David Alejandro Aguirre Bautista
Isaac Guarneros De Anda

Material de Apoyo Curso de Introducción a R

Facultad de Ciencias UNAM

I. Bases

¿Qué es una biblioteca?

Es un conjunto de funciones.

¿Cómo puedo instalar una biblioteca?

Usando el comando **install.packages("Nombre de la biblioteca")**
Ejemplos:
install.packages("dplyr")
install.packages("ggplot2")
Recuerda que instalar una biblioteca no significa que ya esté lista para usarla, para eso es necesario cargarla.

¿Cómo puedo cargar una biblioteca?

Usando la función **library(Nombre de la biblioteca)**
Ejemplos:
library(dplyr)
library(ggplot2)
Después de instalar y cargar la biblioteca ya puedes usarla sin problemas.

¿Cómo puedo saber qué contiene una biblioteca?

En la consola puedes usar los siguientes comandos:
help(Nombre de la biblioteca)
?Nombre de la biblioteca
Ejemplos:
help(dplyr)
?ggplot2

¿Cómo puedo saber qué bibliotecas hemos cargado?

En la consola puedes usar la función:
search( )

Tipos de datos:

- **Numérico:** Como su nombre lo dice, se tratan de variables numéricas. Tienen una asignación simple.
  Ejemplo: **a**← **4**, en la variable **a** estamos guardando el valor de **4**.
- **String/Caracter/Texto:** En este tipo de variables podemos guardar palabras, textos, letras, etc. Al guardar un texto en una variable es importante hacer uso de las comillas (" " o ' ') como se muestra a continuación:
  Ejemplo: **a←"Hola",** con esta instrucción estamos diciendo que en la variable **a** guardaremos la palabra **Hola**

- **Lógico o Booleano:** Las variables de este tipo sólo pueden tener dos valores : **TRUE** o **FALSE**, se refieren a los valores de verdad (lógica), normalmente los utilizaremos al momento de querer buscar que una condición sea válida. Al querer asignarle uno de estos valores a una variable es importante escribir la palabra tal y como se mostró, en mayúsculas.
  Ejemplo: **a**←**TRUE**, en la variable **a** estamos guardando el valor **TRUE**.

Si queremos saber qué tipo de dato es cierta variable, usamos la función **class(variable)**.
Ejemplos:
class(4) al correr esta línea de código nos dará como resultado **"numeric"**
class("casa") al correr esta línea de código nos dará como resultado **"character"**
class(TRUE) al correr esta línea de código nos dará como resultado "**logical"**

Operadores Numéricos:

I. **" + " ,** por medio de este símbolo es que efectuamos una **suma**.
II. **" - " ,** por medio de este símbolo es que efectuamos una **resta**.
III. **" * " ,** por medio del asterisco es que efectuamos una **multiplicación.**
IV. **" / " ,** por medio de la diagonal es que efectuamos una **división.**
V. **"^" ,** por medio de este símbolo es que efectuamos una **potencia**
   Ejemplo: **2^3** , al correr esta línea de código nos dará como resultado 2*2*2 = 8.
VI. **"%/%" ,** por medio de este símbolo es que efectuamos una **división entera**
   Ejemplo: **45 %/% 4**, al correr esta línea de código nos dará como resultado 11, pues esta operación solo nos devuelve la parte entera de la división 45/4.
VII. **"%%" ,** por medio de este símbolo es que obtenemos el **módulo**
   Ejemplo: **38%%6**, al correr esta línea de código nos dará como resultado 2, pues está operación lo que nos devuelve es el residuo de la división 38/6, que en este caso es 2.

Operadores Lógicos:

I. **& ,** significa "y" (Conjunción)
   Recordatorio Tabla de verdad - Conjunciones.
   Sean P y Q dos proposiciones o condiciones.

P	Q	**P & Q**
TRUE	TRUE	**TRUE**
TRUE	FALSE	**FALSE**
FALSE	TRUE	**FALSE**
FALSE	FALSE	**FALSE**

   Notemos que una conjunción es verdadera únicamente cuando **ambas** condiciones se cumplen o son verdaderas, en cualquier otro caso es falsa.

II. **| ,** significa "o" (Disyunción)
   Recordatorio Tabla de verdad - Disyunciones.
   Sean P y Q dos proposiciones o condiciones

P	Q	**P | Q**
TRUE	TRUE	**TRUE**
TRUE	FALSE	**TRUE**
FALSE	TRUE	**TRUE**
FALSE	FALSE	**FALSE**

Notemos que una disyunción es verdadera siempre y cuando al menos una de las proposiciones o condiciones sea verdadera, por otro lado, solo es falsa si ambas condiciones lo son.

III. **!** , significa “no” (Negación).
Al aplicar este operador se cambia el valor guardado en una variable lógica.
Recordatorio Tabla de verdad - Negación
Sean P una proposición o condición.

P	**!P**
TRUE	**FALSE**
FALSE	**TRUE**

Asignar Variables

Al programar es importante asignar variables pues esto nos permite guardar la información para poder realizar operaciones.
Existen varias formas de hacerlo:

- a = 6
- a ← 6 Es la forma más usada por convención del lenguaje R
- 6 → a
- assign(“a”,6)
- a ← b ← 6

Después de correr estas líneas de código, las variables quedan guardadas en nuestro espacio de trabajo.

Getwd y Setwd

Es importante diferenciar estas dos funciones:
setwd - Sirve para asignar un directorio de trabajo.
getwd - Sirve para saber cuál es nuestro directorio de trabajo actual.

Ejemplo:
getwd( )
Nos da como resultado “C:/Users/persona1/Documents” que es el directorio de trabajo actual.

setwd("C:/Users/persona2/Tareas")
Cambia el directorio de trabajo actual y lo establece como "C:/Users/persona2/Tareas".

II. Estructuras de Datos

¿Qué son?
Son objetos que pueden contener más de un valor.

¿Cuáles son?
Vectores, Listas, Matrices y Dataframes.

Vectores:

¿Qué es un vector?
Un vector es un objeto que puede almacenar distintos datos del **mismo tipo**. Si queremos guardar números y palabras en un mismo vector, automáticamente R convertirá los números a caracteres.

¿Cómo se declara un vector?
Con la función **c(valores separados por comas)**
Ejemplos:
mi_vector_1 ← c(5,6,7,8)
mi_vector_2 ← c(1:10) Crea un vector con los números del 1 al 10.
mi_vector_3 ← c("Frutas", "Verduras", "Panes")
mi_vector_4 ← c(TRUE,FALSE,TRUE,FALSE,TRUE)
mi_vector_5 ← c(mode = "numeric" , length = 10)
Crea un vector de datos numéricos de longitud 10, es decir, crea un vector con diez ceros.
mi_vector_6 ← c(mode = "logical" , length = 5)
Crea un vector de datos lógicos de longitud 5, es decir, crea un vector con 5 FALSE.

¿Cómo saber cuántos datos tiene un vector?
Por medio de la función **length(vector)**
Ejemplo:
Tomando los ejemplos de el punto anterior:
length(mi_vector_1) nos dará como resultado 4.
length(mi_vector_3) nos dará como resultado 3.
length(mi_vector_4) nos dará como resultado 5.

Independientemente del tipo de dato que contenga, el vector sólo cuenta la cantidad de datos dentro de este.

¿Cómo saber si un objeto es un vector?
Usamos la función **is.vector(objeto)**
Ejemplo:
is.vector(mi_vector_1) nos dará como resultado TRUE.

¿Cómo obtener elementos en un vector (consultas)?
Tenemos que usar corchetes [ ] después del nombre del vector.
Ejemplos:
mi_vector_1[1] nos da como resultado 5.
mi_vector_2[8:10] nos da como resultado 8, 9 y 10.
mi_vector_4[c(2,4,5)] nos dará como resultado FALSE, FALSE, TRUE.

NOTA: Las posiciones en R se empiezan a contar a partir del 1. Es decir, mi_vector_1[1] hace referencia al primer valor en el vector de izquierda a derecha.

¿Qué es NA?
Son valores faltantes, valores que se dejan en blanco.

¿Cómo identificar que hay un valor NA?
Con la función **is.na(vector)**, el cual nos dará un vector de valores lógicos, si encontramos un **TRUE** dentro de ese vector entonces existe un NA.

¿Qué hace la función **rep(vector)**?
Repite los elementos de un vector.
Ejemplos:
rep(c("A","B"), times = 4) nos dará como resultado "A","B","A","B","A","B","A","B"
rep(c(10,20,30), each = 2) nos dará como resultado 10, 10, 20, 20, 30, 30

¿Qué hace la función **seq(a,b)**?
Crea un vector con los números enteros que van desde **a** hasta **b**.
Ejemplos:
seq(1,10) nos creará el vector (1,2,3,4,5,6,7,8,9,10)
seq(10,5) nos creará el vector (10,9,8,7,6,5)

Notemos que podemos crear sucesiones de números tanto crecientes como decrecientes.
Podemos modificar la distancia que queremos entre cada uno de los números de nuestro vector, para ello debemos agregar "**by=**" dentro de los parámetros de la función. Así, nosotros podremos especificar la distancia que requerimos entre cada uno de los elementos del vector.
Ejemplo:
seq(1,3) = (1,2,3)
seq(1,3, **by= 0.5**) = (1, 1.5, 2, 2.5, 3)
En el segundo caso definimos que la distancia entre cada uno de los elementos sea de 0.5.

También podemos definir el tamaño del vector, esto lo haremos agregando un parámetro a la función, "**length=**".
Ejemplo:
seq(1,3) = (1,2,3)
seq(1,3, by= 0.5) = (1, 1.5, 2, 2.5, 3)
seq(1,3, **length = 6**) = (1, 1.4, 1.8, 2.2, 2.6, 3)
En el tercer caso definimos que el vector tuviera 6 elementos.
NOTA: Fijémonos en que R calcula los elementos de tal modo que cada uno sea **equidistante** tanto de su sucesor como su antecesor, es decir, la distancia de 1.4 a 1 es **0.4** y la distancia de 1.8 a 1.4 es **0.4**

¿Qué hace la función **rev(vector)**?
Invierte las entradas del vector, es decir, cambia la posición de los datos de tal forma que ahora se empieza con el último dato y se termina con el primer dato.
Ejemplo:
rev(mi_vector_1) nos dará como resultado 8,7,6,5

¿Cómo modificar los elementos de un vector?

Para modificar el valor de una de las posiciones dentro de un vector haremos uso de aspectos que hemos visto antes, por un lado tenemos el cómo hacer referencia a una **posición del vector** (punto anterior) y por otro lado, tenemos la **asignación** de valores.
Tomamos los vectores que hemos venido usando a modo de ejemplo.
Si queremos cambiar el valor de la segunda entrada de mi_vector_1 por un 500, aplicamos la siguiente línea de código:

mi_vector_1[2] ← 500

Tras correr esta línea de código nuestro vector tendrá los siguientes valores: (5,**500**,7,8). La forma de entender la línea es bastante intuitiva, con **mi_vector_1[2]** le decimos a R que queremos hacer algo con el segundo elemento de **mi_vector_1** y con ← **500**, le decimos que lo que queremos hacer es asignarle un nuevo valor, en este caso 500.

¿Cómo saber si un valor específico se encuentra en un vector?

Con la función **is.element(valor, vector)**
Ejemplos:
is.element(5, mi_vector_2) nos dará como resultado TRUE (sí está en el vector)
is.element(60, mi_vector_2) nos dará como resultado FALSE (no está en el vector)

Dataframes:

¿Qué es un dataframe?

Un dataframe es una tabla constituída por renglones y columnas.

¿Cómo se crea un dataframe?

Una forma de crear un dataframe es mediante vectores, con la función:
data.frame(vector1, vector2, vector3)
Donde vector1, vector2 y vector3 serán las columnas de la tabla.

Si queremos ponerle nombre a las columnas, debemos hacerlo de la siguiente manera:
data.frame(nombre1 = vector1, nombre2 = vector2, nombre3 = vector3)

¿Cómo visualizar un dataframe?

