Programación en R

Vision general e historia de R

R es el dialecto de S

Que es S?

- S es un lenguaje que fue desarrollado por John Chambers en los ahora desaparecidos Bell Labs - Laboratorios Bell.
- Su desarrollo fue iniciado en 1976 como un entorno de análisis estadístico para uso interno, es decir, un entorno que la gente de Bell Labs pudiese usar para analizar sus datos. Inicialmente se implementó como una serie de librerías en FORTRAN para implementar rutinas que eran tediosas de realizar una y otra vez. Por esta razón había librerías de FORTRAN para repetir estas rutinas estadísticas.
- Las primeras versiones del lenguaje no contenían funciones para modelado estadístico. Eso no fue incluido sino hasta aproximadamente la tercera versión del lenguaje.
- Así, en 1988 el sistema fue reescrito en C para mejorar la portabilidad entre sistemas y fue cuando comenzó a parecerse al sistema que tenemos hoy en día. Eso fue en la tercera versión. Hubo un libro original llamado "Modelos estadísticos en S" escrito por John Chambers y Trevor Hastie, a veces conocido como el "libro blanco", en el que se documenta toda la funcionalidad de análisis estadístico que se incluyó en esa versión del lenguaje.
- La versión cuatro del lenguaje S se lanzó en 1998. Y esa es más o menos la versión que usamos hoy en día. El libro "Programming with Data", que es de referencia para este curso, fue escrito por John Chambers, y en ocasiones es llamado el "libro verde". Ese libro documenta la cuarta versión del lenguaje S. De esta manera, R es una implementación del lenguaje S, que fue originalmente desarrollado en Bell Labs.

Volvamos a R:

- 1991: Fue creado en 1991 en Nueva Zelanda por dos caballeros llamados Ross Ihaka y Robert Gentleman. En un artículo científico publicado en 1996 en el "Journal of Computation and Graphical Statistics", estos dos caballeros contaron su experiencia durante el desarrollo de R.
- En 1993 se hizo público el primer anuncio de R.
- En 1995, Martin Michler convenció a Ross y a Robert de poner R bajo una Licencia GNU para el público general (GNU-GPL). Hablaremos sobre esto un poco más adelante. Ese hecho convirtió R en lo que llamamos "software libre" - free software

_

- En 1996 se crearon un par de listas de correo Una llamada R-help, que es una lista general para preguntas. Y otra llamada R-devel, que es una lista de correo más específica para gente que participa en trabajos de desarrollo para R.
- En 1997 se formó el grupo llamado "R ore group" - grupo básico de R - Que contenía a mucha de la misma gente que desarrolló S-PLUS. El "core group" básicamente controla el código base (source code) de R. El código base fundamental de R sólo puede ser modificado por miembros del "R core group". Sin embargo, algunas personas que no hacen parte del "core group" han sugerido cambios en R que han sido posteriormente aceptados por el "core group".

Dise;o del sistema R:

- Tiene algo de 4000 paquetes en CRAN que han sido desarrollados por usuarios y programadores en todo el mundo
- Hay también algunos paquetes asociados con el proyecto Bioconductor que es un proyecto diseñado para implementar software en R para análisis de datos genéticos y biológicos. (https://bioconductor.org)
- están todos los paquetes hechos por personas y publicados en sus páginas web personales. Y en verdad no hay una manera confiable de rastrear cuántos paquetes están disponibles de esa forma. Entonces, en realidad hay miles de paquetes disponibles en la web que puedes encontrar y usar para analizar datos.

Pidiendo ayuda

Stupid: “Help! Can’t fit linear model!”
Smart: “R 3.0.2 lm() function produces seg fault with large data frame, Mac OS X 10.9.1”
Smarter: “R 3.0.2 lm() function on Mac OS X 10.9.1 – seg faut on large data frame”
Hacer:

- Describe tu objetivo final y luego cuales son los problemas, no lo reduzcas todo al peque;o paso en el que estas teniendo problemas
- Se explicito con tu pregunta, recuerda proporcionar detalles sobre que quieres hacer
- Facilita la cantidad mínima necesaria de información necesaria, no la máxima cantidad de información, sino la munima
- Se cortes y educado, no ofendas a nadie
- Seguimiento con la solución (encontró)

No hacer:

- Afirma que has encontrado un “gub” (error de codigo)
- Publicar la lista de los ejercicios en listas de correos oforos
- Escribir a múltiples listas de correo a la vez, es importante averiguar cual de estas listas es la mas apropiad para la pregunta y enviarla solo a esa lista o foro
- Pedir a otros que depuren tu código sin dar algunas indicaciones acerca de cual puede ser el problema. Especifica donde creemos que esta el problema y lo que estamos intentando hacer

Entrada y evaluación de la consola de R

Las cosas que escribimos en el indicador R se llaman expresiones

Vectores y listas

```
> y <- 0:6
> as.logical(y)
[1] FALSE  TRUE  TRUE  TRUE  TRUE
[6]  TRUE  TRUE
> as.character(y)
[1] "0" "1" "2" "3" "4" "5" "6"
```

La coerción no siempre funciona:

```
> x <- c("a","b","c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introducidos por coerción
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] NA NA NA
Warning message:
NAs introducidos por coerción
```

Las listas son como un vector excepto que cada elemento de la lista puede ser un objeto de una clase diferente lo cual las hace muy útiles para mantener distintos datos

```
> z <- list(1,"s", TRUE, 1+4i)
> z
[[1]]
[1] 1

[[2]]
[1] "s"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Matrices

Vectores que tienen atributo dimensión

```
> # Matrix vacia
> m <- matrix(nrow = 2,ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

```
> dim(m)
[1] 2 3
> attributes(m) # lista de atributos que tiene m
$dim
[1] 2 3

> m2 <- matrix(1:6,nrow = 2,ncol = 3)
> m2
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Las matrices se pueden crear directaente de vectores y aadiendo la dimensión atributo

```
> m3
 [1]  1  2  3  4  5  6  7  8  9 10
> m3 <- 1:10
> m3
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m3) <- c(2,5) # 2 filas y 5 columnas
> m3
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Cbind-ing y rbind-ing

Las matrices se pueden crear añadiendo columnas y filas con `rbind()` y `cbind()`

```
> a <- 1:3
> b <- 10:12
> cbind(a,b) # añade una columna, c de column
     a  b
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(a,b) # añade una fila, r de row
  [,1] [,2] [,3]
a    1    2    3
b   10   11   12
```

Factores

Es un tipo especial de vector que es utilizado para representar categorías. Pueden ser ordenados y no ordenados. Son como un conjunto de datos que tienen etiquetas con categorías pero que no tienen un orden o pueden tener un orden por jerarquía, pero no numérico

Ejm:

- “alto”, “bajo”, “medio”
- 1, 2, 3
- “fuerte” ,”débil”

Los factores son especialmente creados para el modelamiento como `lm()`, y `glm()`

Usar factores con etiquetas es mejor que usar enteros porque factores se autodescriben

```
> f <- factor(c("yes","yes","no","yes","no"))
> f
[1] yes yes no  yes no
Levels: no yes
> table(f)
f
 no yes
  2   3
> unclass(f)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

El orden de los niveles puede ser un cnjunto usando el argumento levels en `factor()`. Esto es importante porque el primer nivel es usado como el nivel base (el primer nivel del factor)

```
> f2 <- factor(c("yes","yes","no","yes","no"),
+              levels = c("yes","no")) # si no ponemos levels, “no”, va primero
que “yes”
> f2
[1] yes yes no  yes no
Levels: yes no
```

Valores perdidos

Son denotados por `NaN` por operaciones matemáticas indefinidas y `NA` este se usa para prácticamente cualquier otro caso

`is.na()` es usado para ver si los objetos son `NA`
`is.nan()` es usado para ver si es `NaN`
Valores `NA` también pueden tener una clase, por lo que puedes tener valores faltantes enteros o de tio carácter o numérico, etc. A pesar de que parezcan que todos los `NA` son iguales pueden tener distintas clases
El valor `NaN`, se considera tamben un `NA`. Pero un `NA` no son necesariamente valores `NaN`. Un `NaN` es tambien un valor faltante.

```
> v <- c(1,2,NA,10,3) # como son números ese NA es de tipo numérico
> is.na(v)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE
> v2 <- c(1,2,NaN, NA,4)
> is.na(v2)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(v2)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Es un tipo de dato clave usado en R y es utilizado para almacenar datos tabulados, la mayoría de datos que usamos son tabulados pero no todos.
Los dataframes se representan básicamente como un tipo especial de lista en la cual cada elemento de la lista tiene la misma longitud.
No todas las columnas son del mismo tipo
Los dataframes pueden ser creados llamando a las funciones "`read.table`" y "`read..csv`"
Se puede crear una matriz a través de un `dataframe` usando la función `data.matrix`, pero claro tienen que tener los mismos tipos de datos sino todos se convierten en una misma clase

```
> ma <- data.frame(foo = 1:4, bar= c(T,T,F,F))
> ma
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(ma)
[1] 4
> ncol(ma)
[1] 2
```

Atributo de nombres

Los objetos en R pueden tambien tener nombres, que es muy útil para escribir código legible y objetos autodescriptivos

```
> na <- 1:3
> names(na) # por defecto no tiene nombres
NULL
> names(na) <- c("foo","bar","norf") # se le da nombre a cada elemento del
vector
> na
 foo  bar norf
   1    2    3
> names(na)
[1] "foo"  "bar"  "norf"
```

Las listas tmb pueden tener nombres

```
> na2 <- list(a=1,b=2,c=3)
> na2
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

La matrices igual

```
> na3 <- matrix(1:4,nrow = 2,ncol=2)
> dimnames(na3) <- list(c("a","b"),c("c","d"))
> na3
  c d
a 1 3
b 2 4
```

Leyendo datos

Hay algunas funciones para leer datos en R

- `read.table`, `read.cvs`, lee datos tabulados
- `readLines`, lee líneas de texto, esto puede ser cualquier tipo de archivo realmente, simplemente le da texto en un como un vector de caracteres de R
- `source` es importante para leer código R (inverso de `dump`)
- `dget` tambien es para leer archivos R pero es para leer objetos R que se han analizado en archivos de texto (inverso de `dput`)
- `load` leen en espacios de trabajo guardados
- `unserializa` para leer objetos binarios en R, igual `load`

Para escribir datos:

- `write.table`
- `writeLines`
- `dump`
- `dput`
- `save`
- `serialize`

Leyendo datos con `read.table`:

Es la más común para leer datos y tiene los siguientes argumentos:

- `file`, es el nombre de la fila o la conexión.
  El nombre del archivo desde el que se leerán los datos
- `header` es un indicativo lógico si la fila tiene una cabecera
- `sep` una cadena que indica como se separan las columnas
- `colClasses` un vector de caracteres que indican la clase de cada columna en el dataset
- `nrows` numero de filas
- `skip` el numero de líneas para saltar desde el principio
- `stringAsFactors` ¿deben codificarse las variables de carácter como factor?

Leyendo tablas largas

Leer los datasets con `read.table`

- Leer la pagina de ayuda de `read.table` que contiene algunas hintuiciones

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals=c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
```

```
allowEscapes = FALSE, flush = FALSE,
stringsAsFactors = default.stringsAsFactors(),
fileEncoding = "", encoding ="unknown", text, skipNul = FALSE)
```

- Hacer un cálculo áspero de la memoria requerida para comprar tu dataset. Si el dataset requiere más memoria que la RAM tú puedes probablemente detenerte ahí
- Establecer `comment.char = “ ”` si no hay líneas de comentario en tu archivo
- El argumento de las clases de llamada es realmente muy importante., si no lo especifica, lo que R hace de forma predeterminada es que pasa por cada columnas y averigua que tipo de dato es . Esto esta bien cuando es un conjunto de datos peque;o o moderado. Pero leer cada una de estas columnas e intentar averiguar que tipo de dato es lleva tiempo, memoria y generalmente puede ralentizar las cosas. Si se especifica R no tiene que perder el tiempo para averiguarlo por si mismo por lo que `read.table` se ejecutara mas rápido
- Fvv

```
initial <- read.table("datatable.txt",nrows = 100)
classes <- sapply(initial, class)
tabALL <- read.table("datatable.txt",
                     colClasses = classes)
```

En fgenerl cuando esta usando R con grades conjuntos de datos hay muchos conjuntos de datos grandes por ahí hoy en día . Es útil tener algunas cosas, algunos bits de información a mano:

- ¿Cuanto de memoria tiene tu computadora? ¿Cuanta ram física hay?
  Un dataset de 1000 filas por 20 columnas donde todos son numéricos:T=1000x20x8bit
- ¿Que aplicaciones están en uso?
- ¿Hay otros usuarios iniciando sesión en el sistema? ¿Esta usando algunos recursos del ordenador?
- ¿Cuál es el sistema operativo?
- EL sistema operativo que esta ejecutando es de 32 o 64 bits

Formatos de datos textuales

Dput() and dump()
Existen otros tios de formato en los que puedes almacenar datos además del formato tabular, los archivo csv o los archivos de texto plano. Estos tmb son formatos de texto pero difieren un poco.
2 funciones principales par la escritura de datos son “dump” y “dput”. Si bien son formatos de texto, no están formateado de la misma manera que una taba pues tmb contiene metadatos
Ej: la clase o el tipo de dato de cada objeto. Si haces un “ump” o “dput” de un “dataframe” incluirá la salida de la clase de cada columna del “dataframe”

Dumping y dputing son utiles porque el formato textual resultant es editable y en el caso de corrupción, potencialmente recuperable

- Unlike escribiendo fuera de una tabla o archivo cvs, dump y dput preserva los metadatos (sacrificando algo de legilibilidad), de modo que otro usuario no tenga que especirficarlo todo de nuevo.
- Textual los formatos textuales pueden funcionar mejor con los programas de versión de control como subersion o git que solo pueden rastrear cambios significativos en archivos de texto.
- Los archivos textuales pueden durar mas, si hay una ruptura en algún lugar del archivo, puede ser mas fácil solucionar el problema.
- Los formatos textuales siguen mas o menos parte de la filosofía Unix la cual guarda todo tipo de información generalmente en formatos de texto
- Desventaja: el formato no es muy eficiente en materia de espacio, Suelen ocupar mucho espacio por lo que frecuentemente se comprimen

```
> tf <- data.frame(a=1,b="a")
> dput(tf)
structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
-1L))
> dput(tf,file="tf.R") # nuevo archive R
> new.tf <- dget("tf.R") # leyendo archivo R
> new.tf
  a b
1 1 a
```

Lo que hace dput es escribir código R que luego puede ser utilizado para econstruir un objeto en R

La función dump es muy parecida a dget, pero el dget solo puedes pasarle un objeto en R a la vez mientras que dump puede utilizarse con múltiples objetos en R a mismo tiempo

```
> t2 <- "foo"
> tf2 <- data.frame(a = 1, b= "a")
> dump(c("t2","tf2"), file = "data.R") # archivo donde quieres guardar los obje
tos
> rm(t2,tf2)
> source("data.R") # los objetos t2 y tf2 han sido reconstruidos
> tf2
  a b
1 1 a
> t2
[1] "foo"
```

Vfv

Interfaces con el mundo exterior

Hay unciones que se utilizan para abrir lo que se llama conexiones con el mundo exterior. Normalmente el tipo más común de conexión es con un archivo, por ejemplo, si se desea leer un archivo, puede crear una conexión de archivo, es posible que desee o leer un archivo comprimido o una ligera variación de eso

- `file` abre una conexión a un archivo
- `gxfile`, abre una conexión a un archivo comprimido con `gzip`

- `bxfile` abre una conexión a un archivo comprimido con `bzip2`
- `url` abre una conexión a una página web

