 2"
## [1] "i vale: 3"
```

Es posible también iterar directamente sobre vectores o partes de vectores.

```r
x <- c("Andrea", "Liz", "Edwin", "Miguel")

for ( i in x ) {
  print(x[i])
}
```

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

```
for ( e in x ) {
  print(e)
}
```

```
## [1] "Andrea"
## [1] "Liz"
## [1] "Edwin"
## [1] "Miguel"
```

```
for ( i in seq(x) ){
  print(x[i])
}

for ( i in 1:length(x) ) print(x[i])
```

Podemos incluir `fors` dentro de `fors`.

```
m <- matrix(1:10, 2)

for( i in seq(nrow(m)) ) {
  for ( j in seq(ncol(m)) ) {
    print(m[i, j])
  }
}
```

8.3. Whiles

Otra manera de iterar sobre comandos es con la estructura `while`. A diferencia del `for`, esta te permite iterar sobre la secuencia de comandos especificada hasta que se cumpla cierta condición. Esta última tiene que variar a lo largo de las iteraciones o es posible generar ciclos infinitos. Esta estructrura da mucha flexibilidad.

```
x <- runif(1)

while ( x < 0.20 | i <= 10 ) {
  print(x)
  x <- runif(1)
  i <- i + 1
}
```

Importante

Asegurate de especificar una manera de salir de un ciclo while.

8.4. Repeat - Break

```
x <- 1
repeat {
  # Haz algo
  print(x)
  x = x+1
  # Hasta que se cumpla lo siguiente
  if (x == 6){
    break
```

```
  }
}
```

8.5. Next

```
for (i in 1:20) {
  if (i %% 2 == 0){
    next
  } else {
    print(i)
  }
}
```

Este ciclo itera sobre los valores del 1 al 20 e imprime los valores impares.

Importante

R no es muy eficiente cuando se combina con estructuras de control tipo for o while. Sin embargo, estas estructuras son muy comunes y es útil conocerlas.
Normalmente, se recomienda utilizar estructuras vectorizadas (como ifelse) pues, de esta manera, R es mucho más eficiente.

9. Funciones

Hay una regla de oro en programación en general: *dry code*. Básicamente esto se reduce a *no te repitas*. Cuando tienes las mismas líneas de código varias veces (cuando estas copy-pasteando mucho) entonces lo que necesitas es escribir una función que realice esa tarea.

En R las funciones son los *building blocks* de básicamente todo. Como todo lo demás en R, las funciones son también objetos. Por default, los argumentos de una función son *flojos* (lazy), es decir, solamente son evaluados cuando se utilizan (esto es importante pues si tienes un error en una función no te darás cuenta cuando ejecutes la misma sino cuando la mandes llamar). Cuando llamas a un objeto en R, casi siempre estas en realidad llamando a una función.

9.1. Componentes de una función

- El `body()` o cuerpo de la función es el código dentro de la misma.
- `formals()` o el listaod de argumentos formales de la función, controla cómo se puede llamar a una función.
- El ambiente `environment()` determina cómo son referidas las variables dentro de la función.
- La lista de argumentos se obtiene con `args()`

```
f <- function(x) x
f
```

```
## function(x) x
```

```
formals(f)
```

```
## $x
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

```
rm(f)
```

9.2. El ambiente

Las variables que se definen dentro de una función existen en un ambiente distinto al ambiente global de R. Si una variable **no** está definida dentro de la función, R busca en el nivel superior por esa variable.

```
x <- 2
g <- function() {
    y <- 1
    c(x, y)
}
g()
```

```
## [1] 2 1
```

```
rm(x, g)
```

Así como fuimos capaces de anidar ciclos for, también podemos anidar funciones. Esta capacidad es muy útil pero hay que tener cuidado con los ambientes y la jerarquía en los mismos.

```
myfuncion <- function() {
  print("Hola")
}
myfuncion()
```

```
## [1] "Hola"
```

Podemos generar funciones con mayor utilidad.

```
suma <- function(x, y){
  return(x + y)
}
vector <- c(1, 2, 3, 4)
sapply(vector, suma, 2)
```

```
## [1] 3 4 5 6
```