Con la función **View(dataframe)**

¿Cómo obtener elementos en un dataframe (consultas)?

Para acceder a los elementos de un dataframe necesitamos usar corchetes [ ].
Ejemplo:
mi_tabla ← data.frame(vector1, vector2, vector3) Creamos un dataframe
mi_tabla[1,1] Nos dará el elemento del primer renglón y la primera columna de la tabla.
mi_tabla[1: 5,] Nos dará los primeros 5 renglones de la tabla.
mi_tabla[,2] Nos dará la segunda columna de la tabla.
mi_tabla[, c(1,3)] Nos dará las columnas 1 y 3 de la tabla.
mi_tabla[,] Nos dará la tabla completa.

Por lo tanto, la forma de acceder a los elementos es: **mi_tabla[# renglón, # columna]**

¿Cómo saber los nombres de las columnas de un dataframe?
Con la función **colnames(dataframe)**

¿Para qué sirve el signo de pesos $?
Este símbolo se utiliza para obtener en forma de vector una determinada columna de un dataframe.
dataframe$nombre_columna
Por ejemplo:
mi_tabla ← data.frame(nombre1 = vector1, nombre2 = vector2, nombre3 = vector3)
Creamos un dataframe

mi_tabla$nombre3
Nos da como resultado un vector con los elementos de la tercera columna de la tabla

¿Cómo hacer consultas más avanzadas en un dataframe?
Usemos como ejemplo de dataframe a mtcars:
mtcars[mtcars$cyl == 8,] Nos da los renglones cuyo valor de la columna cyl es igual a 8
mtcars[mtcars$hp > 200,]
Nos da los renglones cuyo valor de la columna hp es mayor a 200
mtcars[, c(“mpg”, “cyl”)] Nos da las columnas cuyos nombres son mpg y cyl.

¿Cómo saber el número de renglones de un dataframe?
Con la función **nrow(dataframe)**.

¿Cómo saber el número de columnas de un dataframe?
Con la función **ncol(dataframe)**.

¿Cómo saber el número de renglones y columnas de un dataframe?
Con la función **dim(dataframe)**.
El primer valor que nos da es el número de renglones y el segundo valor es el número de columnas.

¿Para qué sirve la función **head(dataframe)**?
Nos da los primeros 6 registros de un dataframe.

¿Para qué sirve la función **tail(dataframe)**?
Nos da los últimos 6 registros de un dataframe.

¿Cómo convertir los nombres de los renglones de un dataframe en una columna?
Para esto es necesario tener la biblioteca tibble (library(tibble)).
Después se aplica la función:
tibble::rownames_to_column(dataframe, “Nombre de la nueva columna”)

Listas:

¿Qué es una lista?
Es una estructura donde puedes guardar elementos de distintos tipos: números, palabras , vectores, otras listas, dataframes, etc.

¿Cómo se crea una lista?

Para crear una lista haremos uso de la función **list()** y como parámetros pondremos los elementos que deseemos poner en la lista separados por comas.

Ejemplos:

lista_0 ← list(1,2,3,4) Creamos una lista que contiene puros números.

lista_1 ← list(1,"a",c(1,2),list(1,2,3))

Creamos una lista que contiene un número, un carácter, un vector y otra lista

lista_2 ← list(número = 4 , letra = "B")

Creamos una lista con nombres, un elemento llamado número y otro elemento llamado letra.

¿Cómo obtener elementos en una lista (consultas)?

Para obtener elementos de una lista hacemos uso de la notación que ya hemos empleado antes, usamos **"[ ]"** (corchetes).

Ejemplos:

lista_0[2] Nos dará como resultado el segundo elemento de la lista, es decir, 2.

lista_1[4] Nos mostrará en consola los elementos que conforman a la lista que está dentro.

¿Cómo modificar características de la lista?

De un modo similar a los vectores podemos modificar los elementos de una lista como se muestra a continuación:

lista_0[[3]] ← c(2,3,4,5)

De este modo estamos cambiando el tercer elemento de la lista por un vector (2,3,4,5), así nuestra lista tiene los siguientes elementos: (1,2,(2,3,4,5),4)

¿Cómo saber y modificar los nombres de una lista?

Con la función **names(lista)** obtenemos los nombres de una lista.

Si queremos cambiar los nombres de una lista debemos asignarle un vector que contenga los nuevos nombres.

Ejemplo:

lista_1 ← list(1,"a",c(1,2),list(1,2,3)) Creamos una lista.

names(lista_1) ← c("Numero", "Letra", "Vector", "Lista") Modificamos los nombres de la lista

¿Cómo puedo crear una lista vacía?

Para crear una lista vacía es necesario utilizar la siguiente línea de código:

lista_vacia ← **vector(mode = "list", length = 5)**

En este caso creamos una lista vacía con longitud 5.

¿Cómo saber la longitud de una lista?

Con la función **length(lista)**.

Matrices:

¿Qué es una matriz?

Es una estructura que almacena elementos de un sólo tipo de dato, a diferencia de los vectores las matrices tienen una estructura rectangular.

Las matrices las podemos ver como usualmente se utilizan en matemáticas para resolver sistemas de ecuaciones lineales o para hallar valores propios, etc.

¿Cómo se crea una matriz?

Para poder crear una matriz hacemos uso de la función

matrix(vector, nrow = a, byrow = TRUE/FALSE)

A la cual le debemos pasar un vector con los datos que tendrá la matriz, con "nrow = a" le decimos a R que deseamos que nuestra matriz tenga "a" renglones (claramente **"a"** debe ser un número entero) y con "byrow" le decimos a R cómo queremos que se llene la matriz. Si seleccionamos el valor TRUE se llenará la matriz por renglones en el orden en el que están, caso contrario, si seleccionamos FALSE llenará la matriz por columnas.

Ejemplos:

A ← matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3, byrow = FALSE)

La cual tendrá la siguiente forma:

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

B ← matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3, byrow = TRUE)

Notemos la diferencia en el orden de los valores ocasionado por **byrow**.

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

¿Cómo obtener información sobre una matriz?

Existen varias funciones para conocer las características de una matriz:

nrow(matriz) Nos da el número de renglones de la matriz

ncol(matriz) Nos da el número de columnas de la matriz

dim(matriz) Nos da la dimensión de la matriz (primero renglones y después columnas)

¿Cómo obtener elementos en una matriz (consultas)?

Para obtener los elementos de una matriz usamos corchetes [ ].

Ejemplos:

A ← matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3, byrow = FALSE) Creamos una matriz

A[1][1] Nos da como resultado el elemento del primer renglón y la primer columna (1)

A[3][2] Nos da como resultado el elemento del tercer renglón y la segunda columna (6)

A[2][3] Nos da como resultado el elemento del segundo renglón y la tercer columna (8)

¿Cómo modificar características de la matriz?

Para modificar elementos de una matriz haremos uso de la notación utilizada en el apartado de dataframes, usaremos los ejemplos anteriores.

A[2,3] ← 50 Cambiamos el elemento del segundo renglón y la tercera columna por 50

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5   50
[3,]    3    6    9
```

¿Qué operaciones se pueden realizar con matrices?
Supongamos que tenemos dos matrices A y B.
-Suma A**+**B
-Multiplicación A*****B
-Resta A**-**B
En estas operaciones se opera elemento a elemento, entrada a entrada.

-Multiplicación de Matrices A**%*%**B
En esta operación el número de columnas de A debe ser igual al número de renglones de B.

-Transponer una matriz **t(**A**)**

¿Cómo se crea una matriz a partir de vectores?
Con la función **rbind(vectores)**.
Ejemplo:
a ← c(1,2,3)
b ← c(4,5,6)
f ← c(7,8,9)
Creamos 3 vectores
D ← rbind(a, b, f) Creamos una matriz D cuyos renglones son los vectores a, b y f.

III. Funciones

¿Para qué sirve una función?

Una función sirve para automatizar procesos o tareas y ahorrarnos líneas de código dentro del script.

¿Cómo podemos crear una función?

Podemos crear una función siguiendo la estructura siguiente:

```
nombre_funcion ← function(argumento1, argumento2, … ){

Cuerpo de la función/Procedimiento/Operaciones

}
```

Los argumentos se refieren a todos aquellos datos que requiere la función para poder realizar el procedimiento, el cuerpo de la función es propiamente el proceso que deseamos desarrollar y en donde hacemos uso de los argumentos (si los hay).

Es importante destacar que todas las variables que declaremos en el cuerpo de la función solo podrán usarse dentro de la función, es decir, son variables locales que sólo existen durante la ejecución de la función, pero si deseamos que una de las variables forme parte de nuestro ambiente podemos realizarlo mediante el uso de “<<-” al declarar la variable, como se muestra a continuación:

```
nombre_funcion ← function(argumento1, argumento2, … ){

VG <<- 9

}
```

Así, **una vez que apliquemos la función** veremos que la variable VG formará parte de nuestro ambiente global de variables y tendrá asignado el valor de 9.

Ejemplos:

```
fuerza_lunar ← function( ){

print(“Sylveon ha usado Fuerza Lunar”)

}

promedio ← function(a, b, c) {

z ← (a+b+c)/3
paste(“El promedio de los 3 números es” , z)

}
```

```
cantar ← function(cancion = “La Cucaracha”){

paste(“Estoy cantando:” , cancion)

}

producto ← function(a, b = 5){

resultado ← a*b
return(resultado)

}
```

Observemos que algunas funciones tienen por default un valor definido para sus argumentos. Si no se especifica un argumento al llamar la función, entonces la función toma ese valor por default para realizar las operaciones o el procedimiento en el cuerpo de la función.

¿Cómo utilizar una función?
Utilicemos las funciones construidas anteriormente:

```
fuerza_lunar( )
promedio(15, 48, 67)
cantar(“Las Mañanitas”)
producto(a = 8)
```

Familia de Funciones “Apply”

I. **apply(dataframe/matriz, MARGIN = 1/2 , función):** Esta función se utiliza para aplicar funciones sobre los renglones o columnas (dependiendo del valor que le demos a MARGIN, puede ser 1 o 2) de un dataframe o una matriz.

   Ejemplo:
   Retomando una de las matrices que creamos en el apartado anterior:

   ```
   A ← matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3, byrow = FALSE)
   ```

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

   apply(A, MARGIN = 2, sum) con esta línea de código estamos calculando la suma de cada una de las **columnas** de la matriz A.

   apply(A, MARGIN = 1, mean) con esta línea de código estamos calculando el promedio de cada uno de los **renglones** de la matriz A.

II. **lapply(lista, función):** Esta función se utiliza para aplicar una función sobre los elementos de una lista, los resultados los devuelve en forma de lista.
Ejemplos:
lista ← list(c(1,2,3,4), c(10,9,8,7,6)) De primera mano creamos una lista que contiene dos vectores.
lapply(lista, mean) Con esta instrucción estamos calculando el promedio de los vectores que componen la lista y la función nos devuelve los resultados en forma de lista **list(2.5,8)**

III. **sapply(lista, función):** Esta función se utiliza para aplicar una función sobre los elementos de una lista, los resultados los devuelve en forma de vector o matriz.
Ejemplo:
mi_lista ← list(c(1,2,3), c(4,5,6)) Creamos una lista con dos elementos
sapply(mi_lista, sum) Realizamos la operación de suma a los elementos de la lista
Obtenemos como resultado un vector con dos elementos **c(6, 15)**

IV. **tapply(vector, Grupo, función):** Con esta función podemos aplicar funciones a vectores de acuerdo a grupos que cumplan con una misma característica.
Ejemplo:
tapply(mtcars$mpg, mtcars$cyl, mean)
El ejemplo visto en el video correspondiente es muy representativo de lo que hace la función tapply. Con esa línea de código estamos calculando el promedio (**mean**) de las millas por galón (**mpg**) de acuerdo al número de cilindros que tiene el auto (**cyl**), es decir, esto nos dará como resultado el promedio de las millas por galón de los autos con 4, 6 y 8 cilindros.

V. **mapply(function(argumento1, argumento2){…}, argumento1 = , argumento2 =):** Con esta función se puede simplificar la creación de una función y la implementación de la misma. El primer argumento de mapply es crear una función como lo vimos anteriormente, y los argumentos posteriores son para indicar los valores que tomarán los argumentos al aplicar la función.
Ejemplo:
mapply(function(a, b){ a+b }, a = 5, b = 15)
Nos dará como resultado 20.

IV. Condicionales Y Loops

Los condicionales y loops son de utilidad para dos aspectos fundamentalmente; en primer lugar, cuando deseamos realizar un determinado tratamiento de los datos con base al valor de una de nuestras variables (**Condicionales**) y, por otro lado, son de utilidad para poder realizar un mismo proceso varias veces a un determinado grupo de datos o hasta cumplir con una cierta condición (**Loops**).

Condicionales:

if else: Nos permite realizar un proceso con base en si se cumple una determinada condición. Si la condición lógica se cumple, entonces se realiza lo que está dentro del if. Si no se cumple la condición lógica, entonces se realiza lo que esté dentro del else.

Estructura:

```
if(condición lógica){
Procedimiento en caso verdadero
} else {
Procedimiento en caso falso
}
```

Ejemplos:

```
x ← TRUE
if(x == TRUE){
  print("Es verdadero")
} else {
  print("Es falso")
}
```

Lo cual nos mostrará en consola "Es verdadero", pues sí se cumple la condición lógica.