Conexiones de archivos

```
> str(file)
function (description = "", open = "", blocking = TRUE,
    encoding = getOption("encoding"), raw = FALSE,
    method = getOption("url.method", "default"))
```

description es el nombre del archivo
open es es la indicación del código
“r” solo leer
“w” escribir
“a” agregar
“rb”, “wb”, “ab” leer escribir y agregar archivos binarios (Windows)

Conexiones son potentes y pueden permitirse navegar archivos y otros objetos extremos de una manera mas sofisticada que simplemente.
En la practica , a menudo no necesitamos lidiar directamente con la interfaz de conexión.

```
> con <- file("foo.txt", "r")
> data <- read.csv(con) # la conexión y lee todo el archivo por defecto
> close(con) # cierra la conexión
```

Es lo mismo que

```
> data <- read.cvs("foo.txt") # en este caso no hubo necesidad de hacer
conexión para leer el archivo
```

Se puede usar readlines para leer elementos de una pagina web
Usando url se crea una conexión a un sitio web

```
> con <- url("http://www.jhsph.edu","r")
> x <- readLines(con) # se abre la conexion
> head(x)
[1] "<!DOCTYPE html>"
[2] "<html lang=\"en\">"
[3] ""
[4] "<head>"
[5] "<meta charset=\"utf-8\" />"
[6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

Tg

Subconjuntos: Conceptos básicos

Extraer subconjuntos de objetos en R

- [siempre retorna un objeto de la misma clase que el original, puedes usarlo para seleccionar mas de un elemento (hay una excepción)

- `[[` es usado para extraer elementos de una lista un data frame; esto puede solo ser usado para extraer un elemento y la clase del objeto retornado no será necesariamente una lista o data frame, pueden tener diferentes clases
- `$` es usado para extraer elementos de una lista o data frame que tienen nombre; semánticamente es similar a `[[` en el sentido que las clases pueden ser distintas

```
> s <- c("a","b","c","c","e")
> s[1]
[1] "a"
> s[2]
[1] "b"
> s[1:4]
[1] "a" "b" "c" "c"
> s[s > "a"]
[1] "b" "c" "c" "e"
> u <- s > "d"
> u
[1] FALSE FALSE FALSE FALSE  TRUE
```

Subconjuntos: Listas

Subestablecer una lista es algo diferente

```
> sl <- list(foo = 1:4, bar = 0.6)
> sl[1]
$foo
[1] 1 2 3 4  # devuelve una lista

> sl$bar
[1] 0.6
> sl[["bar"]]
[1] 0.6
> sl["bar"]
$bar
[1] 0.6
```

Tmb Podemos asarle un vector a los corchetes

```
> sl2 <- list(foo = 1:4, bar = 0.6, baz = "hello")
> sl2[c(1,3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

$ solo se usa para nomres literales y [[]] para índices

```
> name <- "foo"
> sl2[[name]]
[1] 1 2 3 4
> sl2$name
NULL
> sl2$foo
```

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

Entrando con indices a los elementos

```
> sl3 <- list(a = list(10,12,14), b = c(3.14,2.81))
> sl3[[c(1,3)]]
[1] 14
> sl3[[1]][[3]]
[1] 14
> sl3[[c(2,1)]]
[1] 3.14
```

Gb

Subconjuntos: Matrices

Fv

```
> sma <- matrix(1:6,2,3)
> sma[1,2]
[1] 3
> sma[2,1]
[1] 2
> sma
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> sma[1,] # primera fila
[1] 1 3 5
> sma[,2] # segunda columna
[1] 3 4
```

Trtgf\

```
> sma2 <- matrix(1:6,2,3)
> sma2[1,2]
[1] 3
> sma2[1,2,drop = FALSE]
     [,1]
[1,]    3
```

Jytnf

Subconjuntos: Coincidencia parcial

Es una herramienta útil que te ahorra mucho al tipear en la línea de comandos, no es paricularmente útil cuand estas escribiendo programas y funciones, pero si cuando trabajas en línea de comandos

Se permite la coincidencial parcial para nombres con [[y $

```
> cp <- list(aardvark = 1:5)
> cp$a  # objeto asociado con aarvark
[1] 1 2 3 4 5
> cp[["a"]]
NULL  # no hay nngun elemento en la lsta que se llame a
> cp[["a", exact = FALSE]]
[1] 1 2 3 4 5  # esa es la que mejor coincide con la letra “a”
```

Gf

Subconjuntos: Removiendo valores perdidos

Es común en análisis de datos encontrar muchos valores faltantes
Para removerlosse vecesita un vector lógico que diga donde están los NA para que pueda eliminrlos mediante subconfiguracion

```
> vf <- c(1,2,NA,4,NA,5)
> bad <- is.na(vf)
> vf[!bad]
[1] 1 2 4
```

Fv

```
> vf <- c(1,2,NA,4,NA,5)
> vf2 <- c("a","b",NA,"d",NA,"f")
> good <- complete.cases(vf,vf2)
> good
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> vf[good]
[1] 1 2 4 5
> vf2[good]
[1] "a" "b" "d" "f"
```

Tambien elimina filas completes por los datos faltantes

```
> airquality[1:6, ]
  Ozone Solar.R Wind  Temp  Month Day
1    41     190  7.4    67      5   1
2    36     118  8.0    72      5   2
3    12     149 12.6    74      5   3
4    18     313 11.5    62      5   4
5    NA      NA 14.3    56      5   5
6    28      NA 14.9    66      5   6
> good <- complete.cases(airquality) # da as filas que estan completas
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind  Temp  Month Day
1    41     190  7.4    67      5   1
2    36     118  8.0    72      5   2
3    12     149 12.6    74      5   3
4    18     313 11.5    62      5   4
7    23     299  8.6    65      5   7
```

Operaciones vectorizadas

A

```
> ov <- 1:4
> ov2 <- 6:9
> ov < 2
[1]  TRUE FALSE FALSE FALSE
> ov >= 2
[1] FALSE  TRUE  TRUE  TRUE
> ov2 == 8
```

```
[1] FALSE FALSE  TRUE FALSE
> ov * ov2
[1]  6 14 24 36
> ov / ov2
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

D

```
> ov3 <- matrix(1:4, 2, 2)
> ov4 <- matrix(rep(10,4), 2, 2)
> ov3*ov4  # multiplica element por elemento
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> ov3/ov4
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> ov3%*%ov4  # verdadera multiplicación de matrices
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

F

Estructuras de control if-else

F

```
if(x > 3){
  y <- 10
} else{
  y <- 0
}
y <- if(x>3){
  10
} else{
  0
}
```

A

Estructuras de control: buble for

A

```
> x <- c("a","b","c","d")
> for(i in 1:4){
+   print(x[i])
+ }
> for(i in seq_along(x)){ # devuelve como una secuencia desde 1 hasta la
longitude de x
+   print(x[i])
+ }
> for(letter in x){
+   print(letter)
+ }
> for(i in 1:4) print(x[i])
```

Todos devuelven

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Anidacion de bucles

```
> f <- matrix(1:6,2,3)
> for( i in seq_len(nrow(f))){ # seqlen toma un entero
+   for(j in seq_len(ncol(f))){
+     print(f[i,j])
+   }
+ }
[1] 1
[1] 3
[1] 5
[1] 2
[1] 4
[1] 6
```

a

Estructuras de control: buble while

a

```
> count <- 0
> while(count<10){
+   print(count)
+   count <- count +1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

Mas de una condición en la prueba

```
> x <- 5
> while(x >= 3 && x<= 10){
+   print(x)
+   coin <- rbinom(1,1,0.5) # numero aleatorio pero tomando valores 0 y 1
+
+   if(coin == 1){ ## random walk
+     x <- x+1
+   } else{
+     x <- x-1
+   }
+ }
[1] 5
[1] 6
[1] 5
```

```
[1] 4
[1] 5
[1] 4
[1] 5
[1] 4
[1] 3
```

S

Estructuras de control: repeat, next, break

D

```
> x0 <- 1
> tol <- 1e-8
>
> repeat {
+   x1 <- computeEstimate()
+   if(abs(x1-x0) < tol){
+     break
+   } else{
+     x0 <- x1
+   }
+ }
```

Next se usa en cualquier momento de construcción de buble cuando deseas omitir una iteración

```
> for( i in 1:100){
+   if(i <=20){
+     ## skip the firs 20 iterations
+     next
+   }
+   ## do something here
+ }
```

W

Primera function en R

A

```
> add2 <- function(x, y){
+   x + y
+ }
> add2(3,5)
[1] 8
>
> above10 <- function(x){
+   use <- x>10
+   x[use]
+ }
>
> above10(1:20)
 [1] 11 12 13 14 15 16 17 18 19 20
> above <- function(x,n){
+   use <- x>n
+   x[use]
```

```
+ }
> above(1:20,12)
[1] 13 14 15 16 17 18 19 20
>
> columnas <- function(y, removeNA = TRUE){
+   nc <- ncol(y)
+   means <- numeric(nc)
+   for(i in  1:nc){
+     means[i] <- mean(y[,i], na.rm = removeNA)
+   }
+   means
+ }
> columnas(a)
[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

Aw

Funciones Parte 1

Las unciones son importantes y representan realmente la transición de un usuario
Las funciones son almacenadas como objetos en R igual que todo lo demás. Sn objetos que pertenecescen a la clase “function”
Las funciones son consideradas como objetos de primra clase
Podemos pasar funciones como parámetros a otras funciones, las funciones tmb pueden anidarse
Coincidencia de argumentos:

```
> mydata <- rnorm(100) # datos aleatorios de la dostribucion normal
> sd(mydata)
[1] 1.091379
> sd(x = mydata)
[1] 1.091379
> sd(x = mydata, na.rm = FALSE) # na.rm controla si los valores que faltan
deben ser eliminados de la entrada o no, su valor por defecto es FALSE
[1] 1.091379
> sd(na.rm = FALSE, x = mydata)
[1] 1.091379
> sd(na.rm = FALSE, mydata)
[1] 1.091379
```

Las asignaciones de argumentos por nombre y por posición pueden mezclarse y a menudo es bastante útil en funciones con listas de argumentos muy largas

```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
> lm(data = mydata, y ~ x, model = FALSE, 1:100)
> lm(y ~ x, mydata, 1:100, model = FALSE) # son equivalentes
```

Ver de nuevo
Tmb puede haber coincidencia parcial

Funciones Parte 2

Además de no especificar un valor predeterminado, puedes también puedes establecer un argumento como nulo

```
> f <- function(a, b = 1, c = 2, d = NULL){
+   # null es un valor muy comun par asignar a un argumento
+ }
```

Evaluación perezosa:

Es un modelo común en un montón de lenguajes de programación. La forma en la que funciona es que todos los argumentos de una función solo se evalúan a medida que se necesitan

```
> f2 <- function(a, b){
+   a^2
+ }
```

En este caso el argumento b nunca es evaluado

```
> f3 <- function(a,b){
+   print(a)
+   print(b)
+ }
> f3(45)
[1] 45
Error in print(b) : el argumento "b" está ausente, sin valor por omisión
```

Se aplica la evaluación erezosa porque solo se evalua cuando es necesario

El argumento …:

Indica un numero variable de argumentos, que a veces se pueden pasar a otras funciones. Se usan cuando se extienden otras funciones y no quieres copiar casi toda la lista de argumentos de la función original

Existe otro uso del aegumento … y espara lo que se conoce como funciones genéricas

```
> myplot <- function(z, y, type = "l", ...){
+   plot(x, y, type = type, ...)
+ }
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x000001bb43087668>
<environment: namespace:base>
```

D

```
> args(paste)
function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
NULL
> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
NULL
```

Argumentos despues de … :

```
> paste("a","b", sep = ":")
[1] "a:b"
> paste("a","b",se = ":")
[1] "a b :"
```

Los argumentos que aparezcan dsp de … en la lista de argumentos deben tener nombre explicito y además no puede coincidir parcialmente, por lo que no se puee usar coincidencia posicional o

La coincidencia parcial se ignora en se en otras circunstancias haría coincidencia parcial, pero paste no puede usar coincidencia parcial por lo que simplemente lo ignora y asume como una cadena mas que concatenar

Reglas de alcance: Enlace de símbolos

La reglas de alcanc son la forma como el lenguaje resuelve la asociación entre una variable o simbolo libre y su correspondiente valor
Un tema importante es cuando una función ve un símbolo en su cuerpo y se esta ejecutando dentro dell entorno de R ¿Cómo asigna un valor a ese simbolo?

```
> lm <- function(x){
+   x*x
+ }
> lm
function(x){
  x^x
}
```

Definimos una función lm pero ya existe una función lm (usada para regresión lineal, dentro del paquete stats), teniendo las 2 funciones ¿Cómo sabe R que valor asignar al símbolo “lm”?
La idea es que R necesita enlazar un valor con un símbolo. Este símbolo en el ejemplo era “lm” y necesita asociarse con un valor y este valor va a ser una función de algún tipo.
Cuando R trata de asociar un valor a un símbolo lo que hace es buscar a través de una serie de entornos para encontrar el valor apropiado, los entornos son un tipo de listas de objetos y valores o símbolos y valores, y cuando estás trabajando en línea de comandos y necesitas recuperar un valor de un objeto de R, básicamente:

- Buscas en el entorno global el nombre de un símbolo que coincida con el que se ha pedido. El entorno global es solo tu espacio de trabajo y consiste en todas las cosas que has definido o cargado en R. Entonces si hay un símbolo ahí que coincida come el nombre que estas solicitando tomará el símbolo y devolverá el valor asociado con ese símbolo.
- En el caso del ejemplo, he definido “lm” en mi entorno global y porque este existe, si yo estoy trabajando en la línea de comandos, cuando llamo a “lm” va a encontrar ese objeto primero. Entonces si no existe una coincidencia en el entorno global, lo que sucede es que:
- R buscara los “namespace” para que cada paquete en la lista de búsqueda. Así la lista de liusqueda consiste en todos los paquetes de R actualmente cargados en R. Y como se ve, existe un orden en la lista de búsqueda

```
> search()
 [1] ".GlobalEnv"        "tools:rstudio"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

El primero es el entorno global, el 3er paquete es el paquete stats y graphics, grDevices, el ultimo el paquete “base”
Entonces en alguna parte e la lista de paquetes R va a buscar una función llamadas “lm”, primero buscara al entorno global luego a los demás
Enlazando valores a simbolos:

- El entorno global es equivalente al espacio de trabajo de usuario, y siempre es el primero elemento de la lista de búsqueda. Además el paquete base es siempre es el ultimo de la lista de búsqueda
- El orden de los paquets en la lista es importante
- Los usuarios pueden configurar qué paquetes se cargan al arrancar, además los usuarios pueden cargar paquetes en el momento que lo deseen. Así que no puedes asumir que haya una lista de paquetes disponibles o que los paquetes se ordenan de una forma en particular. Pueden ordenarse de diferentes maneras en cualquier momento dependiendo e lo que el usuario decida que necesita para una sesión concreta.
- Cuando el usuario carga un paquete con una función de una librería, lo que ocurre es que el “namespace” del paquete que es el entorno que contiene todos los nombres, todos los simbolos y los valores de los mismos, se inserta en la 2da posición de la lista de búsqueda justo detrás del entorno global. Entonces todo lo demás simplemente baja al nivel siguiente de la lista. Y así cuando se produzca una búsqueda esta incluirá el nuevo paquete además de los otros paquetes que ya estaban en la lista desde un principio
- Notemos que R tiene “namespace” separados para funciones y no-funciones, por lo que es posible que existan a la vez un objeto “c” y una función también llamada “c”. Por supuesto en el entorno global solo puede haber un simbolo llamado “c” pero es posible tener un vector llamado “c” y no tiene por que interferir una función llamada “c” que ya existe

Relas de alcance (scoping) de R:
Son una de las cosas que distinguen a R del lenguaje S original

- La reglas de alcance determinan como un valor se relaciona con una variable libre de una función. Dentro de una función hay 2 tipos de variables: los argumentos (se pasan por medio de la definicion) y pueden haber otras variables y símbolos que están dentro de la función pero no son argumentos y como se le asigna un valor?
- R usa el alcance léxico (lexical scopng o alcance estático). En contraposición al alcance dinámico
- Las reglas de alcance de relacionan con la forma en la que R usa la lista de búsqueda para asignar valor a un símbolo
- Una de las ventajas del alcance léxico es que es util a la hora de simplificar cálculos estadisticos