Toda función *regresa* un valor.

```
x <- 10
f <- function() {
    y <- 25
    g <- function() {
        z <- 30
        c(x = x, y = y, z = z)
    }
    g()
}
f()
```

```
##  x  y  z
## 10 25 30
```

```
f <- function(x) {
  x * 2
}
```

```
g <- function(x) {
  x + 2
}
f(g(2))
```

```
## [1] 8
```

```
g(f(2))
```

```
## [1] 6
```

En este caso, utilizamos una función con parámetros que *recibe* cuando es llamada. También podemos generar funciones con valores predefinidos, es decir, defaults. Éstos son utilizados cuando se llama a la función *a menos que* se especifique lo contrario (es decir, se *overide them*).

```
f <- function(a = 2, b = 3) {
  return(a + b)
}
f()
```

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

```
f(4, 5)
```

```
## [1] 9
```

```
f(b = 4)
```

```
## [1] 6
```

> **Return**
>
> No es necesario especificar lo que regresa la función. Las funciones por default regresan el último elemento o valor computado.

9.3. Reglas de scope

Sabemos que existe la función *c* que nos permite concatenar vectores o elementos a vectores. Sin embargo, es posible asignar un valor a una variable llamada *c* y que la función *c* siga funcionando.

```
c <- 1000
c + 1
```

```
## [1] 1001
```

```
x <- c(1:4)
x
```

```
## [1] 1 2 3 4
```

Esto es debido a que R tienen namespaces separados para funciones y no-funciones. Cuando R intenta concatenar los valores del 1 al 4, busca primero en el ambiente global y, en caso de no encontrarlo, busca en los *namespaces* de cada uno de los paquetes que tiene cargados.

El orden en el que busca se puede encontrar utilizando el comando `search()`.

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
```

```
## [7] "package:methods"   "Autoloads"         "package:base"
```

Los paquetes recien llamados acaban en la posición número 2 y todo lo demás se recorre en el orden de la lista. Nota como el *base* (que se carga por default en toda sesión) está hasta el final.

`.GlobalEnv` es el workspace del que hablamos antes. Si hay un símbolo que corresponde a tu petición entonces tomará el valor en tu workspace para poder ejecutar tu petición. Si no encuentra nada, busca en el namespace de cada uno de los paquetes que has cargado hasta el momento en el *orden* en el que los llamaste.

Esto es **muy** importante. Hay contribuidores de paquetes en todo el mundo y es muy común que utilicen el mismo nombre para implementaciones de distintas cosas y, por lo tanto, a veces nuestros resultados no son lo que esperábamos.

```
library(dplyr)
library(plyr)
is.discrete
```

```
## function (x)
## is.factor(x) || is.character(x) || is.logical(x)
## <environment: namespace:plyr>
```

```
library(plyr)
library(dplyr)
rename
```

```
## function (x, replace, warn_missing = TRUE, warn_duplicated = TRUE)
## {
##     names(x) <- revalue(names(x), replace, warn_missing = warn_missing)
##     duplicated_names <- names(x)[duplicated(names(x))]
##     if (warn_duplicated && (length(duplicated_names) > 0L)) {
##         duplicated_names_message <- paste0("`", duplicated_names,
##             "`", collapse = ", ")
##         warning("The plyr::rename operation has created duplicates for the ",
##             "following name(s): (", duplicated_names_message,
##             ")", call. = FALSE)
##     }
##     x
## }
## <environment: namespace:plyr>
```

```
sessionInfo()
```

```
## R version 3.3.1 (2016-06-21)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 15.04
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=es_MX.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=es_MX.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=es_MX.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=es_MX.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
```

```
## [1] plyr_1.8.4       dplyr_0.5.0.9000
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.9     digest_0.6.11   rprojroot_1.2   assertthat_0.1
##  [5] R6_2.2.0        DBI_0.5-14      backports_1.0.5 magrittr_1.5
##  [9] evaluate_0.10   stringi_1.1.2   rmarkdown_1.3   tools_3.3.1
## [13] stringr_1.1.0   yaml_2.1.14     htmltools_0.3.5 knitr_1.15.1
## [17] tibble_1.2
```