```
manzanas ← 10
if(manzanas  > 15){
   print("Son muchas manzanas")
} else {
   print("Son pocas manzanas")
}
```

Lo cual nos mostrará en consola "Son pocas manzanas", pues no se cumple la condición lógica.

Nota: Es posible anidar tantos condicionales if como sea necesario, es decir, uno puede meter un if dentro de otro if y así sucesivamente, sin embargo, es importante reconocer cuál es el orden de ejecución.

ifelse(): Es una función que simplifica el caso anterior, a la cual nosotros le pasamos como argumentos la condición que deseamos verificar, lo que deseamos que se haga si la condición es verdadera y por último el tratamiento si la condición es falsa.

Estructura:
ifelse(condición lógica, acción de ser verdadera la condición, acción de ser falsa la condición)

Ejemplos:
Tomando los ejemplos del punto anterior con el fin de hacer el símil.

```
x← TRUE
ifelse( x == TRUE, “Es verdadero”, “Es falso”)
```

Lo cual nos mostrará en consola “Es verdadero”, pues al cumplirse la condición se realiza la primera de las acciones que definimos.

```
manzanas ← 10
ifelse( manzanas > 15, “Son muchas manzanas”, “Son pocas manzanas”)
```

Esto nos dará como resultado en consola “Son pocas manzanas”, pues en la variable manzanas guardamos el valor de 10 el cual no es mayor que 15, por lo que se ejecuta la segunda acción que determinamos al no cumplirse la condición.

Switch: Este condicional es de utilidad cuando se tiene una serie de opciones.

Estructura:

```
switch(objeto,
caso1 = acción1,
caso2 = acción2,
        …
)
```

Ejemplos:

```
z ← “Azul”
switch( z,
“Azul” = “Blue”,
“Rosa” = “Pink”,
“Verde” = “Green”,
“Naranja” = “Orange”,
“No está definido ese color”
)
```

Como resultado de este switch obtendremos “Blue”, pues la variable z tiene asignado “Azul”.

```
a ← 1
b ← 2
c ← “Dos”
switch( c,
“Uno” = a+2b+(a^b),
“Dos” = (a*b)-a,
“No existe la operación”
)
```

Como resultado de este switch obtendremos (1*2) - 1 = 1, pues la variable c tiene asignado “Dos”.

Loops:

For: Nos permite repetir un mismo proceso tantas veces como nosotros lo deseemos, la cantidad de ejecuciones del proceso se determinan con base en los índices que nosotros definamos.

Estructura:

```
for ( variable_índice   in   vector_índices){
Proceso
}
```

Ejemplos:

```
for ( i in 1:10){
print(“Springtrap”)
}
```

Este for nos mostrará en consola la palabra “Springtrap” 10 veces, este primer ejemplo es ilustrativo en dos aspectos: que la variable que lleva el control del índice i no tiene por qué formar parte del proceso y, por otro lado, que el tamaño del vector, que en este caso es 10, representa la cantidad de veces que el proceso se va a repetir.

```
for(ind in 1:10){
print( ind )
}
```

Este segundo for nos va a imprimir en consola los números del 1 al 10, este segundo ejemplo destaca una particularidad muy importante del for y es que el valor de ind cambia entre una ejecución y otra. Es decir, en la primera ejecución ind toma el primer valor de nuestro vector de índices, que es 1 y lo imprime, una vez que terminó con el proceso dentro del for lo vuelve a realizar pero ahora ind toma el siguiente valor de nuestro vector de índices, que es 2 y lo imprime, y así sucesivamente hasta que el proceso se ha ejecutado con cada uno de los valores del vector. Razón por la cual nos imprime la enumeración del 1 al 10.

```
for (n in c(“Sylveon”, “Rayquaza”, “Fennekin”, “Gyarados”)){
p ← paste(“Yo te elijo” , n)
print(p)
}
```

Este for nos mostrará en consola “Yo te elijo Sylveon”, “Yo te elijo Rayquaza”, “Yo te elijo Fennekin” y “Yo te elijo Gyarados”, lo cual nos sirve para poder ver con mayor claridad cómo el valor del índice,en este caso **n**, va cambiando en cada ejecución del proceso. Y por otro lado, es un buen ejemplo de que el vector de índices no tiene por qué ser numérico siempre, pues puede contener elementos del tipo texto, numéricos o lógicos.

While: Este loop lo que hace es ejecutar un proceso hasta que se deja de cumplir una condición lógica.

Estructura:

```
while(condición lógica){
Proceso
}
```

Ejemplos:

```
x ← 5
while(x > 0){
print(“Hola”)
x ← x -1
}
```

En este caso se va a imprimir “Hola” en la consola 5 veces.

```
while(TRUE){
print(x)
}
```

Este while nunca va a dejar de ejecutarse, pues siempre se cumplirá la condición lógica. En la consola se imprimirá el valor de x muchas veces y se repetirá infinitamente.

¿Para qué sirve **break**?
Esta palabra lo que hace es interrumpir un loop.

Ejemplo:

```
y ← 50
while(y > 0){
print(y)
if(y == 20){
  break
  }
y ← y -1
}
```

Este while imprimirá los valores del 50 al 20, pero cuando tome el valor de 20 se detendrá, a pesar de que siga siendo un valor mayor a cero. Lo anterior ocurre porque tenemos un if donde se ocupa break.

V. I/O

¿Qué significa CSV?

Comma-separated values o valores separados por comas.

¿Cómo leer archivos CSV?

Existen dos maneras de leer archivos con extensión .csv:

1. Nos fijamos en el apartado de nuestro ambiente, ahí encontraremos una pestaña que dice “Import Dataset”, hacemos click sobre esta opción y nos desplegará un menú, hacemos click en la opción que dice “From Text (readr)...” lo cual nos desplegará una ventana. Hacemos click en el botón que dice “Browse…” y abrimos el archivo que deseamos cargar a R. Finalmente, hacemos click en “Import”.

2. Para cargar un archivo con extensión .csv mediante líneas de código es necesario que primero instalemos y carguemos la biblioteca “**readr**”, pues en esa biblioteca se encuentra la función que nosotros usaremos. La función read_csv() es la que nos va a permitir cargar los datos dentro del csv, dentro de la función debemos escribir la ruta dentro de nuestra computadora del archivo que nosotros queramos cargar, escrito entre comillas. Para hacer más fácil la búsqueda podemos colocar lo siguiente:
   read_csv (“./”) al escribir “./” y presionar la tecla tabulador, R nos mostrará un menú a partir del cual podremos apoyarnos para buscar nuestro archivo.
   Al correr la línea de código ya estará cargada la base de datos, esta manera es la más recomendable pues al estar dentro del código basta con correr las líneas para que la base de datos vuelva a estar cargada.

Ejemplo:

library(readr) Cargamos la biblioteca readr

frutas ← **read_csv(**“Documentos/Datos/frutas.csv”**)**
Seleccionamos el archivo CSV y lo guardamos en la variable frutas

View(frutas) Visualizamos los datos en R

¿Cómo escribir archivos CSV?

Se puede escribir un archivo de extensión .csv con la función siguiente:

write.csv(x = Nombre_Dataframe, file = “Ruta y nombre del archivo”)

Ejemplo:

write.csv(x = mtcars, file = “./Documentos/Carros/tabla_carros.csv”)

Este código nos creará un archivo CSV que se llame tabla_carros y cuya información es la de mtcars.

¿Cómo leer archivos de Excel?

De igual manera existen dos formas de cargar archivos de excel a R:

1. Nos fijamos en el apartado que denominamos como ambiente y hacemos click en la pestaña “Import Dataset”, lo cual desplegará una serie de opciones, dentro de ellas se encuentra “From Excel…”, hacemos click sobre ella. Esto abrirá una ventana que nos permitirá buscar el archivo que deseamos cargar, una vez seleccionado hacemos click sobre el botón que dice “Import”. Siguiendo estos pasos los datos estarán cargados en R.

2. Para lograr el mismo resultado mediante líneas de código es necesario instalar y cargar la biblioteca “readxl”. Una vez hecho esto podemos hacer uso de la función read_excel (), a la cual le pasaremos la ruta dentro de nuestra computadora del archivo que deseamos cargar, escrito entre comillas. Para ello podemos apoyarnos de lo siguiente: **read_excel(“./”)**, al escribir “./” y presionar la tecla tabulador nos mostrará una serie de opciones a partir de las cuales podremos hacer la búsqueda de nuestro archivo. Una vez que corramos la línea, la base de datos ya estará cargada. Es el método más recomendable pues así nos evitamos estar cargando la base de datos cada vez que se requiera usar el código.

Ejemplo:
library(readxl) Cargamos la biblioteca readxl

frutas ← **read_excel(**“Documentos/Datos/frutas.xlsx”**)**
Seleccionamos el archivo de Excel y lo guardamos en la variable frutas

View(frutas) Visualizamos los datos en R

¿Cómo escribir archivos de Excel?

Para realizar esta acción, es necesario instalar y/o cargar primero la biblioteca de openxlsx.
Posteriormente, usamos la función siguiente:

write.xlsx(x = Nombre_Dataframe, file = “Ruta y nombre del archivo”)

Ejemplo:
library(openxlsx)
write.xlsx(x = mtcars, file = “./Documentos/Carros/tabla_carros.xlsx”)

Este código nos creará un archivo de Excel que se llame tabla_carros y cuya información es la de mtcars.

VI. Manejo de NA´s

¿Qué es un NA?

Es la forma en que R identifica un valor faltante, un valor que no está en una base de datos. Un ejemplo de ello es cuando se tienen los resultados de cierta encuesta o formulario y algunas personas dejan espacios en blanco, es decir, no llenan ese campo.

¿Cómo saber si un vector contiene NA´s?

Esto lo podemos hacer con la función **is.na(vector)**.
La función nos devolverá un vector de tipo lógico con elementos TRUE y FALSE, cada TRUE significa que en esa posición del vector se encuentra un NA y cada FALSE indica que en esa posición del vector no hay NA.

Ejemplo:
vector_1 ← c(“Calabaza”, “Elote”, NA, NA, “Aguacate”)
is.na(vector_1)

Esto nos devolverá el vector siguiente: c(FALSE, FALSE, TRUE, TRUE, FALSE)

¿Cómo contar el número de NA´s que tiene un estructura de datos?

Existen varias formas de hacerlo:

- Usando las funciones sum( ) y is.na( )
  Ejemplo:
  Consideremos a vector_1 de la pregunta anterior.
  sum(is.na(vector_1**))**
  Lo anterior nos da como resultado 2, pues cada TRUE lo cuenta como 1 y cada FALSE como 0. Esto concuerda con los 2 NA que tiene el vector.
- Usando las funciones nrow( ) y is.na( )
  Ejemplo:
  Consideremos un dataframe genérico llamado datos, que contiene una columna llamada edad.
  nrow(datos**[is.na(**datos**$**edad**),])**
  Lo anterior nos dará como resultado el número de NA’s que se encuentren en la columna edad del dataframe datos.
  El código lo que realiza es extraer la columna edad del dataframe. datos$edad
  Después selecciona los renglones del dataframe que cuentan con NA en la columna edad. datos[is.na(datos$edad),]
  Posteriormente cuenta el número de renglones filtrados, los cuales contienen NA en la columna edad. nrow(datos[is.na(datos$edad),])
- Usando la función apply( )
  Ejemplo:
  Consideremos un dataframe genérico llamado datos.
  apply(datos**, MARGIN = 2, function(x) sum(is.na(x)))**
  Lo anterior nos dará como resultado el número de NA’s que se encuentren en todas las columnas del dataframe datos.
  Esta última forma permite simplificar los cálculos, pues los métodos anteriores sólo se aplican a un vector o a una sóla columna.

¿Cómo tratar este tipo de valores?

Una forma de eliminar los NA's de nuestra base de datos es sustituir estos valores por el promedio de los datos que sí tenemos.

Ejemplo:

vc ← c(1, 2, NA, 4, NA, NA, 6, 7, 8, NA)

A nuestro vector le aplicamos primeramente la función is.na( ), para poder detectar la posición de los NA dentro del vector.

is.na(vc) = c(FALSE, FALSE, TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)

Una vez hecho esto, procedemos a calcular la media de los datos que sí tenemos dentro del vector y la usamos para sustituir los NA's.

vc[is.na(vc**)]** ← **mean(**vc**, na.rm = TRUE)**

Ya que el vector que nos da is.na(vc) tiene el valor TRUE en la misma posición en donde encontramos un NA, lo que le estamos diciendo a R es que los elementos NA dentro del vector vc los sustituya por la media de los datos que sí conocemos.

El argumento na.rm = TRUE dentro de la función mean() sirve para que ignore los datos NA y calcule la media con base en los datos restantes.

Para realizar este proceso a una de las columnas de un dataframe o una matriz se sigue el mismo razonamiento:

dataframe[is.na(dataframe$columna), "columna"] ← mean(dataframe$columna, na.rm = TRUE)

Donde "dataframe" es el nombre de la variable en donde tenemos guardada nuestra base de datos y "columna" es el nombre de la columna a la cual deseamos removerle los NA's.

Lo mismo se puede hacer con la mediana de los datos:

dataframe[is.na(dataframe$columna),"columna"] ← median(dataframe$columna, na.rm = TRUE)

VII. DPLYR Y MAGRITTR

¿Para qué sirve el operador pipe %>%?

El operador pipe sirve para poder programar de una forma ordenada, para ir definiendo la secuencia del código de una manera más entendible y menos rebuscada.
Para poder usarlo es importante cargar la biblioteca **magrittr**.
De forma general se puede ver así:

```
datos %>%
  funcion_1(argumentos) %>%
  funcion_2(argumentos) %>%
  funcion_3(argumentos)
```

Ejemplo:

```
library(magrittr)
edades ← c(30, 16, 73, 27, 48)      Creamos un vector con edades
edades %>% sum( )
```

Lo anterior nos dará como resultado 194 y es lo mismo que haberlo calculado como:

```
sum(edades)
```

Haremos uso del operador pipe para explicar las funciones dentro de la biblioteca dplyr, con el fin de que los códigos sean más fáciles de comprender.

Funciones de la biblioteca dplyr:

No olvidemos que antes de usar cualquiera de las funciones descritas a continuación, es necesario cargar la biblioteca dplyr con library(dplyr).

Al igual que en el video correspondiente, usaremos de ejemplo la base de datos llamada “diamonds”, la cual podemos usar al cargar la biblioteca de ggplot2 y mediante la siguiente línea de código:

```
data(“diamonds”, package = “ggplot2”)
```

1. **select( )** : Esta función nos permite obtener las columnas que nosotros deseemos de un dataframe, para ello basta con pasarle como argumentos los nombres o posiciones de las columnas deseadas.

Ejemplos:

diamonds %>% select(carat, cut, price): Esta línea de código nos dará como resultado un nuevo dataframe donde tendremos las columnas “carat”, “cut” y “price” de la base de datos original, que en este caso es “diamonds”.

diamonds %>% select(1, 4): Este ejemplo es meramente ilustrativo respecto a que no necesariamente se requiere conocer el nombre de la columna, puedes realizar el mismo proceso solamente con la posición de la columna.

diamonds %>% select(-carat, -price): Al agregarle un signo de menos (-) a los parámetros vamos a generar un dataframe en donde no aparezcan las columnas “carat” y “price”, es decir, las demás columnas de “diamonds” pasarán al nuevo dataframe.

diamonds %>% select(price, everything( )): Mediante esta línea de código vemos que select( ) no sólo sirve para extraer columnas de una base de datos, sino que también sirve para cambiar el orden de las columnas. Por medio de esta línea de código estamos colocando la columna “price” como la primera en el nuevo dataframe y las demás simplemente se recorren.

diamonds %>% select(starts_with(“c”)): Se tienen muchas opciones para seleccionar las columnas que requerimos, ejemplo de ello es este caso. Utilizando esta línea de código lo que estamos haciendo es extraer aquellas columnas cuyo nombre inicie con la letra “c”, pero es necesario conocer que no sólo puede ser una letra, también podemos buscar las columnas cuyo nombre empiece por “co”, “Pa”, etc.

diamonds %>% select(ends_with(“o”)): Caso contrario al anterior, con esta línea estamos buscando extraer las columnas cuyo nombre termine en “o”. De igual forma, es necesario mencionar que no tiene que ser sólo una letra, pueden ser cadenas de texto más largas.

diamonds %>% select(contains(“or”)): Usando esta línea de código vamos a extraer aquellas columnas cuyos nombres contengan “or”. Es importante destacar que si tuviéramos columnas de nombre “floor” y “rope”, la función únicamente tomaría la columna de nombre “floor”, ya que a pesar de que “rope” tiene las letras “o” y “r”, estas no están en el orden que nosotros definimos.

diamonds %>% select(is.numeric()): Finalmente en el caso de tener un tibble, como es el caso de la base de datos “diamonds”, también podemos seleccionar las columnas con base en el tipo de datos que contiene cada columna. En este caso, estamos fijándonos en las columnas que tienen datos de tipo numérico.

2. **filter( )** : Esta función, como su nombre indica, nos permite filtrar aquellos renglones de un dataframe que cumplen con cierta condición.

Ejemplos:

diamonds %>% filter(price < 700): Esto nos dará como resultado todos aquellos renglones del conjunto de datos diamonds cuya columna de precio tenga un valor menor a 700.

diamonds %>% filter(table == 58 & depth > 60): Esto nos dará como resultado aquellos renglones que cumplan ambas condiciones, que table sea 58 y depth sea mayor a 60.

diamonds %>% filter(carat == 0.23 | carat == 0.33): Esto nos dará como resultado aquellos renglones cuya columna de carat tenga el valor de 0.23 o de 0.33.

Una función muy útil cuando trabajamos con strings es **grepl(“patrón”, vector)**, esta función lo que hace es buscar una serie de caracteres en un vector.

Ejemplo:
nombres ← c(“Mario”, “Luis”, “Mariana”, “Pedro”, “Juan”)
grepl(“Mar”, nombres)

Esto nos dará como resultado el vector c(TRUE, FALSE, TRUE, FALSE, FALSE), pues el patrón “Mar” se encuentra en la primera y la tercera entrada del vector nombres.

Si combinamos esta función con la función filter( ), podremos filtrar de otra forma los datos. Por ejemplo:
diamonds %>% filter(grepl(“S”, clarity)): Esto nos dará como resultado aquellos renglones cuya columna clarity contenga “S”.

3. **rename( )** : Esta función nos permite cambiar el nombre de las columnas de la base de datos. Algo importante a mencionar aquí es que la función crea una nueva base de datos con los nombres nuevos, es decir, los nombres no se modifican en la base de datos original. Hay que tener mucho cuidado con la sintaxis al usar la función, para usarla correctamente debemos seguir la siguiente estructura:

rename(nombre_nuevo = nombre_original)
Si invertimos el orden de los nombres, R no encontrará la columna a la que le quieres cambiar el nombre.

Ejemplos:
diamonds %>% rename(precio = price): Usando esta línea estamos generando una nueva base de datos en donde a la columna “price” la nombramos “precio”.

4. **transform( )** : Esta función se utiliza para modificar una columna de un conjunto de datos.

Ejemplos:

diamonds %>% transform(carat = ifelse(carat > 28, “Diamante Pesado”, “Diamante Ligero”)): Esto sustituirá los valores de la columna carat. Si el valor era mayor a 28, se le asigna “Diamante Pesado”. Si esto no ocurre, entonces se asigna “Diamante Ligero”.

Otra forma de hacerlo es la siguiente:

```
diamonds %>% transform(carat = case_when(
    carat > 28 ~ “Diamante Pesado”,
    TRUE ~ “Diamante Ligero”
))
```

Si quisiéramos, por ejemplo, convertir la columna table en carácter, lo podemos hacer como:
diamonds ← diamonds %>% transform(table = as.character(table))

La forma de comprobar el cambio sería usando la función **glimpse(**diamonds**)**.

5. **mutate( )** : Por medio de esta función podemos generar una nueva base de datos con una columna adicional. Tiene un funcionamiento bastante similar con **tranform ( )**.

Ejemplos:
diamonds %>% mutate(nueva_columna = 1): Al hacer uso de esta línea de código le estamos agregando a la base de datos “diamonds” una nueva columna de nombre “nueva_columna” la cual está conformada por puros unos.

```
diamonds %>% mutate(Criterio_Precio = case_when(
price > 500 & price < 1000 ~ "Bueno",
price <= 500 ~ "Barato",
TRUE ~ "Muy Caro"
))
```

Por medio de este código nosotros podemos generar una columna adicional en la base de datos “diamonds” en donde a cada uno de los diamantes se le asignó, dependiendo de su precio, si los consideramos baratos, con un precio bueno o que están muy caros. Este ejemplo nos muestra lo similares que son las funciones mutate y transform, con la diferencia de que mutate agrega una columna y transform altera los datos de una de las columnas.

diamonds %>% mutate(clarity_number = parse_number(clarity)): Esta línea de código tiene un uso muy particular, podemos notar que los datos de la columna “clarity” de “diamonds” están conformados por una serie de letras y al final tienen un número, ¿cómo podríamos extraer el número que está al final y colocarlo en una nueva columna?, pues esencialmente eso es lo que hace esta línea de código.

6. **separate( )** : Esta función se utiliza para separar en columnas los datos que se encuentran en un dataframe.

Ejemplo:
datos ← data.frame(x = c(“a,A”, “b,B”, “c,C”))
datos %>% separate(x , c(“Minúscula”, “Mayúscula”), “,”)

Lo anterior nos dará como resultado la siguiente tabla:

Minúscula	Mayúscula
a	A
b	B
c	C

Por lo tanto, la función dividió los datos en dos columnas, separándolos por la coma que tenían en medio.

Función group_by( ):
Esta función nos permite hacer diferentes cálculos sobre nuestros datos, clasificándolos de acuerdo a un determinado criterio. En otras palabras, nos permite agrupar nuestros datos de acuerdo a una característica y hacer cálculos sobre esos grupos.

Para los ejemplos haremos uso de la base de datos “diamonds” ya que es la que se ha trabajado anteriormente y por lo tanto permite que los ejemplos sean mucho más comprensibles.

Ejemplos:

```
diamonds%>%
    group_by(cut)%>%
    summarise(conteo = n())
```

Para comprender cómo funciona group_by (), explicaremos en esta primera ocasión paso a paso lo que hace R por medio de esta línea de código. Al aplicar la función **group_by(cut)** estamos clasificando nuestros datos de acuerdo al corte que tiene el diamante, es decir, estamos generando subgrupos de acuerdo al corte. Una vez creados estos subgrupos, les aplicamos la función **summarise(conteo = n())**, con lo cual simplemente estamos contando cuántos datos hay en cada subgrupo, cuántos de los diamantes hay con los distintos cortes.

Otra forma de hacer esto es con ayuda de la función tally( ):

```
diamonds%>%
    group_by(cut)%>%
    tally( )
```

Si quisiéramos ordenar los resultados de menor a mayor, tendríamos que añadir la función arrange( ):

```
diamonds%>%
    group_by(cut)%>%
    tally( )%>%
    arrange(n)
```

Si quisiéramos ordenar los resultados de mayor a menor, tendríamos que añadir la función arrange( ):

```
diamonds%>%
    group_by(cut)%>%
    tally( )%>%
    arrange(desc(n))
```

Usando la siguiente línea de código podemos calcular el precio promedio de los diamantes de acuerdo al corte que estos presentan. Es decir, nos da el precio promedio de un diamante con corte Premium, Good, etc.