```
> f4 <- function(x, y){ # tiene 2 argumentos formales
+   x^2 + y/ z # el símbolo “z” es lo que se llama variable libre, puesto que no ha sido definido en la cabecera de la función
+ }
```

Que valor se le asigna a z?
Las reglas de alcance de un lenguaje serán lo que determine que valores se le asignaran a las variables libres como z

Alcance léxico:
Significa que “los valores que se le darán a las variables se buscaran en el entorno en el que la funcion se ha definido”
Que es un entorno?

- Un entorno es una colección de parejas símbolo – valor ((símbolo valor)). Por ejemplo: “x” es un símbolo y “3.14” puede ser su valor
- Cada entorno tiene un “entorno padre”, como si el estuviese por encima de el y heredarse de el. Un entorno puede tener varios descendientes de manera que puede haber un entorno padre y varios entornos hijos
- Solo hay un entorno sin padre, y se trata del entorno vacio. R usa muchos tipos de entorno. Puedes pensar que el entorno global que es tu espacio de trabajo es un juego e parejas símbolo valor. Así que tienes un pu;ado de objetos creados e tu espacio de trabajo y que todos ienen nombres y cada uno de estos tiene asociado un objeto (que puede ser u vector numérico, data frame o lista o lo que sea)
- La clave es que si tomamos una funcion y le asociamos un entorno entonces eso crea lo que se le conoce como una “clausura” o “clausura de una funcion”

Bubscando el valor de la variabe libre:

- Si el valor del símbolo no se encuentra en el entorno en la cual la funcion se definió, entonces la búsqueda continua en el entorno padre
- La búsqueda continua abajo, la secuencia de padres entornos hasta que lleguemos al nivel de entorno mas alto, esto usualmente es el entorno de nivel superior es el namespace de ese paquete
- Después del entorno de nivel superior, la búsqueda continua en la lista de búsqueda hasta que se llegue al entorno vacio. Si no puedes encontrar un símbolo en todos estos entornos y llegas al entorno vacion, entonces se devuelve un mensaje de error diciendo que no se puede encontrar un valor para ese simbolo

Reglas de alcance: Reglas de alcance en R

Por que es important todo esto?
Normalmente las funciones se definen en el entorno global, de manera que los valores de las variables libres se hallan en el espacio de trabajo del usuario. Sin embargo se puede devolver una funcion en una funcion de manera que el entorno en el que ha sido definida no es el entorno global sino el interior de otra funcion. Y aquies donde las cosas se vuelven interesantes

```
> make.power <- function(n){
+   pow <- function(x){
+     x^n
+   }
+   pow
```

```
+ }
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Explorando una función clausura

¿De que manera puedes saber que es el entorno de una función?

Puedes mirar dentro del entorno en el que se ha definido una función llamando a la función ls sobre el entorno

```
> ls(environment(cube))        # lista de objetiso que contiene la funcion
[1] "n"   "pow"
> get("n",environment(cube)) # el valor de n es 3
[1] 3
> ls(environment(square))
[1] "n"   "pow"
> get("n",environment(square))
[1] 2
```

Alcance léxico vs alcance dinámico

El alcance léxico se emplea en R y el dinámico en otros lenguajes de programación

```
> y <- 10
> f <- function(x){
+   y <- 2
+   y^2 + g(x) # en la función f “y” y la función “g” son variables libre
+ }
> g <- function(x){
+   x^y
+ } # aquí “Y” es una variable libre
> f(3)
[1] 59053
```

Con el alcance léxico el valor de y en la función g se busca en el entorno es el que ha sido definida la función “f” que en este caso es el entorno global. De manera que y en el interior de la función g es 10. Con alcance dinámico el valor de y se busca en el entorno desde el que se ha llamado a la función al que a veces se le llama “entorno de llamada”

En R el entorno de llamada se conoce como “padre padre”

En este caso el valor de y seria 2

Llamar a la función f dará diferentes resultados dependiendo de si se usa alcance léxico o dinamico

Cuando una función es definida en el entorno global y sea llamada desde el entorno global, en este caso el entorno donde se ha definido y el entorno de llamada son exactamente el mismo lo que da la apariencia de usar alcance dinámico incluso cuando este no existe

```
> g2 <- function(x){ # x es un argumento formal
+   a <- 3
+   x+a+w
+ }
> g2(2)
```

```
Error in g2(2) : objeto 'w' no encontrado
> w <- 3
> g2(2) # si es capaz de encontrar w en el entorno global, parece que w ha sido buscado en el entorno de llamada pero en realidad ha sido buscado en el entorno de definicion
[1] 8
```

Otros lenguajes que soportan alcance léxico

- Scheme
- Perl
- Python
- Common lisp (todos los lenguajes convergen en Lisp)

Consecuencias del alcance léxico

Todos los objetos deben estar almacenados en memoria, si trabajas con un lenguaje de programación en el que usas objetos peque;os esto no representa un gran problema. Pero debido a la naturaleza de las reglas de alcance la complejidad del entorno y la forma en laque todos se relación es difícil implementar ese modelo de forma que se extienda mas allá de la memoria física

Cada función tiene un puntero hacia su respectivo entorno de definición y ese entorno de definición puede literalmente estar en cualquier parte porque puede haber funciones definidas en el interior de otras funciones si una función devuelve otra función tiene un puntero donde se almacena el entorno de definición

Reglas de alcance: Ejemplo de optimizacion

¿Porque esta información es útil?

Hay un par de rutinas de optimización en R llamadas “optim”, “nlm” y “optimize” todas requieren que les ases una función cuyo argumento sea un vector de parámetros

Peo en estadística la función objetivo que queremos minimizar o maximizar (por ejemplo una probabilidad logaritmica) depende de otras cosas además de los paametros sobre los que va sa maximizar. Así en particular dependerá de otras cosas como los datos

Cuando hagas este tipo de optimzacion, en muchos caso es útil mantener costantes ciertos parámetros y por ejemplo, fijar un parámetro en un cierto valor luego optimizar sobre el resto de parámetros.

Maximizar una distribución normal:

La idea básica de todo problema de optimización en R es que puedes crear una función de tipo contructor que construya una función objetivo. La idea es que la función objetivo contenga todos los datos, y el resto de las cosas de las que dependa se incluirían en el entorno de definición de esa función, de manera que tendría asociados esas cosas como una especie de equipaje en su entorno de definición. De esta manera no es necesario especificar toda esa serie de cosas cada vez que se llama a la función, solo hace falta especificar el parametro

```
> make.NegLogLik <- function(data, fixed = c(FALSE, FALSE)){
+   params <- fixed
+   function(p){ # p es el vector de parámetros sobre el que voy a optimizar
+     params[!fixed] <- p
+     mu <- params[1]
+     sigma <- params[2]
+     a <- -0.5*length(data)*log(2*pi*sigma^2)
+     b <- -0.5*sum((data-mu)^2) / (sigma^2)
+     -(a+b)
+   }
+ } # este es un constructor que crea una probabilidad logarítmica negativa la
cual queremos minimizar
```

Las funciones mencionadas antes minimizan la función por defecto

```
> set.seed(1)
> normals <- rnorm(100,1,2)
> nLL <- make.NegLogLik(normals)
> nLL
function(p){
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a+b)
  }
<bytecode: 0x000001c07883dcf0>
<environment: 0x000001c075a087c8> # valor hexadecimal
> ls(environment(nLL))
[1] "data"   "fixed"   "params"   # estas son variabes lbres en el interior de
la función de probabilidad logarítmica negativa pero se han definido en el
entorno de definición
```

D

```
> optim(c(mu=0,sigma = 1), nLL)$par
      mu    sigma
1.218239 1.787343
```

Fijando \sigma = 2

```
> nLL <- make.NegLogLik(normals, c(FALSE,2)) # temenos que volver a construir
mi función objetivo llamando a la función y fijando fixed como falso
> optimize(nLL, c(-1,3))$minimum # optimize minimiza funciones de una sola
variable
[1] 1.217775 # algo ligeramente distinto a la optimización anterior
```

Fijando \mu = 1

```
> nLL <- make.NegLogLik(normals, c(1,FALSE))
> optimize(nLL, c(1e-6,10))$minimum
[1] 1.800596
```

Poemos plotear la probabilidad o la probabilidad logarítmica

códigos de ploteo

Resumen:

Puedes construir estas funciones objetivos que contengsn todos los datos necesarios y todo el resto de cosas que son necesarias para evaluar la función dentro del entorno de la función de manera que no tengas que especificar todos los datos
El código puede ser muy simple y impio porque no tienes que usar listas de argumentos largas
copiar referencia

Reglas de alcance: Estándares de codificacion

Los estándares e codigifaicacion hacen al cogido mas legible y permite a otras personas entender lo que hace el código las legible y entendible por otros

- Se debería escribir siempre usando un editor de texto y luego guardarlo en un fichero de texto (un estándar muy básico, no tiene ningún tipo de aspecto en concreto, solo es texto, normalmente esta en texto ASCII)
- Identar el código es la forma de diferencias lineasd e código, 4 caracteres por identacion
- Limitar la anchura del las líneas de código (80 caracteres)
- Limitar la longitud de las funciones para poder agruparlas en piezas lógicas de tu programa

Reglas de alcance: Tiempos y fechas en R

R ha estado desarrollando una especial representación de fechas y tiempos

- Las fechas son representadas por la clase Date
- Las horas están representadas por 2 clases: `POSIXct` y `POSIXlt`
- Las fechas no tienen horas unidas a ellas, representan un día en un a;o y en un mes desde 1970-01-01
- Las horas están almacenadas internamente como el numero de segundos desde el día 1 de enero del 1970

Fechas en R:
Las fechas son representadas por una clase Date y puede coercionarse de un carácter string usando la función as.date()

```
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
> unclass(x)
[1] 0
> unclass(as.Date("1970-01-02"))
[1] 1
```

Las fechas estan almacenadas como objetos de la clase “date”

Horas en R:
Las horas están representadas por 2 clases: `POSIXct` y `POSIXlt` . POSIX es una familia de estándares de computación sobre como deberían hacerse las cosas en cierto tipo de

computadoras o como dberian representarse los datos así que hay una familia de estándares para representar fechas y horas, que es parte del estándar POSIX

- En POSIXct, las horas se representan simplemente como enteros muy largos. Es un tipo de clase útil, por ejemplo, para almacenar horas en un "data frame" o algo así, porque es básicamente como un gran vector de enteros
- POSIXlt almacena una hora como una lista por debajo, y almacena otra mucha información útil sobre una hora dada, por ejemplo cual es el día de la semana de esa horam cual es el día del a;o del día del mes o del propio mes

Hay un numero de funciones genéricas que operan tanto sobre fechas como sobre horas
Weekdays: da el día de la semana
Months: da el nombre del mes
Quarters: da el numero de trimestre ("Q1","Q2","Q3","Q4")

Puedes forzar a cambiar una y otravez entre `POSIXct` y `POSIXlt`

```
> x <- Sys.time()
> x  # hora actual, imprime a;o mes día, horas minutos segundos
[1] "2021-03-10 16:46:44 EST-05"  # EST (Tiempo Estándar del Este)
> p <- as.POSIXlt(x)
> names(unclass(p)) # Podemos mirar los elementos en la lista si los sacas de
la clase
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
 [7] "wday"   "yday"   "isdst"  "zone"   "gmtoff"
> p$sec
[1] 44.54147 # te da los segundos en fracciones de segundos
```

También puedes usar POSIXct

```
> x12 <- Sys.time()
> x12
[1] "2021-03-10 16:58:24 -05"
> unclass(x12)
[1] 1615413504 # numero de segundos desde el 01-01-0970
> x12$sec
Error in x12$sec : $ operator is invalid for atomic vectors
> p <- as.POSIXlt(x12)
> p$sec
[1] 24.05803
```

Hay una función strptime que convierte fechas escritas en formato cadena de caracteres en objetos de fecha y hora

```
> datestring <- c("January 10 2012 10:40", "December 9 2011 9:10")
> x <- strptime(datestring, "%B %d %Y %H:%M")
> x
[1] NA NA
> z <- strptime("20/2/06 11:16:16.683", "%d/%m/%y %H:%M:%OS") # mirar que
significan cada uno con ?strptime
> z
[1] "2006-02-20 11:16:16 -05"
> class(z)
[1] "POSIXlt" "POSIXt"
```

Operaciones con fechas y tiempos

```
> d13 <- as.Date("2012-01-01")  # d13 es un objeto date
> d14 <- strptime("9/1/2011 11:34:21", "%d/%m/%Y %H:%M:%S")
> d14 # d14 es un objeto POSIXlt
[1] "2011-01-09 11:34:21 -05"
> d13-d14
Error in d13 - d14 : argumento no-numérico para operador binario
Además: Warning message:
Métodos incompatibles ("-.Date", "-.POSIXt") para "-"
> d13 <- as.POSIXlt(d13)
> d13-d14
Time difference of 356.3095 days
```

Aveces mantiene la cuenta de a;os bisiestos, segundos bisiestos, horario de verano, zonas horarias

```
> d15 <- as.Date("2012-03-01")
> d16 <- as.Date("2012-02-28")
> d15-d16
Time difference of 2 days
> d17 <- as.POSIXct("2012-10-25 01:00:00")
> d18 <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT") # Tiempo Medio de
Greenwich
> d18-d17
Time difference of 0 secs
```

Funciones de bucle: lapply - sapply

Las funciones que ya conocemos cono for, while son buenas pero son menos compactas en cierto modo. Hay algunas funciones que implementan los siguientes bubles para hacerlo mas fácil.

Algunas funciones son: `lapply, sapply, apply, tapply, mapply`:

- `lapply`: se usa cuando tienes una lista y quieres aplicar cierta función a cada elemento de la lista, es poderoso para hacer cálculos en pocas líneas de código
- `sapply`: es lo mismo que `lapply` pero simplifica los resultados
- `apply`: es una función que opera sobre los márgenes de un array por lo que es útil si quieres calcular resúmenes de matrices u otros arrays multidimensionales
- `tapplt` es una abreviación de table `apply` (applay para tablas) y aplica una función sobre subconjuntos de un vector
- `mapply` es una versión multivariable de `lapply`

también existe ptra función llamada Split que aunque no alica nada sobre los objetos es amenudo útil en conjunto con funciones como lapply o sapply ya que separa objetos en partes

Lapply

Lapply recibe 3 aegumentos: una lista, el nombre de la función y el argumento "…". (se usa para pasar los argumentos de la función que se están aplicando a cada elemento de la lista) Si x no es una lista se tendrá que transformar en una usando as.list(x)

```
> lapply
function (X, FUN, ...)
{
```

```
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
<bytecode: 0x000001db427e7ca8>
<environment: namespace:base>
```

El resto de la función lapply esta implementado eninternamente en C para hacer todo un poco mas rápido

```
> la <- list(a = 1:5, b = rnorm(10))
> lapply(la, mean)
$a
[1] 3

$b
[1] 0.05289394
```

La función va a devover por cada objeto individual en la lista y los valores de retorno se guardaran en una nueva lista

```
> la2 <- list(a = 1:4, b = rnorm(10), c = rnorm(20,1), d = rnorm(100,5))
> lapply(la2, mean)
$a
[1] 2.5

$b
[1] 0.286006

$c
[1] 0.3657849

$d
[1] 4.94664
```

Lista de 4 elementos

```
> la3 <- 1:4
> lapply(la3, runif)
[[1]]
[1] 0.224035

[[2]]
[1] 0.1727623 0.6139190

[[3]]
[1] 0.3058438 0.2278970 0.7823690

[[4]]
[1] 0.8042639 0.4472964 0.2162329 0.8282036
```

En este caso crea la3 numeros aleatorios de la distribución uniforme

```
> la4 <- 1:4
> lapply(la4, runif, min = 0, max = 10)
[[1]]
[1] 6.929043

[[2]]
```

```
[1] 3.072406 5.449254

[[3]]
[1] 6.413817 7.448855 1.976536

[[4]]
[1] 7.129655 9.254009 5.994656 1.460979
```

Min y max son argumentos de runif

```
> la5 <- list(a=matrix(1:4, 2,2), b = matrix(1:6, 3, 2))
> la5
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> lapply(la5, function(elt) elt[,1]) # se puede definir la función ahí mismo y esta función es anónima ya que no tiene nombre
$a
[1] 1 2

$b
[1] 1 2 3
```

Sapply:

- Sapply simplifica el trabajo de lapply si es posible
- Si el resultado es una lista donde todos los elementos son de longitud 1, entonces sapply devolverá un vector de todos estos elementos, sapply simplificara eso en un solo vector
- Si el resultado es una lista dode cada elemento es un vector de la misma longitud sapply pondrá todos los elementos de la lista en una matriz de el numero de misma longitud por el largo de la lista
- Si no pode resolver las cosas devuelve una lista

```
> sapply(la2, mean)
        a         b         c         d
2.5000000 0.2860060 0.3657849 4.9466399
> mean(la2) # no se puede sacar promedio a una lista
[1] NA
Warning message:
In mean.default(la2) : argument is not numeric or logical: returning NA
```

Funciones de bucle: apply

Es usado para evaluar una función sobre los márgenes de un array. Normalmente una función anónima, como ya vimos con lapply o podría ser una función que ya existe como mean.

- Habitualmente se usa para aplicar una función a las filas o columnas de una matriz
- Puede ser usado con arrays en general por ejemplo tomando el promedio de un array de matrices
- No es realmente mas rápido que un bucle for, pero trabaja en una línea

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- X es un array
- MARGIN es un vector entero indicando que márgenes debemos recuperar
- FUN la función ue quieres que se aplique a cada uno de los márgenes
- … argumentos que les quieres pasar a la función

```
> ap <- matrix(rnorm(200), 20, 10)
> apply(ap, 2, mean) # ese 2 es por la 2da dimensión que tiene 10 columnas
 [1]  0.177372298 -0.288661173 -0.144750690  0.057461744 -0.008418646
 [6]  0.168624714  0.213025655  0.202872874 -0.079765651 -0.410540936
> apply(ap, 1, sum) # ese 1 es por la 1ra dimensión que tiene 20 filas
 [1] -1.13583565  3.76615342  1.56438621 -2.76435034 -1.39737315  0.32789030
 [7] -2.81234764 -2.33721856 -4.56618777  0.82656636 -0.08854927  2.44662279
[13]  0.04245915  3.86338734 -0.01968220  2.00311720  2.72414509 -2.42467217
[19]  3.05397093 -5.32807825
```

Col/row sums and means

Para sumas y promedios de dimensiones tenemos algunas funciones especiales

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

Otros caminos para apply

Cuantiles de las filas de una matriz

```
> ap2 <- matrix(rnorm(200), 20,10)
> apply(ap2, 1, quantile, probs = c(0.25, 0.75))  # calculamos los percentiles
25 y 75 de cada fila, el 1 preserva las filas
          [,1]      [,2]       [,3]       [,4]       [,5]       [,6]
25% -0.1738387 -0.135712 -0.7274876 -0.4197203 -0.6741742 -0.9168302
75%  0.2516207  1.093887  0.7647408  0.3435389  0.7955904  0.2983223
          [,7]       [,8]       [,9]      [,10]      [,11]      [,12]
25% -0.8414886 -0.7813656 -0.8404375 -0.4398169 -0.4122411 -0.9604051
75%  0.2947889  0.3636455  0.4199356  1.0324353  0.2528558  0.5173754
         [,13]      [,14]      [,15]      [,16]      [,17]      [,18]
25% -1.2659748 -0.5309255 -0.9371469 -0.2094468 -0.3603879 -0.3420772
75%  0.5875643  0.6701840  0.3496666  0.7195482  1.1628004  0.7112059
         [,19]      [,20]
25% -0.1044477 -0.2630908
75%  0.7866063  0.6215797
```

Promedio de una matrix en un array:

```
> ap3 <- array(rnorm(2 * 2 * 10), c(2, 2, 10)) # 2 filas, 2 columnas y la 3era
dimensión es 10
```

```
> apply(ap3, c(1, 2), mean) # quiere mantener la 1 y la 2 y colapsar la 3era
            [,1]       [,2]
[1,] -0.08469675 0.28535193
[2,]  0.07974993 0.06533332
> rowMeans(ap3, dim=2)
            [,1]       [,2]
[1,] -0.08469675 0.28535193
[2,]  0.07974993 0.06533332
```

A

Funciones de bucle: mapply

Es una función de bucle que es una variable multivariable de las funciones tipo lapply y sapply

La idea es aplicar una función en paralelo sobre un conjunto de diferentes argumentos

Si quisiéramos evaluar 1 funcion a 2 listas lo podriams hacer con un for pero mapply lo pude hacer

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL,
    SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

- FUN: la function a aplicar
- … contiene argumentos para aplicarlo sobre la función
- MoreArgs: es una lista de otros argumentos que pasar a FUN
- SIMPLIFY es similar a los argumentos SIMPLIFY indica si la función debe ser simplificada

```
> list(rep(1,4), rep(2,3), rep(3,2), rep(4,1))
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
# en lugar de escribir lo de arriba usamos mapply
> mapply(rep, 1:4, 4:1) # el primer argumento de rep va de 1 a 4 y el 2do va de
4 a 1
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

Mapply puede usarse para aplicar una función a múltiples conjuntod de argumentos

Vectorizando una función

```
> noise <- function(n,mean,sd){
+   rnorm(n,mean,sd)
+ }
> noise(5, 1, 2)
[1]  0.1951514  2.9955916 -2.7144590 -0.3971137 -0.8547778
> noise(1:5, 1:5, 2)
[1]  1.27321677  1.01177815  5.14327617  9.53899019 -0.02577199
```

La function mapply vectoriza la función noise

```
> mapply(noise,1:5, 1:5, 2)
[[1]]
[1] 0.2517976

[[2]]
[1] 2.6295646 0.9021774

[[3]]
[1] 0.2305939 4.4887530 5.4169770

[[4]]
[1] 2.009722 4.451214 5.408040 1.224897

[[5]]
[1] 2.853435 5.983511 7.110694 2.386783 6.586275
```

Es lo mismo que:

```
> list(rnorm(1,1,2), rnorm(2,2,2), rnorm(3,3,2), rnorm(4,4,2), rnorm(5,5,2))
[[1]]
[1] 3.68321

[[2]]
[1] -0.9677539  1.2816229

[[3]]
[1] 0.8568003 1.0460520 0.5631331

[[4]]
[1] 3.455506 2.067758 4.375750 2.077061

[[5]]
[1]  2.189676  5.083690  2.941338  7.644325 10.028469
```

Funciones de bucle: tapply

Tapply de usa para aplicar una función a subconjuntos de un vector. La idea es que normalmente vas a tener un vector compuesto por números, un vector numérico y habrá partes de ese vector sobre las que quieres calcular algún resumen estadístico

```
> str(tapply)
function (X, INDEX, FUN = NULL,
    ..., default = NA, simplify = TRUE)
```

X es un vector

INDEX es un factor o una lista de factores (o sino son coercionados (convertido) a factores)
FUN es una función para ser aplicado
… contiene otros argumentos que podrían pasarse a FUN
Simplify, si debería ser simplificado el resultado de manera similar a sapply

```
> ta <- c(rnorm(10), runif(10), rnorm(10,1))
> fta <- gl(3,10)
> fta
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(ta,fta,mean)
         1          2          3
0.05013158 0.52876179 0.93380106
```

Tomando grupos son simplificación

```
> tapply(ta,fta,mean,simplify=FALSE)
$`1`
[1] 0.05013158

$`2`
[1] 0.5287618

$`3`
[1] 0.9338011
```

Encontrando rangos de group

```
> tapply(ta, fta, range) # range da el minimo y máximo de las observaciones de
cada subconjunto de ta
$`1`
[1] -1.225861  1.400706

$`2`
[1] 0.1758254 0.8056149

$`3`
[1] -0.2591304  2.1248138
```

A

Funciones de bucle: split

Tapply es útil porque divide un vector en porciones pequelas y aplica una función o resumen estadístico sobre esas porciones
Split no es una función de bucle pero es una función muy practica que se puede usar junto a funciones como lapply o sapply, Split recibe un vector, es como tapply pero sin aplicar el resumen estadístico.

```
> str(split)
function (x, f, drop = FALSE,
    ...)
```

X es un vector o un objeto
F es una variable de tipo factor (o transfromado a uno) o lista de factores
drop indica si los niveles de factor vacio deben eliminarse

```
> sp <- c(rnorm(10), runif(10), rnorm(10,1))
```

```
> fsp <- gl(3,10)
> split(sp, fsp) # divide el vector en 3 partes
$`1`
 [1] -0.82469782  0.66570841 -0.52730559 -0.14518626  0.90675236
 [6] -1.93753829 -0.39836478  0.04744754  0.16117973  0.41698907

$`2`
 [1] 0.03231051 0.22795443 0.72128981 0.16388276 0.20970521 0.39198654
 [7] 0.07326879 0.37686754 0.23978801 0.31729703

$`3`
 [1]  3.072546521 -0.338104054  2.446480781 -0.009490788  2.024151298
 [6]  1.458460579  1.300082403  1.162664300  0.029468447 -0.343146919
```

El idioma comun es Split seguido por un lapply

```
> lapply(split(sp,fsp), mean)
$`1`
[1] -0.1635016

$`2`
[1] 0.2754351

$`3`
[1] 1.080311
```

Puede ser usada para separar tipos de objetos mucho mas complicados

```
> airquality <- read.table("C:/hw1_data.csv",header=TRUE)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Vamos a dividir el dataframe de airquality usando el factor Month

```
> sp2 <- split(airquality, airquality$Month)
> lapply(sp2, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$`5`
   Ozone  Solar.R     Wind
      NA       NA 11.62258

$`6`
    Ozone   Solar.R      Wind
       NA 190.16667  10.26667

$`7`
     Ozone    Solar.R       Wind
        NA 216.483871   8.941935

$`8`
   Ozone  Solar.R     Wind
      NA       NA 8.793548

$`9`
```

```
   Ozone  Solar.R     Wind
      NA 167.4333  10.1800
```

La idea es que sapply simplifique el resultado porque cada elemento de la lista que devuelve tiene un vector de longitud 3, son todos de la misma longitud entonces pondrá todo a una matriz

```
> sapply(sp2, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
               5         6          7        8        9
Ozone         NA        NA         NA       NA       NA
Solar.R       NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667   8.941935 8.793548  10.1800
> sapply(sp2, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
+                                  na.rm=TRUE))
                5         6          7          8         9
Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Separando en mas de un nivel

En el ultimo ejeplo solo tenia una única variable de tipo factor

```
> sp3 <- rnorm(10)
> f1 <- gl(2,5) # 2 niveles
> f2 <- gl(5,2) # 5 niveles
> f1
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1,f2) # conbinacion de niveles
 [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

Las interacciones pueden crear niveles vacios

```
> str(split(sp3,list(f1,f2)))
List of 10
 $ 1.1: num [1:2] 0.688 2.097
 $ 2.1: num(0)
 $ 1.2: num [1:2] 0.972 0.387
 $ 2.2: num(0)
 $ 1.3: num 0.469
 $ 2.3: num 1.11
 $ 1.4: num(0)
 $ 2.4: num [1:2] -0.388 -1.447
 $ 1.5: num(0)
 $ 2.5: num [1:2] -0.147 -0.15
```

Usando drop omotiremos los niveles vacíos creados durante la separación, esto puede ser útil cuando estas combinando varios factores diferentes. Si solo estas usando un factor entonces ese argumento no hace nada, porque simplemente usaras todos los niveles, pero si tienes varios factores normalmente vas a tener niveles vacíos, ya que no hay observaciones de cada combinación individual de los dos factores, entonces drop =TRUEA omitirá esos niveles vacions y entonces te devolverá una lista con solo los niveles para los cuales existen observaciones

```
> str(split(sp3,list(f1,f2), drop=TRUE))
```

```
List of 6
 $ 1.1: num [1:2] 0.688 2.097
 $ 1.2: num [1:2] 0.972 0.387
 $ 1.3: num 0.469
 $ 2.3: num 1.11
 $ 2.4: num [1:2] -0.388 -1.447
 $ 2.5: num [1:2] -0.147 -0.15
```

Herramientas de depuración – Diagnostico del problema

Las herramientas de depuración no vienen en ningún paquete y pueden ser útiles para averiguar cualquier cosa como qué esta mal cuando has descubierto que nay un problema

Indicaciones que algo no es correcto:

Message: aUn mensaje es una notificación muy controlada. Podría ser un mensaje de diagnostico de que algo ha pasado. Pero podría no ser nada. Por lo tanto message no parara la eefcucuion de la función simplemente imprimirá un mensaje

Warning: Un aviso (warning) es otro indicador de que algo esta mal pero no fatal; la ejecución de la función continua pero obtendrás un mensaje al terminar. La función estaba esperando algo y recibió otra cosa

Error: un error es un problema fatal. Estos mensajes de error son generados por la función stop

Condition: es el concepto de mas alto nivel, estos 3 indicadores son condiciones, puedes crear tus propias condiciones. Es un concepto genérico para indicae que algo inesperado puede ocurrir

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : Se han producido NaNs
```

```
> printmessage <- function(x){
+   if(x>0)
+     print("x is greater than zero")
+   else
+     print("x is less than or equal to zero")
+   invisible(x)
+ }
> printmessage(1)
[1] "x is greater than zero"
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print("x is less than
or equal to zero") :
  valor ausente donde TRUE/FALSE es necesario  # NA:0 no es ni V ni F
```