```
diamonds%>%
    group_by(cut)%>%
    summarise(promedio = mean(price))
```

Si quisiéramos obtener los porcentajes de acuerdo a una columna con ciertas categorías, tendríamos que hacerlo de la siguiente manera:

```
diamonds%>%
    group_by(columna)%>%
    summarise(n = n( )/nrow(.))
```

El punto nos ayuda a hacer referencia al dataset con el que estamos trabajando, en este caso hace referencia al número de renglones del dataset diamonds.

```
diamonds%>%
    group_by(cut, clarity)%>%
    tally()
```

Este ejemplo es meramente ilustrativo, ya que con él podemos ver que no estamos restringidos a clasificar nuestros datos sólo a una de las columnas, sino que podemos hacerlo con más. Por medio de esta línea de código estamos contando la cantidad de diamantes que hay por cada una de las diferentes combinaciones de corte de diamante y el nivel de claridad que presenta.

```
diamonds %>%
    group_by(cut)%>%
    summarise(min_price = min(price), max_price= max(price))
```

Este ejemplo, al igual que el anterior, es para mostrar que R no restringe a una sola función, en este caso estamos calculando dos diferentes indicadores (máximo y mínimo), tomando en cuenta la misma clasificación por corte.

Si quisiéramos obtener el número de datos que cumplen una condición y el número de aquellos que no la cumplen, tendríamos que usar:

```
diamonds%>%
    group_by(condición lógica)%>%
    tally( )
```

VIII. GGPLOT2

¿Para qué sirve la biblioteca ggplot2?

Nos sirve para realizar gráficas.

¿Cómo usar la función qplot?

Primero debemos cargar la librería **library(**ggplot2**)**.
Posteriormente se debe seguir la estructura siguiente:

qplot(x = , y = , data = ,	geom = **“point”**, colour = , shape = **)**	Gráfico de Dispersión
	geom = **“boxplot”,** fill = **)**	Diagrama de Caja
	geom = **“violin”,** fill = **)**	Diagrama de Violín
	geom = **“dotplot”**, stackdir = , binaxis = , dotsize = **)**	Diagrama de Puntos
	geom = **“histogram”**, fill = **)**	Histograma
	geom = **“density”**, color = , linetype = **)**	Función de Densidad

Notas:

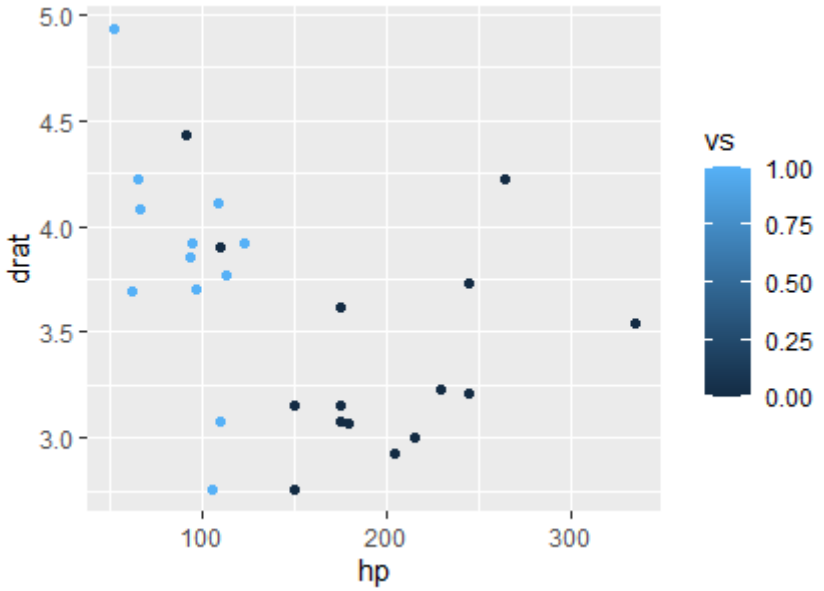
- Los parámetros colour, shape, fill, color y linetype sirven para indicar la variable que se usará para identificar los datos con distintos colores, tonalidades o formas.
- No todas las gráficas requieren el parámetro para y, pues algunas sólo utilizan los valores en x, como lo son “density” e “histogram”.
- El parámetro stackdir sirve para ubicar la dirección de los puntos, este puede tomar los valores “center”, “up” y “down”.
- El parámetro binaxis es para ubicar el eje de referencia, ya sea “x” o “y”.
- El parámetro dotsize es para ajustar el tamaño de los puntos, este valor es un número.

Ejemplos:
Tomemos como base de datos a mtcars.

1)