```
> printmessage2 <- function(x){
+   if(is.na(x))
```

```
+     print("x is a missing value!")
+   else if(x>0)
+     print("x is greater than zero")
+   else
+     print("x is less than or equal to zero")
+   invisible(x)
+ }
> x2 <- log(-1)
Warning message:
In log(-1) : Se han producido NaNs
> printmessage2(x2)
[1] "x is a missing value!"
```

Algo esta mal

¿Como se que algo esta mal con una función?

- ¿Cuál fue tu entrada? ¿Como llamaste a la función?
- ¿Que estabas esperando? ¿Salidas, mensajes, otros resultados?
- ¿Qué obtuviste?
- ¿Como difiere lo que obtuviste de lo que esperabas?
- ¿Fue la correcta expectativa en primer lugar?
- ¿Puedes reproducir lel problema exactamente? O sea que el código se reproduzca sin error

Herramientas de depuración – Herramientas básicas

¿Cuáles son las herramientas que pueden utilizar en R para ayudarte a depurar un programa?

Existen 5 funciones básicas y unas pocas asociadas de las que hablaremos

- `Traceback`: imprime la pila de llamadas de la función después de que ocurra un error; no hace nada si no hay error, o sea te dice cuantas llamadas a funciones se han hecho y donde ha currido el error así que puedes identificar en que punto de la secuencia de llamadas a funciones ocurrió el error
- `Debug:` es la mas útil, se le pasa una función como argumento y la marcar para modo depuración. Esto significa que cada vez que ejecutes esa función cada vez que sea llamada por otra función se parara y suspenderá la ejecución de la función en la primera línea y podremos, en lo que se conoce como navegador(browser) ir avanzando paso a paso por la funcion
- `browser:` suspende la ejecución de la función donde sea que llame y pone la función en modo de depuración
- `trace:` permite insertar código de depuración en una función sin llegar a editar la función en si misma. Es útil si estas depurando el código de otra persona. Por ejemplo, si tienes un paquete de código, y no tienes una vía fácil para editar ese código, o tienes código base de R, o algo parecido, y no quieres editar ese código porque no puedes encontrar el archivo o por cualquier otro

motivo, entonces puedes usar la función "trace" para insertar un pequeño trozo de código, y usando ese pequeño trozo de código, llamar al navegador. Y entonces, puedes navegar por esa función, y después desactivar el modo "trace" para que la función vuelva a la normalidad

- `recover`: te permite modificar el comportamiento del error así que tu puedes navegar la pila de llamadas de la función. Es un manejador de errores loque significa que en el momento que encuentres un error, en cualquier parte de una función, en lgar de volver a la consola el interprete de R detendrá la ejecución de la función justo en el punto en el que ha ocurrido el error y le pausara en ese punto. Y a continuación imprimirá la pila de llamadas de la función para qe puedas ver a que profundidad estas y puedas moverte entre las diferentes llamadas a funciones para echar un vistazo. En ese momento, estas en el navegador, y puedes mirar en las distintas llamadas a funciones para saber que ha pasado, como se han tratado los datos y cosas así.

Esta funciones te periten separar los peque;os detalles, las líneas de código e intentar descubrir en que pnto se encuentra exactamente el error de programación. También puedes introducir cosas como sentencias “prnt” y “cat” en el código para imprimir el valor de algunas cosas de manera que si quieres saber cual es el valor de “x” en la línea 42 sencillamente puedes imprimirlo y ver lo que vale. Se puede obtener muchísima información solamente introduciendo sentencias print

Herramientas de depuración – Usando las herramientas

Traceback:

```
> mean(x)
Error in mean(x) : objeto 'x' no encontrado
> traceback() # nos dice donde ocurrió el error y el error ocurre nada mas empezar al principio de la función mean, que no ha llegado a llamar a otras funciones
1: mean(x)
```

Otro ejemlo de traceback

```
> lm(y20 ~ x20)
Error in eval(predvars, data, env) : objeto 'y20' no encontrado
> traceback()
7: eval(predvars, data, env) # en el séptimo fue donde hallo el error, básicamente porque no encontró los valores de y y x. Por lo tanto imprime algo como Error in eval(…) esto puede ser muy útil si estas intentando obtener ayuda de alguien y están intentando depurar tu función juntos
6: eval(predvars, data, env)
5: model.frame.default(formula = y20 ~ x20, drop.unused.levels = TRUE)
4: stats::model.frame(formula = y20 ~ x20, drop.unused.levels = TRUE)
3: eval(mf, parent.frame())
2: eval(mf, parent.frame()) # lm ha llamado a evak sobre lo que se conoce como “model frame”
1: lm(y20 ~ x20)
```

Debug

```
> debug(lm) # realmente no funciona bien en formato estático como este, pero
primero imprime la funcion
> lm(y20 ~ x2)
debugging in: lm(y20 ~ x2)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset",
        "weights", "na.action", "offset"),
        names(mf), 0L)
    mf <- mf[c(1L, m)]
    mf$drop.unused.levels <- TRUE
    mf[[1L]] <- quote(stats::model.frame)
    mf <- eval(mf, parent.frame())
    if (method == "model.frame")
        return(mf)
    else if (method != "qr")
        warning(gettextf("method = '%s' is not supported. Using 'qr'",
            method), domain = NA)
    mt <- attr(mf, "terms")
    y <- model.response(mf, "numeric")
    w <- as.vector(model.weights(mf))
    if (!is.null(w) && !is.numeric(w))
        stop("'weights' must be a numeric vector")
    offset <- model.offset(mf)
    mlm <- is.matrix(y)
    ny <- if (mlm)
        nrow(y)
    else length(y)
    if (!is.null(offset)) {
        if (!mlm)
            offset <- as.vector(offset)
        if (NROW(offset) != ny)
            stop(gettextf("number of offsets is %d, should equal %d (number of
observations)",
                NROW(offset), ny), domain = NA)
    }
    if (is.empty.model(mt)) {
        x <- NULL
        z <- list(coefficients = if (mlm) matrix(NA_real_, 0,
            ncol(y)) else numeric(), residuals = y, fitted.values = 0 *
            y, weights = w, rank = 0L, df.residual = if (!is.null(w)) sum(w !=
            0) else ny)
        if (!is.null(offset)) {
            z$fitted.values <- offset
            z$residuals <- y - offset
        }
    }
    else {
        x <- model.matrix(mt, mf, contrasts)
        z <- if (is.null(w))
            lm.fit(x, y, offset = offset, singular.ok = singular.ok,
```

```
                ...)
        else lm.wfit(x, y, w, offset = offset, singular.ok = singular.ok,
            ...)
    }
    class(z) <- c(if (mlm) "mlm", "lm")
    z$na.action <- attr(mf, "na.action")
    z$offset <- offset
    z$contrasts <- attr(x, "contrasts")
    z$xlevels <- .getXlevels(mt, mf)
    z$call <- cl
    z$terms <- mt
    if (model)
        z$model <- mf
    if (ret.x)
        z$x <- x
    if (ret.y)
        z$y <- y
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]>
# navegador es como tu espacio de trabajo en R, puedes verlo como un espacio de
trabajo embebido dentro de otro espacio de trabajo
```

Luego le damos “n” de next para que ejecute las líneas de código

```
Browse[2]> n
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n # hasta llegar al error y ahora sabes donde se produce el error
debug: m <- match(c("formula", "data", "subset", "weights",
    "na.action", "offset"), names(mf), 0L)
# puedes depurar funciones dentro del depurador, COMPLETAR APUNTE
```

Recover: puedes configurar la función recover como manejador de errores usando la función options diciendo “options(error = recover)”, establecerá una opción global

```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : no se puede abrir la conexión
Además: Warning message:
In file(file, "rt") :
  no fue posible abrir el archivo 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header,
3: file(file, "rt")

Selection: 1
```

```
Called from: top level
Browse[1]> n

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header,
3: file(file, "rt")

Selection: 2
Called from: read.csv("nosuchfile")
Browse[2]>
```

Debugging:

Existen 3 indicadores principals de algún tipo de problema o condición son message (mensaje), warning (aviso), error y de los 3 solo el error pasara a la ejecución de la función

- Cuando estas analizando una función y tiene un problema, asegúrate de que puedes reproducir dicho problema y asegúrate de que puedas expresar que es lo que esperas obtener y como difiere el resultado de lo que esperabas o mejor dicho, lo que estas esperando
- Puedes usar las herramientas interactivas “traceback”, debug, browser, trace y recover
- Las herramientas de depuración no son un sustituto de pensar, siempre debes oensar a la hra de escribir el código

La función “str”

La función puede ayudarte a ver objetos en R

Str: muestra de fora compacta la estructura interna de un objeto en R, puedes imaginarte que str significa estructura, puedes mirar un objeto y saber que es y que tiene por dentro

Una función de diagnostico muy simple y muy versatil se puede usar como una alternativa a “summary”

Es especialmente bueno adecuado para mostrar de forma compacta el contenido de listas largas que pueden contener listas anidadas.

Su objetivo es producir mas o menos una línea por objeto básico en la salida, por ejemeplo si tenemos un objeto simple como un vector obtendremos una línea de salida que se imprima en la consola. Así que el objetivo básico de “str” es responder a la pregunta ¿Qué hay en esete objeto?

```
> str(str)
function (object, ...)
```

puedes ver que es una función que toma un objeto

```
> str(lm)
function (formula, data, subset, weights,
    na.action, method = "qr", model = TRUE,
    x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)  # lo que da son los argumentos de la
función lm, se puede ver un resumen muy breve
```

```
> str(ls)
function (name, pos = -1L, envir = as.environment(pos),
    all.names = FALSE, pattern, sorted = TRUE)
```

```
> xst <- rnorm(100,2, 4)
> summary(xst)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -7.403  -1.195   1.131   1.580   4.306  11.096
> str(xst)
 num [1:100] 0.414 1.998 -5.373 1.098 -1.196 ... # d una salida de una línea y
dice que xst es un vector numérico donde hay 100 elementos
```

veamos para un factor

```
> f <- gl(40, 10)
> str(f)
 Factor w/ 40 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
> summary(f) # da el numero de elementos en cada uno de los 40 niveles
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

Veamos con dataframes

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> str(airquality)
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

La salida de str es muy útil para hacer un examen rápido de los datos que podrías tener en R y cual es la estructura de los diferentes objetos R

```
> m4 <- matrix(rnorm(100), 10, 10)
> str(m4)
 num [1:10, 1:10] -0.778 -0.861 -4.511 0.729 1.518 ...
> m4[, 1]
 [1] -0.7778514 -0.8614837 -4.5113672  0.7294645  1.5175892  1.1929203
 [7]  2.3849894  0.7312586 -0.6561582  0.5884006
> sm <- split(airquality, airquality$Month)
> str(sm) # una lista de que contiene 5 data frames, donde cada uno representa
un mes dado
List of 5
 $ 5:'data.frame':  31 obs. of  6 variables:  # representa mes de mayo
  ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
  ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
```

```
 ..$ Wind    : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 ..$ Temp    : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
 ..$ Month   : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
 ..$ Day     : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
$ 6:'data.frame':  30 obs. of  6 variables:
 ..$ Ozone   : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
 ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
 ..$ Wind    : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
 ..$ Temp    : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
 ..$ Month   : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
 ..$ Day     : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
$ 7:'data.frame':  31 obs. of  6 variables:
 ..$ Ozone   : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
 ..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 272 175 ...
 ..$ Wind    : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 5.7 7.4 ...
 ..$ Temp    : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
 ..$ Month   : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
 ..$ Day     : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
$ 8:'data.frame':  31 obs. of  6 variables:
 ..$ Ozone   : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
 ..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
 ..$ Wind    : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 8.6 ...
 ..$ Temp    : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
 ..$ Month   : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
 ..$ Day     : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
$ 9:'data.frame':  30 obs. of  6 variables:
 ..$ Ozone   : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
 ..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
 ..$ Wind    : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 10.9 9.7 ...
 ..$ Temp    : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
 ..$ Month   : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
 ..$ Day     : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

Simulación – Generando números aleatorios

Hay un ar de funciones disponibles para simular números o variables de determinadas distribuciones de probabilidad, y ytal vez la mas importante de ellas es la distribución normal
Rnorm: simula valores aleatorios de una distribucion normal con determinada media y desviación estándar

Rpois: genera variables aleatorias de una distribución de poisson con una determinada frecuencia
Por cada una de las funciones de distribución de probabilidad existen 4 funciones asociadas con ellas. Y entonces para cada distribución dada habrá una función que empieza con d, r, p, q

- d evalua la densidad de probabilidad

- r de generación de números random
- p evalua la distribución acumulada
- q de la función cuantil

Trabajando con la distribución normal requerimos usar estas cuatro funciones:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Generando algunos valores aleatorios

```
> grn <- rnorm(10)
> grn
 [1]  0.1952090 -0.8374288 -0.9453173 -0.5995490 -1.2646903 -1.5603328
 [7]  1.4554720  1.1257785  0.4652329  1.0970684
> grn2 <- rnorm(10,20,2) # mean = 20, sd = 2
> grn2
 [1] 20.00374 17.41089 17.75479 21.48026 14.79548 19.56927 18.49514 19.59742
 [9] 18.47184 18.47047
> summary(grn2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.80   17.93   18.48   18.60   19.59   21.48
```

Es importante que fijemos el valor de la semilla en el generador de números aleatorios y esto se puede hacer con set.seed, la semilla es cualquier numero entero

Si queremos generar el nmismo conjunto de datos aleatorios nuevamente es posible con set.seed

```
> rnorm(5)
[1]  1.5117812  0.3898432 -0.6212406 -2.2146999  1.1249309
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078  # a
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078  # a
```

Con la function de poisson

```
> rpois(10, 1)  # frecuencia = 1 (o lambda)
 [1] 0 0 1 1 2 1 1 4 1 2
> rpois(10, 2)
 [1] 4 1 2 0 1 1 0 1 4 1
> rpois(10, 20) # distribucion acumulada
 [1] 19 19 24 23 22 24 23 20 11 22
> ppois(2, 2) # Pr(x<=2)
[1] 0.6766764
> ppois(4, 2) # Pr(x<=4)
[1] 0.947347
> ppois(6, 2) # Pr(x<=6)
[1] 0.9954662
```

F

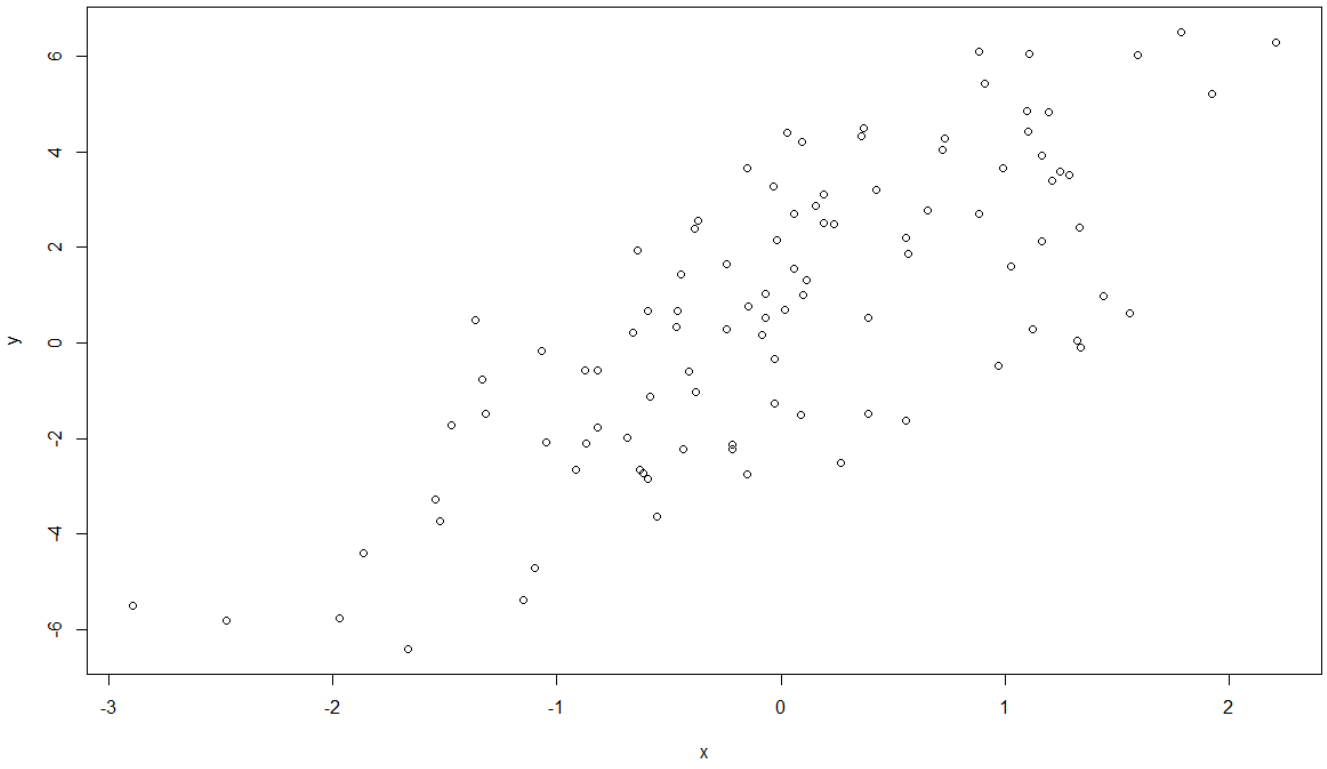
Simulación – Simulando un modelo lineal

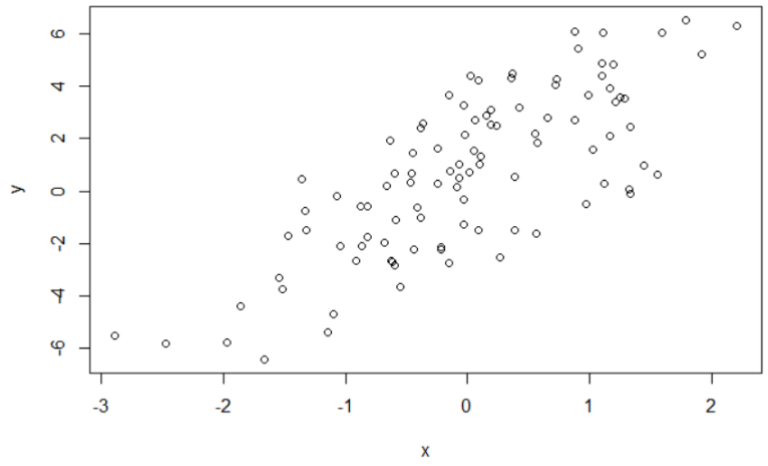
Supongamos que queremos modelar el siguiente modelo lineal

$$y = \beta_0 + \beta_1 x + e$$

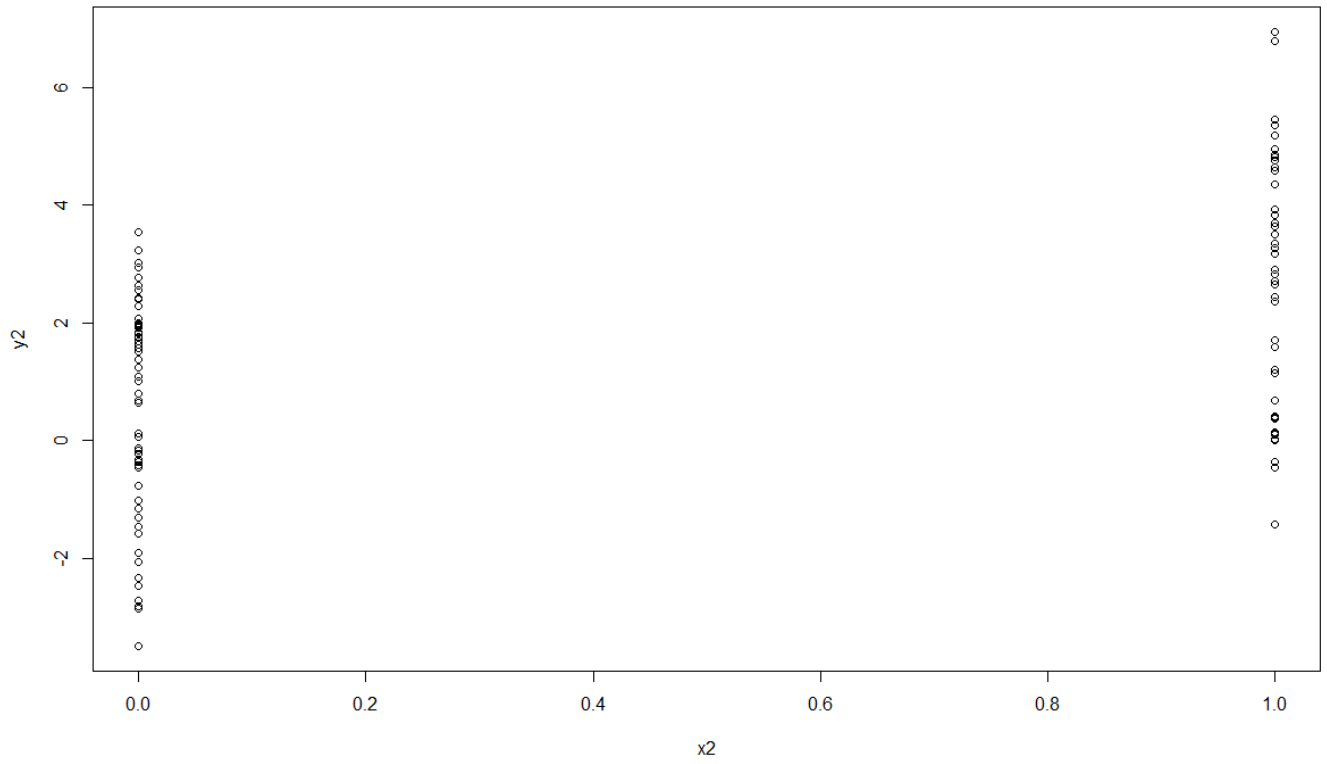
Donde $e \sim (0,2^2)$. Asumimos $x \sim N(0,1^2), \beta_0 = 0.5$ y $\beta_1 = 2$

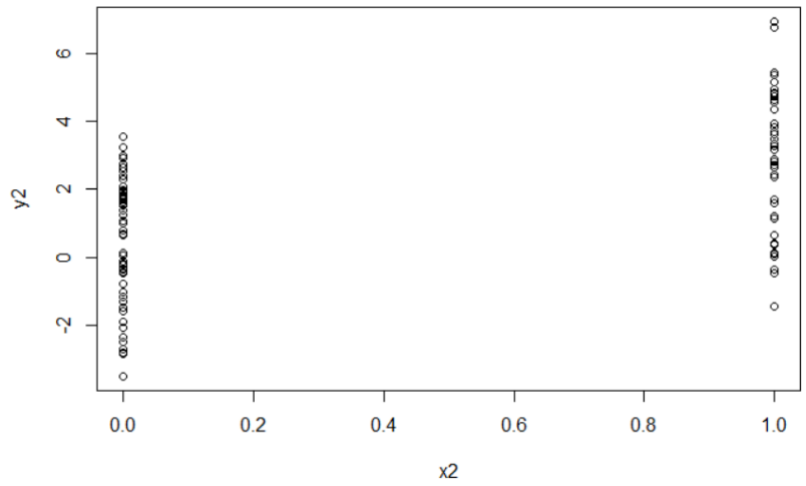
```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2) # e es como un ruido aleatorio
> y <- 0.5 + 2*x + e
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-6.4084 -1.5402  0.6789  0.6893  2.9303  6.5052
> plot(x,y)
```






```
> set.seed(10)
> x2 <- rbinom(100, 1 , 0.5) · la probabilidad de que sea 1 es 0.5
> e2 <- rnorm(100, 0, 2)
> y2 <- 0.5 +2*x2 + e2
> summary(y2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.4936 -0.1409  1.5767  1.4322  2.8397  6.9410
```

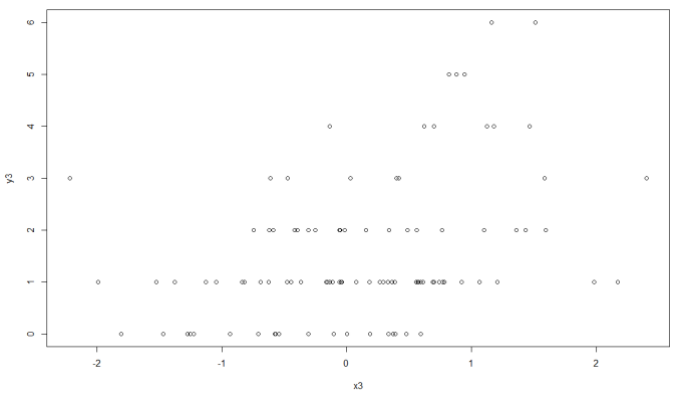


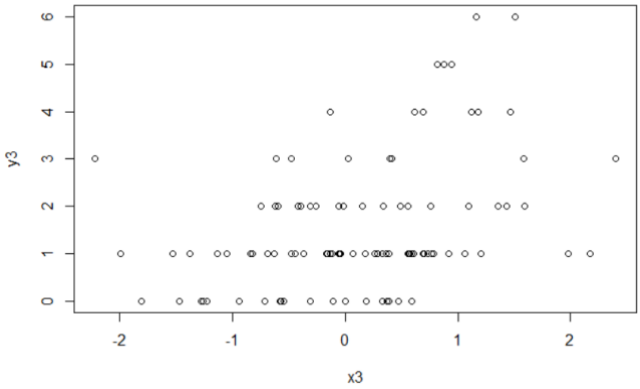


Supongamos que queremos simular desde una distribución de Poisson donde Y~Posson(\mu)

$Log\ \mu = \beta_0 + \beta_1 x$ y $\beta_0 = 0.5\ y\ \beta_1 = 0.3$. Necesitamos usar la función rpois para esto

```
> set.seed(1)
> x3 <- rnorm(100)
> log.mu <- 0.5 + 0.3*x3
> y3 <- rpois(100, exp(log.mu))
> summary(y3)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    1.00    1.00    1.55    2.00    6.00
> plot(x3, y3) # a medida que x3 aumenta, el recuento para y3 generalmente se
hace mas grande
```





Simulación – Muestreo aleatorio

La función sample (muestreo) permite sacar aleatoriamente un elemento de un conjunto de objetos que especifiques. De este modo, si le indicas un vector de números, esta te permite tomar una muestra aleatoria de ese vector de números Así, puedes crear una distribución arbitraria que quieras, especificando un vector de objetos y luego tomando muestras de este.

```
> set.seed(1)
> sample(1:10,4)
[1] 3 4 5 7
> sample(1:10,4) # numeros distintos porque es sin reposicion
[1] 3 9 8 5
> sample(letters,5)
[1] "q" "b" "e" "x" "p"
> sample(1:10) # permutation
 [1]  4  7 10  6  9  2  8  3  1  5
> sample(1:10)
 [1]  2  3  4  1  9  5 10  8  6  7
> sample(1:10, replace = TRUE) # sample w/replacement, pueden tomar valores
iguales porque es con reposicion
 [1] 2 9 7 8 2 8 5 9 7 8
```

Resumen de la simulación:

- Puedes sacar muestras aleatorias de distribución de probabilidad especificas con las funciones de R como rnorm, rpois, rbinom
- Las distribuciones estándares están integradas: normal, poisson, binomial, gamma, etc
- La función sample puede ser usada para tomar muestras aleatorias de vectores arbitrarios
- Es importante recordar establecer la semilla del generador de numero aleatorio cada vez que simules data en R, de esta manera puedes volver a reproducir los datos que obtuviste

R profiler - Parte 1

El “profiler” en R es una herramienta muy practica para aquellas ocasiones en las que estas desarrollando programas grandes o haciendo análisis de big data y como consecuencia, estas corriendo código en R que esta tomando bastante tiempo, mas

tiempo del que te gustaría tener qu esperar. Por supuesto todo esto es relativo de la aplicación que estas usando

El profiles esuna herameinta muy practica para determinar la razón por la cual s esta demorando y para sugerir estrateias para arreglar el problema

¿Esta mi código corriendo lentamente?

- Hacerle profiling al programa es una forma de examinar la cantidad de tiempo que esta siendo gastada en parres distintas del programa
- Es particularmente útil cuando estas tratando de optimizar tu código, o cuando estas intentando mejorar la eficiente de algún programa
- A menudo el código correo bien una vez, pero ¿cuando entramos a un bucle de 1000 iteraciones? ¿Esto es suficientemente rápido?
- Profiling is better tan guessing

Sobre la optimizacion de tu código

Obtener mayor impato en gatar el código depende de saber donde esta gastando el tiempo

Esto es muy difícil o enalguna ocasiones casi imposible de hacer sin la ayuda de análisis formal de rendimiento o “profiling”

we should forget about small efficient , say about 97% of the time: premature optimization is the root of all evil

deberíamos olvidarnos de los pequeños eficientes, digamos alrededor del 97% del tiempo: la optimización prematura es la raíz de todos los males

Principios generales e optimización:

- Dise;ar primero, luego optimizar
- Recuerda: la optimización es la raíz de todos los males. Si intentas optimizar desde e principio, muy seguramente introduciras errores antes incluso de que puedas cotar con un programa que funcione correctamente
- Mide (colecta datos) no adivines, recolecta datos para analizarla. El proceso de recolectar la información es lo que conoce como “profiling”
- Si vas a ser un científico, necesitas aplicar el mismo principio en la que lo harías en otro contexto

Usando system.time():

- Es una función que hace tomar una expresión arbitraria en R y evaluarla para decirte la cantidad de tiempo que se demoro en evaluar esa expresión. Ahora la expresión puede ser algo muy sencillo tal como un llamado a función o algo muy complicado si lo encieras entre corchetes. Básicamente te regresa el tiempo en segundos que fue necesario para ejecutar dicha expresión. Si existe un error de código mientras la expresión estaba siendo evaluada entonces obtendrás el tiempo hasta antes de que el error se presente
- Ahora hay 2 tipos de nociones importantes del tiempo cuando estas ejecutando expresiones en el computador
    - Usertime: es la cantidad de tiempo que es cargada en el CPU o las CPUs, con el fin de ejecutar esta expresión, es aproximadamente la cantidad de tiempo que es usada por el computador

    - Elapsed time: es la cantidad de tiempo que tu como usuario experimentas. Hay que fijarnos que cuando tu eres el usuario tu no eres el tiempo de usuario, tu realmente experimentas el tiempo transcurrido
- Usualmente el user time y elapsed so relativamente cercanos debido a que la cantidad de tiempo que el computador gasta en ejecutar tu función o programa es aproximadamente igual a la cantidad de tiempo que tú esperas por el, sin embargo esto aplica a tareas de computación estándar
- Habrá ocasiones en las que elapsed time será mayor que el user time si el CPU demora mucho tiempo esperando. Esto pasa si tu demoras mas tiempo esperando que lo que el comptdor realmente gasta en la ejecución de tu código. La razón es que el computador puede gastar mucho tiempo esperando a que otras cosas sucedan, cosas que son externas y no relacionadas a tu programa como tal
- Habrá ocasiones en las que elapsed time será menor que el user time si tu maquina tiene múltiples procesadores/cores (y es capaz de usarlos a la misma vez). Este es el caso de la mayoría de computadoras de hoy en día las cuales tienen 2, 4 o mas cores
- Multi – threaded BLAS libraries (vecLib / Accelerate, ATLAS, ACML, MKL)
- Librerias de procesamiento paralelo: el paquete parallel que puede usar varios cores y también varioas computadores

C

```
> system.time(readLines("http://www.jhsph.edu"))
   user  system elapsed  # el cpu no gasta tiempo ejecutando este programa
porque la mayoría del tiempo es gastado es esperando hasta que la información
sea transmitida a través de la red
   0.19    0.09    2.76
>
> hilbert <- function(n){
+   i <- 1:n
+   1/outer(i - 1, i, "+")
+ }
> hi <- hilbert(1000)
> system.time(svd(hi))
   user  system elapsed
   4.50    0.02    4.81
```

Cronometrando expresiones mas largas