```
qplot(x = hp, y = drat, data = mtcars, geom = "point", colour = vs)
```

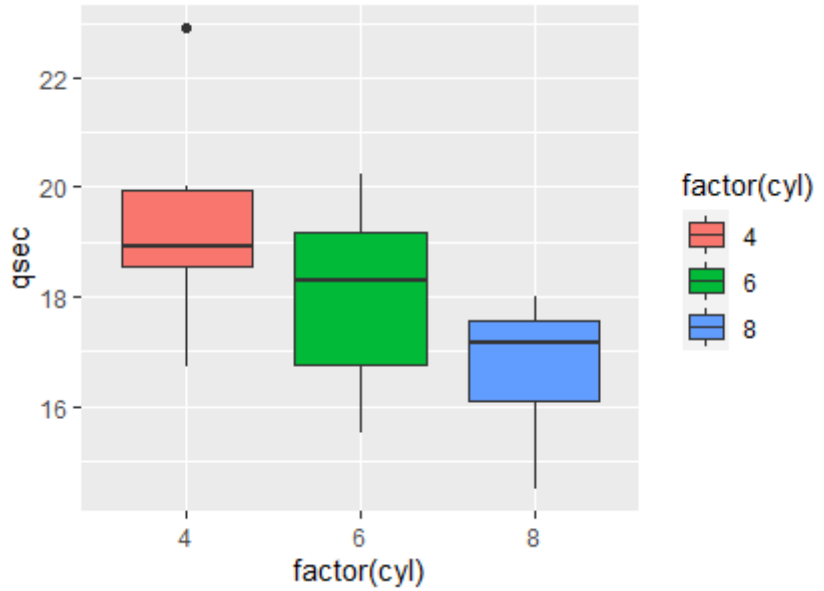
Con esta línea de código estamos generando un gráfico de dispersión, donde el eje x contiene los valores de hp y el eje y los valores de drat. Además estamos identificando los puntos con distintas tonalidades de azul para indicar el valor de vs que tienen.



2)

qplot(x = factor(cyl), y = qsec, data = mtcars, geom = "boxplot",fill = factor(cyl))

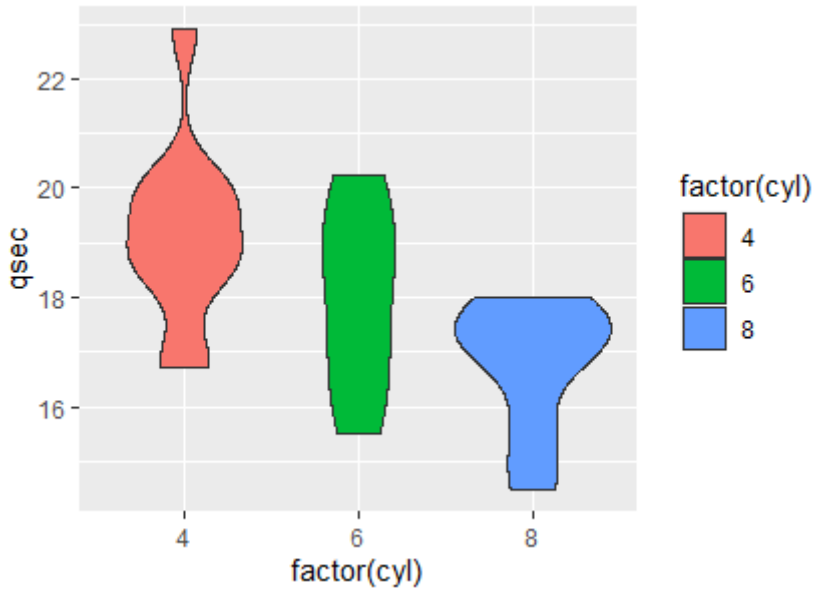
Con esta línea de código estamos generando un diagrama de caja, donde el eje x contiene los valores de cyl (convertidos en factor) y el eje y los valores de qsec. Además estamos identificando cada caja con un color de acuerdo con el valor de cyl que tiene.



3)

qplot(x = factor(cyl), y = qsec, data = mtcars, geom = "violin",fill = factor(cyl))

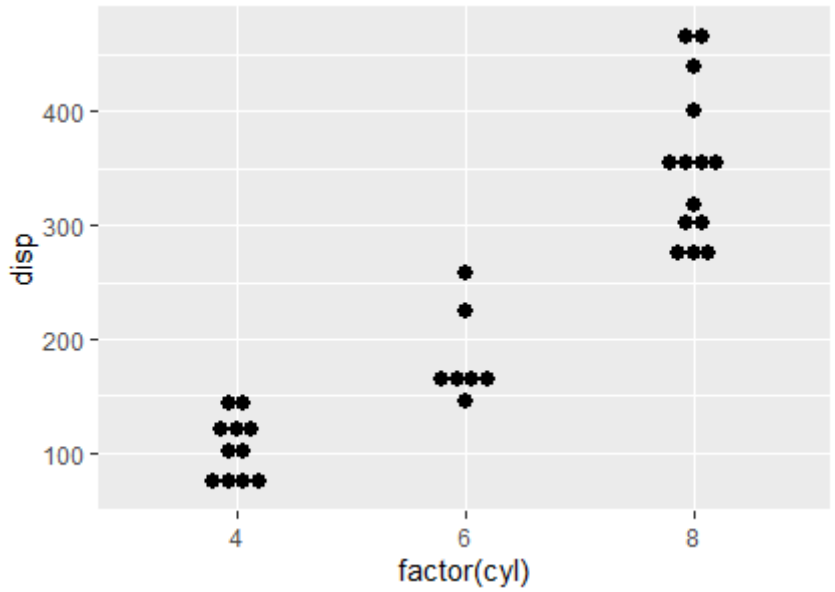
Ahora estamos haciendo lo mismo que en la gráfica anterior, pero ahora generamos un diagrama de violín, el cual muestra la forma que toma la distribución de los datos.



4)

qplot(x = factor(cyl), y = disp, data = mtcars, geom = "dotplot", stackdir = "center", binaxis = "y", dotsize = 1)

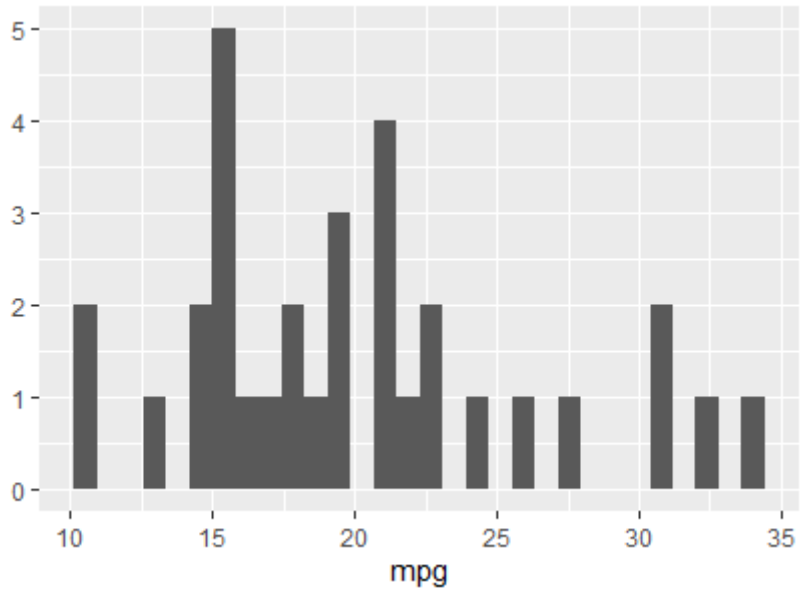
Con esta línea de código estamos generando un diagrama de puntos, donde el eje x contiene los valores de cyl (convertidos en factor) y el eje y los valores de disp. Definimos el tamaño de los puntos en 1 y su orientación centrada con respecto al eje y.



5)

qplot(x = mpg ,data = mtcars, geom = "histogram")

Con esta línea de código estamos generando un histograma, donde el eje x contiene los valores de mpg y el eje y muestra la frecuencia de los datos.

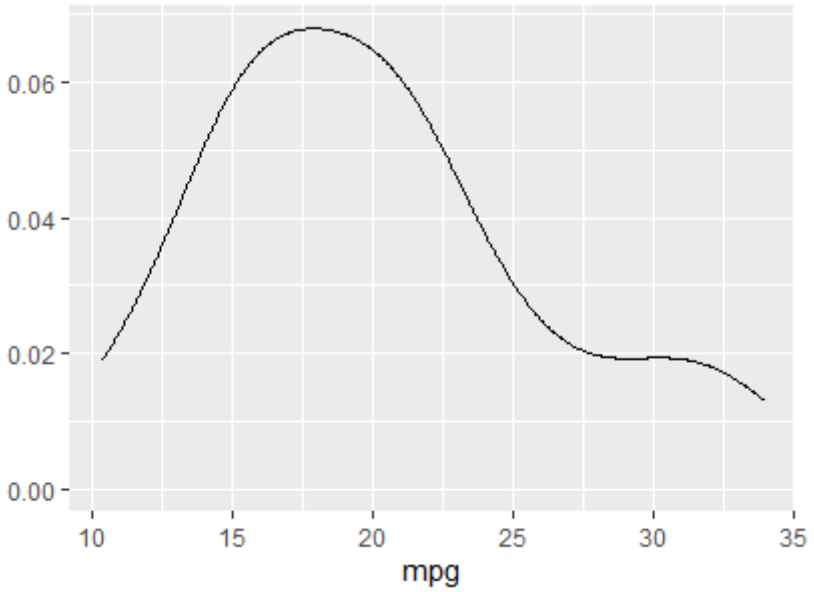


.

6)

qplot(x = mpg ,data = mtcars, geom = "density")

Con esta línea de código estamos graficando la función de densidad de mpg, podemos ver que toma la forma del histograma anterior pero ahora traza una curva continua y suave donde se encontraban las puntas de las barras.



La función qplot( ) es amigable para realizar gráficos sencillos, sin embargo, a continuación veremos otra función con la que se tiene una mayor flexibilidad para personalizar las gráficas.

¿Cómo usar la función ggplot?

Primero debemos cargar la librería **library(**ggplot2**)**.
Posteriormente se debe seguir la estructura siguiente:

ggplot(data = , aes(x = , y =)**) +**	**geom_point(**size = , shape = **)**	Gráfico de Dispersión
	geom_density( )	Función de Densidad
	stat_density( )	Función de Densidad (Sombreada)
	geom_dotplot( )	Gráfica de Puntos
	geom_histogram( )	Histograma
	stat_ecdf( )	Función de Distribución Empírica

Notas:

- Los parámetros size y shape son números
- No todas las gráficas requieren el parámetro para y, pues algunas sólo utilizan los valores en x, como lo son geom_density( ), stat_density( ), geom_dotplot( ), geom_histogram( ) y stat_ecdf( ).

Adicional a esto podemos añadirle nombre a la gráfica y etiquetas a los ejes:

+ **ggtitle(**"Título de la gráfica"**)**
+ **xlab(**"Etiqueta eje x"**)**
+ **ylab(**"Etiqueta eje y"**)**

Así como cambiar el color o tema del gráfico, algunos de ellos son:

+ **theme_light( )**
+ **theme_dark( )**
+ **theme_classic( )**
+ **theme_gray( )**

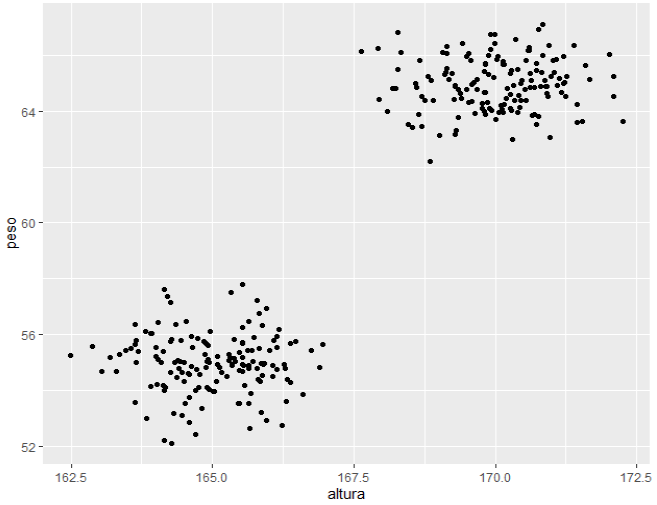
Si queremos fijar un tema para todas las gráficas, se debe usar la función: **theme_set( )**.
Ejemplos:
theme_set(theme_dark( ))
theme_set(theme_gray( ))

Tomemos como ejemplo un dataframe que incluye los datos de peso, altura y género de 300 personas, el cual simulamos por medio del siguiente código:

```
datos <- data.frame(genero = factor(rep(c("M", "F"), each = 150)),
                    altura = c(rnorm(150, 170), rnorm(150, 165)),
                    peso   = c(rnorm(150, 65), rnorm(150, 55)))
```

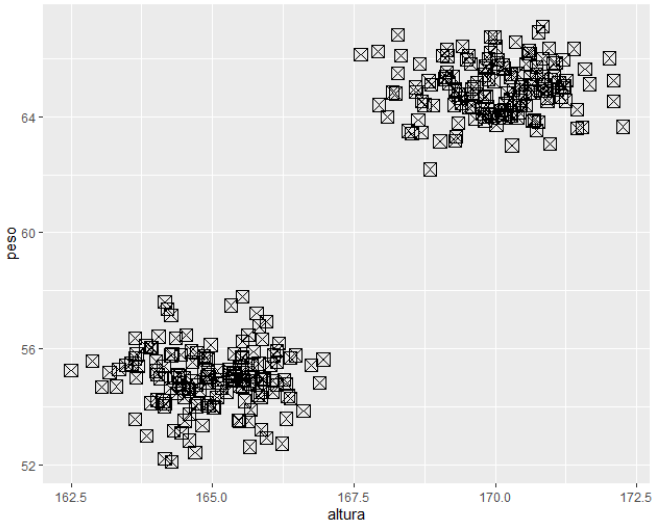
Con este dataframe en consideración, mostraremos algunas de las gráficas que podemos realizar utilizando la biblioteca ggplot2:

ggplot(data = datos, aes(x=altura, y= peso)) + geom_point( )



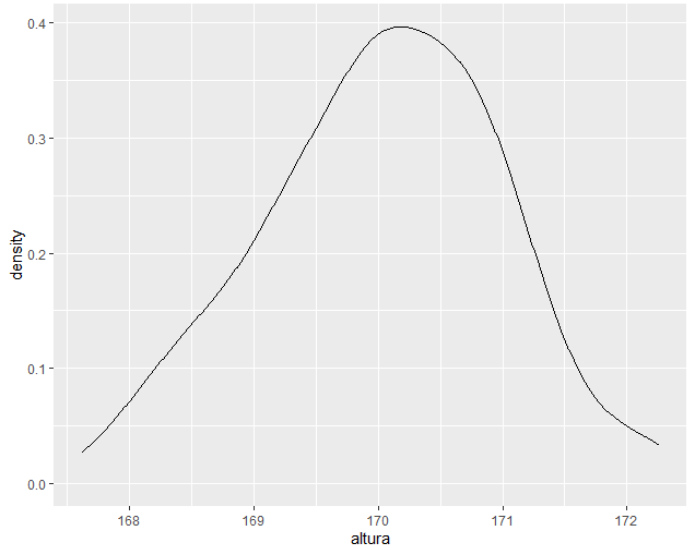
Como podemos observar estamos graficando el peso y la altura de cada una de las personas, los cuales se marcan por medio de un punto. En este caso se observan dos acumulaciones de puntos correspondientes a la diferencia en peso y altura de hombres y mujeres.

ggplot(data = datos, aes(x=altura, y= peso)) + geom_point(size = 4, shape= 7)



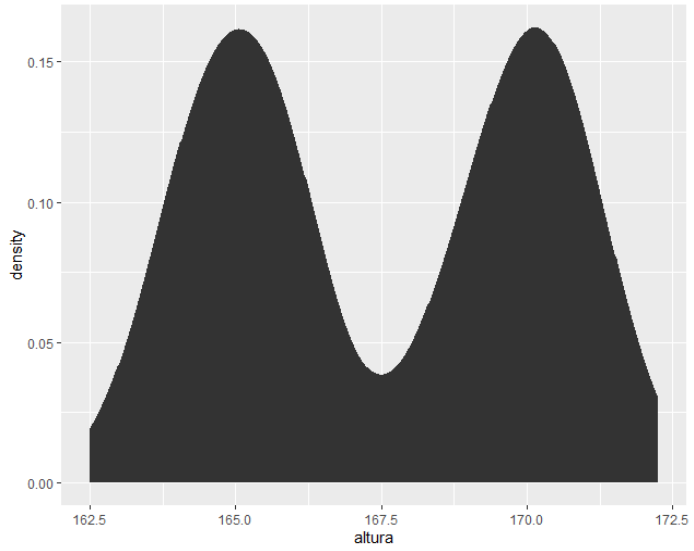
En esta gráfica estamos graficando los mismos datos, pero notemos que hay una diferencia en cuanto a la forma de los puntos y al tamaño de estos, los cuales podemos cambiar con ayuda de los parámetros **“size”** y **“shape”** dentro de la función geom_point( ).

ggplot(data = datos %>% filter(genero == "M"),aes(x=altura)) + geom_density( )



Por medio de esta gráfica podemos observar la función de densidad que está asociada con la altura, enfocándonos en el género Masculino (M).

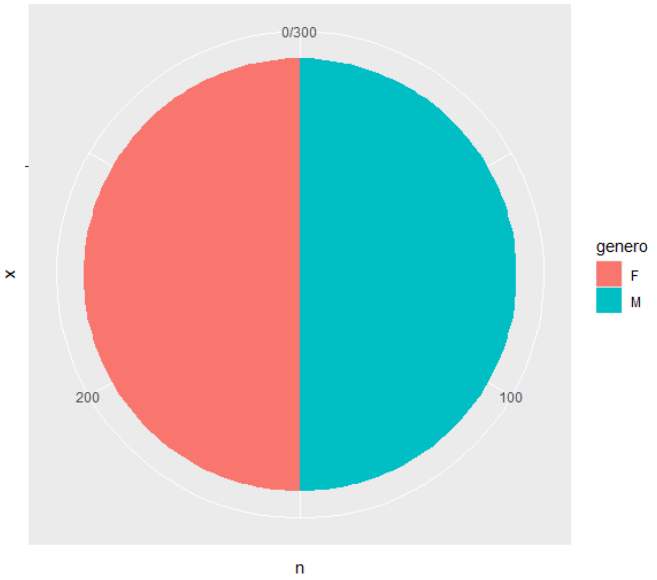
ggplot(data = datos, aes(x = altura)) + stat_density( )



Esta línea de código nos permite graficar la función de densidad asociada a los datos, pero con la diferencia de que el área bajo la curva se verá sombreada. En este caso vemos dos picos debido a que uno corresponde a los datos de las mujeres y el otro a los hombres.

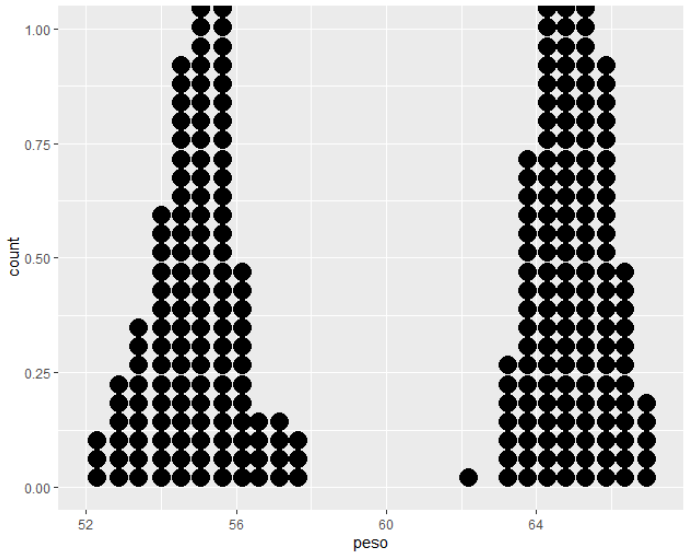
```
pay <- datos %>%
  group_by(genero)%>%
  summarise(n = n())