```
> system.time({
+   n <- 1000
+   r <- numeric(n)
+   for (i in 1:n){
+     x <- rnorm(n)
+     r[i] <- mean(x)
+   }
+ })
   user  system elapsed
   0.12    0.00    0.14
```

Mas alla de system.time()

- Esta es una función muy útil si deseas tomar un programa peque;o y averiguar que tanto tiempo se dmeora en ser ejecutado, o en ocasiones en la que tienes un programa y quieres estudiarlo línea por línea para averiguar cules de ellas están gastando la mayoría del tiempo
- El problema se sustem.time() es que asume que tu sabes en donde buscar o donde esta el problema y que eres capaz de llamar la función system.time ara la expresión respectiva
- Que hago si no engo idea de donde puedan estar los problemas o donde empezar a buscar?

S

R profiler – Parte 2

La función Rprof arranca el analizador en R

- R debe ser compilado con el soporte del analizador dado que esto no esta icluido en todos los casos pero puedo afirmar que es ciero en el 99.9% de los casos. Lo mas probable es que tu versión de R pueda usar profiler o analizador
- Otra función útil es summaryRprof que toma los resultados del analizador y los resume de manera fácil de entender, porque la salida original del organizador generalmente no se puede usar
- Es importante entender que no se debe usar la función system.time al tiempo con la función Rprof porque no están diseñadas para trabajar juntas. Se debe usar una u otra pero no ambas
- Lo que hace Rprof es un seguimiento en intervalos regulares a la pila de llamadas a funciones. Básicamente, mientras que la función se ejecuta Rprof consulta a la pila de llamadas a funciones, es decir , las funciones que llaman a otras funciones que a su vez llaman a otras funciones he imprime eso en pantalla., las llamadas a las funciones con intervalos de tiempo muy rápidos, cada 0.02 segundos.
- Notamos que la función se toma menos de 0.02 segundos en ejecutarse, entonces este profile (analizador) no sirve para nada porque nunca alcanza a consultar la pila de llamadas a funciones. En general si tu programa se ejecuta muy rápido, el nalizador no es muy utily de todos nodos no hace falta el analizador dado que el código no necesita mejoras

Código
Usando sumaryRprof():
Ya que los resultados en crudo no son particularmente fáciles de leer usaos Rprof
Hay 2 metodos para normalizar los datos:

- By.total: divide el tiempo invertido en cada función por el tiempo eejecucion total
- Bt.self que hace los mismo pero resta el tiempo empleado en las funciones precedentes en de la pila de llamadas

Normalizando by.tiotal y by.self:
La idea es que el normalizador respecto al tiempo total empleado en ejecutar la función,

ResumenÑ

- Rprof es una herramienta muy útil para evaluar el desempeño del código en R y obtener información muy útil.
- La función summaryRprof() resume la salida de Rprof y nos da el porcentaje de tiempo que pasamos en cada función (con 2 tipos de normalización. la qe es útil para descubrir cuellos de botella e nuestro código es by.self)
- Es bueno para sibdividir nuestro código en funciones de manera que en lufar de tener una función enorme es útil dividir nuestro código en partes lógicas e imlmementaras en diferentes funciones
- C y Fortran no tienen para hacer profiler

Dgb

Swirl 1: Basic building blocks

Bg sbg

```
// swirl tarea 1
Selection: 1

  |                                                                      |   0%

| In this lesson, we will explore some basic building blocks of the R
| programming language.

...

  |==                                                                    |   3%
| If at any point you wouldd like more information on a particular topic relate
d to
| R, you can type help.start() at the prompt, which will open a menu of
| resources (either within RStudio or your default web browser, depending on
| your setup). Alternatively, a simple web search often yields the answer
| you are looking for.

...

  |====                                                                  |   5%
| In its simplest form, R can be used as an interactive calculator. Type 5 + 7
| and press Enter.

> 5+7
[1] 12

| You are the best!

  |=====                                                                 |   8%
| R simply prints the result of 12 by default. However, R is a programming
| language and often the reason we use a programming language as opposed to a
| calculator is to automate some process or avoid unnecessary repetition.

...

  |=======                                                               |  11%
| In this case, we may want to use our result from above in a second
| calculation. Instead of retyping 5 + 7 every time we need it, we can just
| create a new variable that stores the result.

...

  |=========                                                             |  13%
| The way you assign a value to a variable in R is by using the assignment
| operator, which is just a 'less than' symbol followed by a 'minus' sign. It
```

```
| looks like this: <-

...

  |===========                                                     |  16%
| Think of the assignment operator as an arrow. You are assigning the value on
| the right side of the arrow to the variable name on the left side of the
| arrow.

...

  |=============                                                   |  18%
| To assign the result of 5 + 7 to a new variable called x, you type x <- 5 +
| 7. This can be read as 'x gets 5 plus 7'. Give it a try now.

> x <- 5+7

| Excellent work!

  |===============                                                 |  21%
| You will notice that R did not print the result of 12 this time. When you use
| the assignment operator, R assumes that you don t want to see the result
| immediately, but rather that you intend to use the result for something else
| later on.

...

  |=================                                               |  24%
| To view the contents of the variable x, just type x and press Enter. Try it
| now.

> x
[1] 12

| Keep working like that and you will get there!

  |===================                                             |  26%
| Now, store the result of x - 3 in a new variable called y.

> y <- x-3

| You are really on a roll!

  |=====================                                           |  29%
| What is the value of y? Type y to find out.

> y
[1] 9

| That is a job well done!

  |=======================                                         |  32%
| Now, let is create a small collection of numbers called a vector. Any object
| that contains data is called a data structure and numeric vectors are the
```

```
| simplest type of data structure in R. In fact, even a single number is
| considered a vector of length one.

...

  |=======================                                               |  34%
| The easiest way to create a vector is with the c() function, which stands
| for 'concatenate' or 'combine'. To create a vector containing the numbers
| 1.1, 9, and 3.14, type c(1.1, 9, 3.14). Try it now and store the result in a
| variable called z.

> z <- c(1.1,9,3.14)

| You are doing so well!

  |========================                                              |  37%
| Anytime you have questions about a particular function, you can access R is
| built-in help files via the `?` command. For example, if you want more
| information on the c() function, type ?c without the parentheses that
| normally follow a function name. Give it a try.

> ?c

| You got it!

  |=========================                                             |  39%
| Type z to view its contents. Notice that there are no commas separating the
| values in the output.

> z
[1] 1.10 9.00 3.14

| All that practice is paying off!

  |==========================                                            |  42%
| You can combine vectors to make a new vector. Create a new vector that
| contains z, 555, then z again in that order. Don t assign this vector to a
| new variable, so that we can just see the result immediately.

> c(z,555,z)
[1]   1.10   9.00   3.14 555.00   1.10   9.00   3.14

| You got it!

  |============================                                          |  45%
| Numeric vectors can be used in arithmetic expressions. Type the following to
| see what happens: z * 2 + 100.

> z*2+100
[1] 102.20 118.00 106.28

| Excellent work!

  |=============================                                         |  47%
```

```
| First, R multiplied each of the three elements in z by 2. Then it added 100
| to each element to get the result you see above.

...

  |=================================                                     |  50%
| Other common arithmetic operators are `+`, `-`, `/`, and `^` (where x^2
| means 'x squared'). To take the square root, use the sqrt() function and to
| take the absolute value, use the abs() function.

...

  |==================================                                    |  53%
| Take the square root of z - 1 and assign it to a new variable called
| my_sqrt.

> my_sqrt <- sqrt(z-1)

| Perseverance, that is the answer.

  |===================================                                   |  55%
| Before we view the contents of the my_sqrt variable, what do you think it
| contains?

1: a vector of length 0 (i.e. an empty vector)
2: a vector of length 3
3: a single number (i.e a vector of length 1)

Selection: 2

| Your dedication is inspiring!

  |====================================                                  |  58%
| Print the contents of my_sqrt.

> my_sqrt
[1] 0.3162278 2.8284271 1.4628739

| Nice work!

  |=====================================                                 |  61%
| As you may have guessed, R first subtracted 1 from each element of z, then
| took the square root of each element. This leaves you with a vector of the
| same length as the original vector z.

...

  |======================================                                |  63%
| Now, create a new variable called my_div that gets the value of z divided by
| my_sqrt.

> my_div <- z/my_sqrt

| That is the answer I was looking for.
```

```
  |=============================================                         |  66%
| Which statement do you think is true?

1: my_div is undefined
2: my_div is a single number (i.e a vector of length 1)
3: The first element of my_div is equal to the first element of z divided by th
e first element of my_sqrt, and so on...

Selection: 3

| You are the best!

  |==============================================                        |  68%
| Go ahead and print the contents of my_div.

> my_div
[1] 3.478505 3.181981 2.146460

| Excellent job!

  |===============================================                       |  71%
| When given two vectors of the same length, R simply performs the specified
| arithmetic operation (`+`, `-`, `*`, etc.) element-by-element. If the
| vectors are of different lengths, R recycles the shorter vector until it
| is the same length as the longer vector.

...

  |================================================                      |  74%
| When we did z * 2 + 100 in our earlier example, z was a vector of length 3,
| but technically 2 and 100 are each vectors of length 1.

...

  |=================================================                     |  76%
| Behind the scenes, R is recycling the 2 to make a vector of 2s and the 100
| to make a vector of 100s. In other words, when you ask R to compute z * 2 +
| 100, what it really computes is this: z * c(2, 2, 2) + c(100, 100, 100).

...

  |==================================================                    |  79%
| To see another example of how this vector "recycling" works, try adding c(1,
| 2, 3, 4) and c(0, 10). Don t worry about saving the result in a new
| variable.

> c(1,2,3,4) + c(0,10)
[1]  1 12  3 14

| That is the answer I was looking for.

  |===================================================                   |  82%
| If the length of the shorter vector does not divide evenly into the length
```

| of the longer vector, R will still apply the 'recycling' method, but will
| throw a warning to let you know something fishy might be going on.

...

  |=== | 84%
| Try c(1, 2, 3, 4) + c(0, 10, 100) for an example.

> c(1,2,3,4) +c(0,10,100)
[1]   1 12 103   4
Warning message:
In c(1, 2, 3, 4) + c(0, 10, 100) :
  longitud de objeto mayor no es múltiplo de la longitud de uno menor

| Nice work!

  |=== | 87%
| Before concluding this lesson, I wouldd like to show you a couple of time-
saving
| tricks.

...

  |=== | 89%
| Earlier in the lesson, you computed z * 2 + 100. Let s pretend that you made
| a mistake and that you meant to add 1000 instead of 100. You could either
| re-type the expression, or...

...

  |===| 92%
| In many programming environments, the up arrow will cycle through previous
| commands. Try hitting the up arrow on your keyboard until you get to this
| command (z * 2 + 100), then change 100 to 1000 and hit Enter. If the up
| arrow doesn t work for you, just type the corrected command.

> z*2+1000
[1] 1002.20 1018.00 1006.28

| You are doing so well!

  |===| 95%
| Finally, let s pretend you d like to view the contents of a variable that
| you created earlier, but you can t seem to remember if you named it my_div
| or myDiv. You could try both and see what works, or...

...

  |===| 97%
| You can type the first two letters of the variable name, then hit the Tab
| key (possibly more than once). Most programming environments will provide a
| list of variables that you have created that begin with 'my'. This is called
| **auto**-completion and can be quite handy when you have many variables in your
| workspace. Give it a try. (If **auto**-completion doesn t work for you, just

```
| type my_div and press Enter.)

> my_div
[1] 3.478505 3.181981 2.146460

| Your dedication is inspiring!

  |======================================================================| 100%
| Would you like to receive credit for completing this course on Coursera.org?

1: Yes
2: No
```

Gbr b

Swirl 2: Workspace and files

Fvdva

```
Selection: 2

  |                                                                      |   0%

| In this lesson, you will learn how to examine your local workspace in R and
| begin to explore the relationship between your workspace and the file system
| of your machine.

...

  |==                                                                    |   3%
| Because different operating systems have different conventions with regards
| to things like file paths, the outputs of these commands may vary across
| machines.

...

  |====                                                                  |   5%
| However it is important to note that R provides a common API (a common set of
| commands) for interacting with files, that way your code will work across
| different kinds of computers.

...

  |=====                                                                 |   8%
| Let is jump right in so you can get a feel for how these special functions
| work!

...

  |=======                                                               |  10%
| Determine which directory your R session is using as its current working
| directory using getwd().

> getwd()
[1] "C:/Coursera/Programming with R/RProgramming"
```

```
| Great job!

  |=========                                                             |  13%
| List all the objects in your local workspace using ls().

> ls()
 [1] "a"       "b"       "f"       "f2"      "m"       "m2"      "m3"
 [8] "ma"      "my_div"  "my_sqrt" "na"      "na2"     "na3"     "v"
[15] "v2"      "x"       "y"       "z"

| You got it!

  |===========                                                           |  15%
| Some R commands are the same as their equivalents commands on Linux or on a
| Mac. Both Linux and Mac operating systems are based on an operating system
| called Unix. It is always a good idea to learn more about Unix!

...

  |============                                                          |  18%
| Assign 9 to x using x <- 9.

> x <- 9

| You are the best!

  |==============                                                        |  21%
| Now take a look at objects that are in your workspace using ls().

> ls()
 [1] "a"       "b"       "f"       "f2"      "m"       "m2"      "m3"
 [8] "ma"      "my_div"  "my_sqrt" "na"      "na2"     "na3"     "v"
[15] "v2"      "x"       "y"       "z"

| All that practice is paying off!

  |================                                                      |  23%
| List all the files in your working directory using list.files() or dir().

> dir()
[1] "RProgramming.Rproj" "Semana1.R"

| You are amazing!

  |==================                                                    |  26%
| As we go through this lesson, you should be examining the help page for each
| new function. Check out the help page for list.files with the command
| ?list.files.

> ?list.files

| Nice work!
```

```
  |==================                                                    |  28%
| One of the most helpful parts of any R help file is the See Also section.
| Read that section for list.files. Some of these functions may be used in
| later portions of this lesson.

...

  |=====================                                                 |  31%
| Using the args() function on a function name is also a handy way to see what
| arguments a function can take.

...

  |======================                                                |  33%
| Use the args() function to determine the arguments to list.files().

> args()
Error in args() : el argumento "name" está ausente, sin valor por omisión
> args(list.files())
NULL

| Keep trying! Or, type info() for more options.

| Type args(list.files) to see the arguments to list.files.

> args(list.files
+ )
function (path = ".", pattern = NULL, all.files = FALSE,
    full.names = FALSE, recursive = FALSE, ignore.case = FALSE,
    include.dirs = FALSE, no.. = FALSE)
NULL

| That is the answer I was looking for.

  |========================                                              |  36%
| Assign the value of the current working directory to a variable called
| "old.dir".

> old.dir <- getwd()

| Keep working like that and you will get there!

  |==========================                                            |  38%
| We will use old.dir at the end of this lesson to move back to the place that
| we started. A lot of query functions like getwd() have the useful property
| that they return the answer to the question as a result of the function.

...

  |===========================                                           |  41%
| Use dir.create() to create a directory in the current working directory
| called "testdir".

> dir.create(testdir)
```

```
Error in dir.create(testdir) : objeto 'testdir' no encontrado
> dir.create("testdir")

| That is correct!

  |=============================                                         |  44%
| We will do all our work in this new directory and then delete it after we
| are done. This is the R analog to "Take only pictures, leave only
| footprints."

...

  |==============================                                        |  46%
| Set your working directory to "testdir" with the setwd() command.

> setwd()
Error in setwd() : el argumento "dir" está ausente, sin valor por omisión
> setwd(testdir)
Error in setwd(testdir) : objeto 'testdir' no encontrado
> setwd("testdir")

| Keep up the great work!

  |===================================                                   |  49%
| In general, you will want your working directory to be someplace sensible,
| perhaps created for the specific project that you are working on. In fact,
| organizing your work in R packages using RStudio is an excellent option.
| Check out RStudio at http://www.rstudio.com/

...

  |====================================                                  |  51%
| Create a file in your working directory called "mytest.R" using the
| file.create() function.