ggplot(pay, aes(x = "", y = n, fill = genero)) +
  geom_bar(stat = "identity") +
  coord_polar("y", start = 0)
```



Las anteriores líneas de código nos permiten realizar una gráfica de pastel dividiendo nuestros datos con base en una de las columnas, en este ejemplo la división se realizó por género. Debido a que al simular los datos se tomaron en cuenta 150 mujeres y 150 hombres, vemos representado en la gráfica la mitad de hombres y la mitad de mujeres.

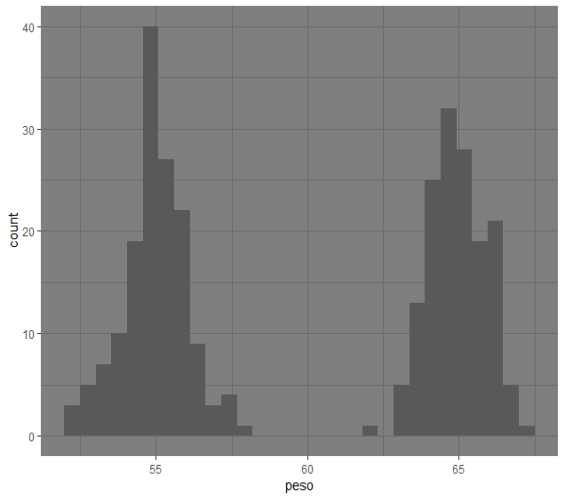
ggplot(datos, aes(x= peso)) + geom_dotplot( )



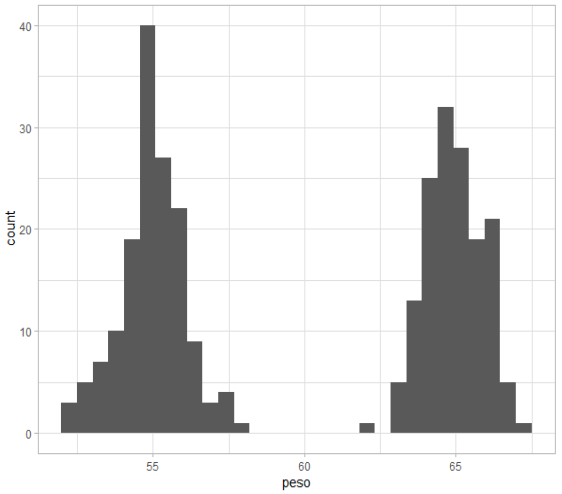
La gráfica generada con geom_dotplot( ) es muy similar a la generada con geom_density( ), con la diferencia de que esta gráfica está conformada por puntos y no por una sola línea. Por lo tanto, nos representa por medio de puntos la función de densidad asociada con los datos.

1 → ggplot(datos, aes(x= peso)) + geom_histogram() + theme_dark()

2 → ggplot(datos, aes(x= peso)) + geom_histogram() + theme_light()



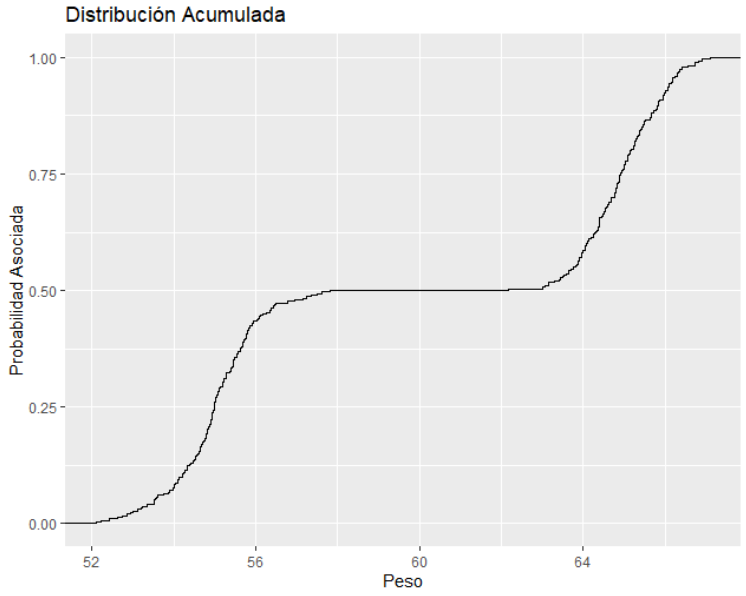
1



2

Utilizando geom_histogram( ) podemos graficar el histograma de los datos. Adicionalmente podemos observar el efecto de las funciones **theme_dark( )** y **theme_light( )**

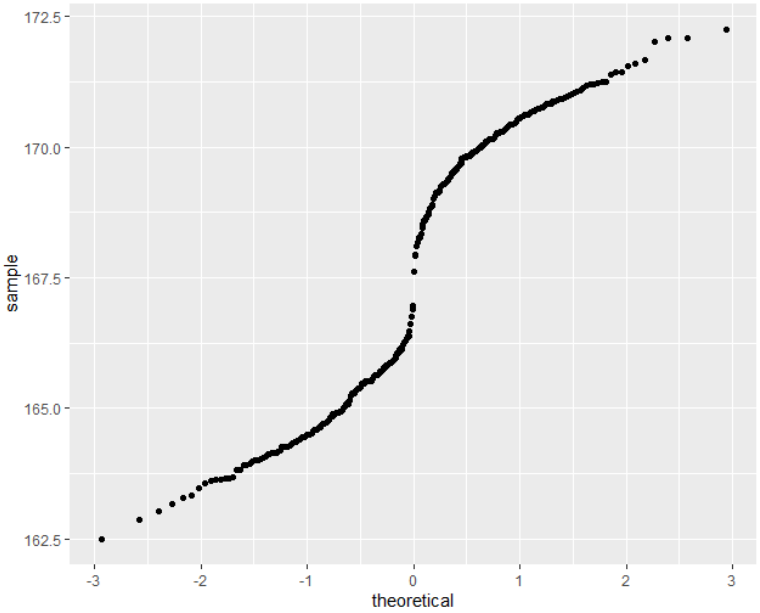
ggplot(datos, aes(x= peso)) + stat_ecdf() + ggtitle("Distribución Acumulada") + xlab("Peso") + ylab("Probabilidad Asociada")



Ésta función nos permite graficar la función de probabilidad acumulada de nuestros datos. Además, podemos observar cómo ponerle nombre al gráfico y a los ejes.

ggplot(data = datos, aes(sample = altura)) + stat_qq( )

qqplot: Esta gráfica nos permite comparar los percentiles de nuestros datos con los percentiles de una distribución normal, es decir, nos permite observar si nuestros datos presentan un cierto comportamiento cercano a la distribución normal. Si el gráfico se asemeja a la función identidad, entonces es muy probable que los datos provengan de una distribución normal.



En este caso no podemos afirmar que los datos provienen de una distribución normal, pues no se obtuvo una recta en el qqplot.

IX. LUBRIDATE

¿Para qué sirve la biblioteca lubridate?
Nos sirve para manipular datos que sean del tipo fecha.

¿Qué funciones podemos encontrar en esta biblioteca?
A continuación se muestra un listado de las funciones más importantes de esta biblioteca, así como la descripción de qué es lo que realiza cada una:

as_datetime( ) : Esta función nos indica la fecha resultante después de sumarle los segundos que pasamos como parámetro a la fecha 01/01/1970.
Ejemplo:
as_datetime(5000)
Esto nos dará como resultado la fecha de "1970-01-01 01:23:20 UTC" , que es la fecha resultante de sumarle 5000 segundos al día 01/01/1970.

as_date( ) : Esta función la podemos utilizar de dos maneras.
Primeramente, dándole como parámetro un número.
Ejemplo:
as_date(25)
Esto nos dará como resultado la fecha correspondiente a 25 días después del 1 de enero de 1970, es decir, 1970-01-26. El 1 de enero de 1970 es la fecha referencia, por lo que, el número que nosotros le pasemos a la función será la cantidad de días que le sume a esa fecha.

Por otro lado, esta fecha también puede ser utilizada para convertir datos del tipo string o cadena de texto en datos de tipo fecha, siempre y cuando la fecha esté colocada en el orden: “Año-Mes-Día”.
Ejemplo:
fs ← "2001-03-25"
f ← as_date(fecha)
Con estas líneas hemos guardado en la variable f la fecha que teníamos guardada en f1 (como string), pero con la diferencia de que en f ya es un dato del tipo fecha.

year( ) : Esta función nos devuelve el año de un dato tipo fecha.
Ejemplo:
fecha ← as_date("2021-12-10")
year(fecha)
Esto nos dará como resultado 2021.

month( ) : Esta función nos devuelve el mes de un dato tipo fecha.
Ejemplo:
fecha ← as_date("2021-12-10")
month(fecha)
Esto nos dará como resultado 12.

day( ) : Esta función nos devuelve el día de un dato de tipo fecha.

Ejemplo:
fecha ← as_date("2021-12-10")
day(fecha)
Esto nos dará como resultado 10.

semester( ) : Esta función nos permite conocer en qué semestre del año se encuentra un dato de tipo fecha. Es decir, si se encuentra en el primer o segundo semestre.
Ejemplo:
fecha ← as_date(“2021-09-13”)
semester(fecha)
Esa línea de código nos devolverá un 2, debido a que septiembre (el mes 9) se encuentra en el segundo semestre del año.

quarter( ) : Esta función nos permite conocer en qué trimestre se encuentra un dato de tipo fecha.
Ejemplo:
fecha ← as_date(“2021-06-5”)
quarter(fecha)
Esto dará como resultado un 2 debido a que el mes de junio forma parte del segundo trimestre del año.

wday( ) : Esta función nos dice en qué día de la semana se encuentra una determinada fecha. Es importante destacar que se sigue el siguiente criterio:
1 - Domingo
2 - Lunes
3 - Martes
4 - Miércoles
5 - Jueves
6 - Viernes
7 - Sábado

Ejemplo:
fecha_1 ← as_date("2019-05-17")
wday(fecha_1)
Esto nos da como resultado el número 6, es decir, ese día fue viernes.
wday(fecha_1, label = TRUE)
Esto nos da como resultado vie o Fri (dependiendo la configuración de la computadora).

today( ) : Por medio de esta función podemos obtener la fecha del día de hoy. Es decir, si corremos la línea el 24 de diciembre de 2021, esa fecha es la que nos devolverá.

now( ) : Esta función es muy similar a la función today( ), con la diferencia de que además del día también te devuelve la hora exacta en la que la línea de código se corrió.

OlsonNames( ) : Para estas dos últimas funciones es necesario destacar que podemos consultar la hora de una determinada zona horaria, ésta función nos proporciona la lista de los nombres de las zonas horarias con el fin de ayudarnos a referenciar correctamente la zona horario de la que nosotros deseamos conocer la fecha y hora.

X. Manipulación de Strings

¿Para qué sirve la biblioteca stringr?

Nos sirve para manipular los datos de tipo string, también conocidos como caracteres.

¿Qué funciones podemos encontrar en esta biblioteca?

A continuación se muestra un listado de las funciones más importantes de esta biblioteca, así como la descripción de qué es lo que realiza cada una:

str_length( ) : Por medio de esta función podemos conocer cuántos caracteres conforman el string. Es decir, contabiliza la cantidad de letras, espacios, signos, etc.
Ejemplo:
str ← c(“¡Sylveon!”, “Tsareena y Crustle”)
str_lenght(str)
Esta línea de código nos dará como resultado 9 y 18, que es la cantidad de letras, signos y espacios que tiene cada uno de los strings que conforman el vector str.

str_c( ) : Con esta función podemos concatenar/juntar los elementos de un vector de caracteres.
Ejemplo:
x ← c("Hola", "Hi", "Hello")
str_c(x, collapse = " - ")
Esto nos dará como resultado el carácter "Hola - Hi - Hello".

str_sub( ) : Esta función nos permite extraer una determinada parte del string.
Ejemplo:
s ← “Spiderman No Way Home”
str_sub(string = s, start = 5, end = 11)
Esto nos dará como resultado “erman N“, ya que lo que realizó la función fue extraer desde el carácter número 5 hasta el 11.

str_extract( ) : A través de esta función podemos identificar patrones de strings en un vector.
Ejemplo:
y ← c("Sky", "Blue", "Try", "Orange")
str_extract(y, pattern = "[aeiou]")
Esto nos dará como resultado el vector c(NA, “u”, NA, “a”), pues el patrón que le estamos pasando como argumento hace referencia a buscar vocales en minúsculas dentro de la palabra. En este caso “Sky” y “Try” no tienen vocales, pero “Blue” tiene a “u” como primer vocal y “Orange” tiene a “a” como primer vocal en minúscula.
Nota: Los patrones deben ser expresiones regulares.

str_subset( ) : Esta función nos permite tomar los elementos de un vector que cumplen con un determinado patrón.
Ejemplo:
r ← c(“Wii”, “Switch”, “Xbox”, “Play”, “DS”)
str_subset(string = r, pattern = “[aeiou]”)

Esto nos dará como resultado un vector en donde encontraremos la mayoría de los elementos, excepto “DS”, debido a que determinamos que nuestro patrón fuera una vocal, es decir, todos los strings con vocales serán los que elegiremos y como “DS” no tiene vocal quedará excluido.

str_count( ) : Con esta función podemos contar el número de letras que cumplen cierto patrón en una palabra.
Ejemplo:
z ← c("Ana", "Pedro", "Luisa", "Mariano")
str_count(z, pattern = "[aeiou]")
Lo anterior nos dará como resultado el vector c(1, 2, 3, 4). En el caso de “Ana” nos dice que sólo tiene una vocal porque recordemos que R sí distingue entre mayúsculas y minúsculas.

str_detect( ) : Esta función nos devuelve un vector, conformado por datos tipo boolean, en el cual nos dirá qué elementos de un vector de strings cumplen con un determinado patrón.
Ejemplo:
Usaremos el mismo ejemplo que en el caso de str_subset( ).
r ← c(“Wii”, “Switch”, “Xbox”, “Play”, “DS”)
str_detect(string = r, pattern = “[aeiou]”)
Esta línea de código nos devolverá el siguiente vector c(TRUE, TRUE, TRUE, TRUE, FALSE). Cómo designamos que nuestro patrón fueran las vocales, asigna un valor de TRUE a los elementos de r que tienen vocales, como DS no tiene vocales le asignó el valor de FALSE.

str_replace( ) : Esta función nos sirve para reemplazar en un vector de strings las letras que cumplan con cierto patrón, para ello se debe indicar la palabra o letra que las reemplazará.
Ejemplo:
w ← c("Año", "Almendra", "Antes", "Abeja")
str_replace(w, pattern = "A", replacement = "a")
Esto nos dará como resultado el vector c(“año”, “almendra”, “antes”, “abeja”), pues le estamos pidiendo que a toda “A” que encuentre la reemplace por “a”.

str_split( ): Por medio de esta función podemos separar los strings que conforman un vector, conforme a un carácter de nuestra selección.
Ejemplo:
p ← c(“Rodrigo, Luna”, “Pedro, Karen, María”, “Isaac”)
str_split(string= p, pattern= “, “)
Este código nos dará como resultado la siguiente lista:
[1] “Rodrigo” “Luna”
[2] “Pedro” “Karen” “María”
[3] “Isaac”
Con esto podemos observar de una mejor manera lo que hace la función: nos devuelve una lista en donde los elementos son vectores, los cuales, a su vez, tienen por elementos los nombres que conformaban cada uno de los strings originales.

XI. Probabilidad

Distribución Binomial

rbinom(n = , size = , prob =)

La anterior línea de código se usa para generar simulaciones de la distribución binomial, donde:

- **n** es el número de observaciones que queremos que se generen.
- **size** es el número de ensayos o experimentos.
- **prob** es la probabilidad de que suceda el caso favorable.

Ejemplo:
rbinom(n = 10, size = 20, prob = 0.7)

Esto genera 10 observaciones, donde cada una consiste en 20 ensayos con probabilidad de éxito de 0.7.

Tenemos como resultado el vector c(13, 16, 14, 15, 14, 14, 15, 14, 14, 12)
Si observamos la primera entrada podemos notar que se obtuvieron 13 éxitos de 20.

length(sample[sample == m])

Por medio de esta línea de código podemos contar en cuántos de esos ensayos obtuvimos "m" éxitos, suponiendo que en la variable "sample" guardamos la muestra de nuestros experimentos con distribución binomial.

muestra ← data.frame (exitos = rbinom(n = 1000, size = 5, prob = 0.5))
ggplot (muestra, aes (x = exitos)) + geom_histogram (binwidth = 1)

Por medio de estas líneas de código estamos generando un dataframe con los datos obtenidos a partir de la muestra, y posteriormente usamos ese dataframe para poder generar la gráfica correspondiente. Vamos a analizar con mayor detenimiento el código anterior.
Si hacemos alusión al ejemplo de la moneda, en la primera línea notaremos que estamos lanzando 5 monedas 1000 veces, siendo la probabilidad de obtener un éxito es de 0.5, y estos datos los estamos guardando en un dataframe.
Posteriormente hacemos uso de este dataframe para poder mostrar en una gráfica esta información, es decir, en cuantos experimentos obtuvimos 0,1, 2, 3, 4 o 5 éxitos.

dbinom(x = , size = , prob =)

La anterior línea de código se usa para saber cuál es la probabilidad de que al hacer **size** experimentos binomiales, en **x** se tenga el caso favorable, donde:

- **x** es el número de observaciones que queremos que salgan favorables.
- **size** es el número de ensayos o experimentos.
- **prob** es la probabilidad de que suceda el caso favorable.

Ejemplo:
dbinom(x = 14, size = 20, prob = 0.7)

Con la anterior línea de código estamos preguntando: ¿Cuál es la probabilidad de que 14 de 20 experimentos binomiales tengan éxito (si la probabilidad de éxito es de 0.7)?
Esto nos da como resultado 0.191639.

pbinom(q = , size = , prob =)

A través de esta función podemos obtener la función de probabilidad acumulada de la distribución binomial en un punto dado. Dicho de otra manera, esta función nos permite conocer cuál es la probabilidad de obtener **q** éxitos o menos en un experimento que consta de **size** intentos y cada uno tiene probabilidad **prob** de éxito.

Ejemplo:
pbinom(q = 4, size = 10, prob = 0.5)

qbinom(p = , size = , prob =)

La anterior línea de código se usa para calcular los cuantiles de una distribución binomial, donde:

- **p** es el cuantil.
- **size** es el número de ensayos o experimentos.
- **prob** es la probabilidad de que suceda el caso favorable.

Ejemplo:
qbinom(p = 0.5, size = 40, prob = 0.3)

La anterior línea de código nos da el cuantil que acumula el 50% de nuestros datos en 40 ensayos, donde la probabilidad de éxito es de 0.3.
Esto nos da como resultado 12.

Distribución Normal:

rnorm(n = , mean = , sd =)

Al igual que en el caso anterior, esta función nos permite generar una muestra con **n** datos a partir de una distribución normal con parámetros **mean** y **sd**.

Ejemplo:
rnorm (n = 1500, mean = 50, sd = 5)
Al correr lo anterior obtendremos 1500 datos que provienen de una distribución normal con media 50 y desviación estándar 5.

rnorm(n =)

Esta función es la misma que la anterior, pero en este caso no le estamos pasando los datos de la media y la desviación estándar. De este modo la función nos generará **n** datos

correspondientes a una distribución normal con media 0 y desviación estándar de 1, los cuales son los valores que la función asigna por default a estos parámetros y que corresponden a los de una normal estándar.

dnorm(x = , mean = , sd =)

Esta función sirve para calcular el valor de la función de densidad en el punto **x**, donde la media es **mean** y la desviación estándar es **sd**.

Ejemplo:
dnorm(x = 40 , mean = 50, sd = 10)
La anterior línea de código nos da el valor de la función de densidad normal con media 50 y desviación estándar 10 en el punto x = 40.
Esto nos da como resultado 0.02419707.

pnorm(q = , mean = , sd =)

Esta función nos permite calcular la función de distribución o de probabilidad acumulada en un punto dado **q** para una distribución normal de media **mean** y desviación estándar **sd**.

Al igual que en los casos anteriores, podemos omitir los parámetros mean y sd, con lo cual R les asignará los valores de 0 y 1 respectivamente, que corresponden a los de una distribución normal estándar.

Ejemplos:
pnorm(q = 0, mean = 15, sd = 8), para una normal de media 15 y desviación estándar 8.
pnorm(q = -1.64), para una normal estándar.

sample ← rnorm(2350)
sampleDen ← dnorm(sample)
ggplot(data.frame(x = sample, y = sampleDen)) + aes (x = x , y = y) + geom_line( )
+ labs (x = “x”, y = “Density”)

Por medio de estas líneas de código podemos graficar los datos correspondientes a una muestra de una distribución normal. Primero estamos generando la muestra de los datos, en este caso de una normal estándar, para posteriormente aplicarles la función de densidad. Con la última línea de código estamos juntando estos dos vectores en un dataframe para poder graficarlo con la función de ggplot( ).