> file.create("mytest.R")
[1] TRUE

| That is the answer I was looking for.

  |=====================================                                 |  54%
| This should be the only file in this newly created directory. Let is check
| this by listing all the files in the current directory.

> ls()
 [1] "a"       "b"       "f"       "f2"      "m"       "m2"      "m3"
 [8] "ma"      "my_div"  "my_sqrt" "na"      "na2"     "na3"     "old.dir"
[15] "v"       "v2"      "x"       "y"       "z"

| Try again. Getting it right on the first try is boring anyway! Or, type
| info() for more options.

| list.files() shows that the directory only contains mytest.R.
```

```
> list.files()
[1] "mytest.R"

| Perseverance, that is the answer.

  |=======================================                               |  56%
| Check to see if "mytest.R" exists in the working directory using the
| file.exists() function.

> file.exists("mytest.R")
[1] TRUE

| Great job!

  |=========================================                             |  59%
| These sorts of functions are excessive for interactive use. But, if you are
| running a program that loops through a series of files and does some
| processing on each one, you will want to check to see that each exists
| before you try to process it.

...

  |===========================================                           |  62%
| Access information about the file "mytest.R" by using file.info().

> file.info("mytest.R")
         size isdir mode               mtime               ctime
mytest.R    0 FALSE  666 2021-03-04 19:55:03 2021-03-04 19:55:03
                       atime exe
mytest.R 2021-03-04 19:55:03  no

| That is the answer I was looking for.

  |=============================================                         |  64%
| You can use the $ operator --- e.g., file.info("mytest.R")$mode --- to grab
| specific items.

...

  |===============================================                       |  67%
| Change the name of the file "mytest.R" to "mytest2.R" by using
| file.rename().

> file.rename("mytest.R","mytest2.R")
[1] TRUE

| All that hard work is paying off!

  |=================================================                     |  69%
| Your operating system will provide simpler tools for these sorts of tasks,
| but having the ability to manipulate files programatically is useful. You
| might now try to delete mytest.R using file.remove('mytest.R'), but that
| won t work since mytest.R no longer exists. You have already renamed it.
```

```
...

  |=================================================                     |  72%
| Make a copy of "mytest2.R" called "mytest3.R" using file.copy().

> file.copy("mytest2.R")
Error in file.copy("mytest2.R") :
  el argumento "to" está ausente, sin valor por omisión
> file.copy("mytest2.R","mytest3.R")
[1] TRUE

| You are amazing!

  |==================================================                    |  74%
| You now have two files in the current directory. That may not seem very
| interesting. But what if you were working with dozens, or millions, of
| individual files? In that case, being able to programatically act on many
| files would be absolutely necessary. Do not forget that you can, temporarily,
| leave the lesson by typing play() and then return by typing nxt().

...

  |===================================================                   |  77%
| Provide the relative path to the file "mytest3.R" by using file.path().

> file.path("mytest3.R")
[1] "mytest3.R"

| You are the best!

  |====================================================                  |  79%
| You can use file.path to construct file and directory paths that are
| independent of the operating system your R code is running on. Pass
| 'folder1' and 'folder2' as arguments to file.path to make a
| platform-independent pathname.

> file.path("folder1", "folder2")
[1] "folder1/folder2"

| You are quite good my friend!

  |=====================================================                 |  82%
| Take a look at the documentation for dir.create by entering ?dir.create .
| Notice the 'recursive' argument. In order to create nested directories,
| 'recursive' must be set to TRUE.

> ?dir.create

| Nice work!

  |======================================================                |  85%
| Create a directory in the current working directory called "testdir2" and a
| subdirectory for it called "testdir3", all in one command by using
| dir.create() and file.path().
```

```
> dir.create(file.path("testdir2", "testdir3"), recursive = TRUE)

| You got it right!

  |=============================================================          |  87%
| Go back to your original working directory using setwd(). (Recall that we
| created the variable old.dir with the full path for the orginal working
| directory at the start of these questions.)

> setwd()
Error in setwd() : el argumento "dir" está ausente, sin valor por omisión
> setwd(old.dir)

| You are really on a roll!

  |==============================================================        |  90%
| It is often helpful to save the settings that you had before you began an
| analysis and then go back to them at the end. This trick is often used
| within functions; you save, say, the par() settings that you started with,
| mess around a bunch, and then set them back to the original values at the
| end. This is not the same as what we have done here, but it seems similar
| enough to mention.

...

  |===============================================================      |  92%
| After you finish this lesson delete the 'testdir' directory that you just
| left (and everything in it)

...

  |================================================================     |  95%
| Take nothing but results. Leave nothing but assumptions. That sounds like
| 'Take nothing but pictures. Leave nothing but footprints.' But it makes no
| sense! Surely our readers can come up with a better motto . . .

...

  |=================================================================    |  97%
| In this lesson, you learned how to examine your R workspace and work with
| the file system of your machine from within R. Thanks for playing!

...

  |======================================================================| 100%
| Would you like to receive credit for completing this course on Coursera.org?

1: Yes
2: No
```

Vfvf

Swirl 3: Sequence of numbers

Rgb

```
Selection: 3

  |                                                                      |   0%

| In this lesson, you will learn how to create sequences of numbers in R.

...

  |===                                                                   |   4%
| The simplest way to create a sequence of numbers in R is by using the `:`
| operator. Type 1:20 to see how it works.

> 1:20
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

| All that practice is paying off!

  |======                                                                |   9%
| That gave us every integer between (and including) 1 and 20. We could also
| use it to create a sequence of real numbers. For example, try pi:10.

> pi:10
[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593

| You are quite good my friend!

  |=========                                                             |  13%
| The result is a vector of real numbers starting with pi (3.142...) and
| increasing in increments of 1. The upper limit of 10 is never reached, since
| the next number in our sequence would be greater than 10.

...

  |============                                                          |  17%
| What happens if we do 15:1? Give it a try to find out.

> 15:1
 [1] 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1

| Great job!

  |===============                                                       |  22%
| It counted backwards in increments of 1! It's unlikely we'd want this
| behavior, but nonetheless it is good to know how it could happen.

...

  |==================                                                    |  26%
| Remember that if you have questions about a particular R function, you can
| access its documentation with a question mark followed by the function name:
| ?function_name_here. However, in the case of an operator like the colon used
| above, you must enclose the symbol in backticks like this: ?`:`. (NOTE: The
| backtick (`) key is generally located in the top left corner of a keyboard,
```

```
| above the Tab key. If you do not have a backtick key, you can use regular
| quotes.)

...

  |====================                                                   |  30%
| Pull up the documentation for `:` now.

> ?:
Error: inesperado ':' in "?:"
> :
Error: inesperado ':' in ":"
> ?':'

| You got it right!

  |========================                                               |  35%
| Often, we'll desire more control over a sequence we're creating than what
| the `:` operator gives us. The seq() function serves this purpose.

...

  |============================                                           |  39%
| The most basic use of seq() does exactly the same thing as the `:` operator.
| Try seq(1, 20) to see this.

> seq(1,20)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

| You nailed it! Good job!

  |================================                                       |  43%
| This gives us the same output as 1:20. However, let is say that instead we
| want a vector of numbers ranging from 0 to 10, incremented by 0.5. seq(0,
| 10, by=0.5) does just that. Try it out.

> seq(1,10,by=0.5)
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
[16]  8.5  9.0  9.5 10.0

| Not quite, but you are learning! Try again. Or, type info() for more options.

| You are still using the seq() function here, but this time with an extra
| argument that tells R you want to increment your sequence by 0.5. Try seq(0,
| 10, by=0.5).

> seq(0,10,by=0.5)
 [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
[16]  7.5  8.0  8.5  9.0  9.5 10.0

| Excellent work!

  |================================                                       |  48%
| Or maybe we do not care what the increment is and we just want a sequence of
```

```
| 30 numbers between 5 and 10. seq(5, 10, length=30) does the trick. Give it a
| shot now and store the result in a new variable called my_seq.

> my_seq <- seq(5,10,length(30))

| One more time. You can do it! Or, type info() for more options.

| You are using the same function here, but changing its arguments for
| different results. Be sure to store the result in a new variable called
| my_seq, like this: my_seq <- seq(5, 10, length=30).

> my_seq <- seq(5,10,length=30)

| You are quite good my friend!

  |=====================================                                 |  52%
| To confirm that my_seq has length 30, we can use the length() function. Try
| it now.

> length(my_seq)
[1] 30

| All that hard work is paying off!

  |======================================                                |  57%
| Let's pretend we don't know the length of my_seq, but we want to generate a
| sequence of integers from 1 to N, where N represents the length of the
| my_seq vector. In other words, we want a new vector (1, 2, 3, ...) that is
| the same length as my_seq.

...

  |=========================================                             |  61%
| There are several ways we could do this. One possibility is to combine the
| `:` operator and the length() function like this: 1:length(my_seq). Give
| that a try.

> 1:length(my_seq)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30

| You got it!

  |============================================                          |  65%
| Another option is to use seq(along.with = my_seq). Give that a try.

> seq(along.with = my_seq)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30

| You are really on a roll!

  |===============================================                       |  70%
| However, as is the case with many common tasks, R has a separate built-in
```

```
| function for this purpose called seq_along(). Type seq_along(my_seq) to see
| it in action.

> seq_along(my_seq)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30

| Keep working like that and you will get there!

  |=======================================================               |  74%
| There are often several approaches to solving the same problem, particularly
| in R. Simple approaches that involve less typing are generally best. It is
| also important for your code to be readable, so that you and others can
| figure out what is going on without too much hassle.

...

  |========================================================              |  78%
| If R has a built-in function for a particular task, it is likely that
| function is highly optimized for that purpose and is your best option. As
| you become a more advanced R programmer, you will design your own functions t
o
| perform tasks when there are no better options. We'll explore writing your
| own functions in future lessons.

...

  |=========================================================             |  83%
| One more function related to creating sequences of numbers is rep(), which
| stands for 'replicate'. Let is look at a few uses.

...

  |==========================================================            |  87%
| If we're interested in creating a vector that contains 40 zeros, we can use
| rep(0, times = 40). Try it out.

> rep(0,times=40)
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[38] 0 0 0

| Your dedication is inspiring!

  |============================================================          |  91%
| If instead we want our vector to contain 10 repetitions of the vector (0, 1,
| 2), we can do rep(c(0, 1, 2), times = 10). Go ahead.

> rep(c(0,1,2), times= 10)
 [1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2

| That is the answer I was looking for.

  |=============================================================         |  96%
| Finally, let is say that rather than repeating the vector (0, 1, 2) over and
```

```
| over again, we want our vector to contain 10 zeros, then 10 ones, then 10
| twos. We can do this with the `each` argument. Try rep(c(0, 1, 2), each =
| 10).

> rep(c(0,1,2), each= 10)
 [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2

| All that practice is paying off!

  |======================================================================| 100%
| Would you like to receive credit for completing this course on Coursera.org?

1: No
2: Yes
```

rgbr

Swirl 4: Vectors

Gr bg

```
Selection: 4

  |                                                                      |   0%

| The simplest and most common data structure in R is the vector.

...

  |==                                                                    |   3%
| Vectors come in two different flavors: atomic vectors and lists. An atomic
| vector contains exactly one data type, whereas a list may contain multiple
| data types. We'll explore atomic vectors further before we get to lists.

...

  |====                                                                  |   5%
| In previous lessons, we dealt entirely with numeric vectors, which are one
| type of atomic vector. Other types of atomic vectors include logical,
| character, integer, and complex. In this lesson, we'll take a closer look at
| logical and character vectors.

...

  |=====                                                                 |   8%
| Logical vectors can contain the values TRUE, FALSE, and NA (for 'not
| available'). These values are generated as the result of logical
| 'conditions'. Let is experiment with some simple conditions.

...

  |=======                                                               |  11%
| First, create a numeric vector num_vect that contains the values 0.5, 55,
| -10, and 6.
```

```
> num_vect <- c(0.5,55,-10,6)

| You are the best!

  |=========                                                             |  13%
| Now, create a variable called tf that gets the result of num_vect < 1, which
| is read as 'num_vect is less than 1'.

> tf <- num_vect<1

| Excellent work!

  |===========                                                           |  16%
| What do you think tf will look like?

1: a single logical value
2: a vector of 4 logical values

Selection: 1

| You almost had it, but not quite. Try again.

| Remember our lesson on vector arithmetic? The theme was that R performs many
| operations on an element-by-element basis. We called these 'vectorized'
| operations.

1: a vector of 4 logical values
2: a single logical value

Selection: 1

| You got it!

  |=============                                                         |  18%
| Print the contents of tf now.

> tf
[1]  TRUE FALSE  TRUE FALSE

| Keep working like that and you will get there!

  |===============                                                       |  21%
| The statement num_vect < 1 is a condition and tf tells us whether each
| corresponding element of our numeric vector num_vect satisfies this
| condition.

...

  |================                                                      |  24%
| The first element of num_vect is 0.5, which is less than 1 and therefore the
| statement 0.5 < 1 is TRUE. The second element of num_vect is 55, which is
| greater than 1, so the statement 55 < 1 is FALSE. The same logic applies for
| the third and fourth elements.
```

```
...

  |=================                                                     |  26%
| Let is try another. Type num_vect >= 6 without assigning the result to a new
| variable.

> num_vect >= 6
[1] FALSE  TRUE FALSE  TRUE

| You are amazing!

  |===================                                                   |  29%
| This time, we are asking whether each individual element of num_vect is
| greater than OR equal to 6. Since only 55 and 6 are greater than or equal to
| 6, the second and fourth elements of the result are TRUE and the first and
| third elements are FALSE.

...

  |======================                                                |  32%
| The `<` and `>=` symbols in these examples are called 'logical operators'.
| Other logical operators include `>`, `<=`, `==` for exact equality, and `!=`
| for inequality.

...

  |========================                                              |  34%
| If we have two logical expressions, A and B, we can ask whether at least one
| is TRUE with A | B (logical 'or' a.k.a. 'union') or whether they are both
| TRUE with A & B (logical 'and' a.k.a. 'intersection'). Lastly, !A is the
| negation of A and is TRUE when A is FALSE and vice versa.

...

  |=========================                                             |  37%
| It is a good idea to spend some time playing around with various combinations
| of these logical operators until you get comfortable with their use. We'll
| do a few examples here to get you started.

...

  |===========================                                           |  39%
| Try your best to predict the result of each of the following statements. You
| can use pencil and paper to work them out if it's helpful. If you get stuck,
| just guess and you've got a 50% chance of getting the right answer!

...

  |============================                                          |  42%
| (3 > 5) & (4 == 4)

1: FALSE
2: TRUE
```

```
Selection: 1

| You nailed it! Good job!

  |================================                                      |  45%
| (TRUE == TRUE) | (TRUE == FALSE)

1: TRUE
2: FALSE

Selection: 1

| Your dedication is inspiring!

  |=================================                                     |  47%
| ((111 >= 111) | !(TRUE)) & ((4 + 1) == 5)

1: TRUE
2: FALSE

Selection: 1

| Nice work!

  |==================================                                    |  50%
| Don't worry if you found these to be tricky. They're supposed to be. Working
| with logical statements in R takes practice, but your efforts will be
| rewarded in future lessons (e.g. subsetting and control structures).

...

  |====================================                                  |  53%
| Character vectors are also very common in R. Double quotes are used to
| distinguish character objects, as in the following example.

...

  |=====================================                                 |  55%
| Create a character vector that contains the following words: "My", "name",
| "is". Remember to enclose each word in its own set of double quotes, so that
| R knows they are character strings. Store the vector in a variable called
| my_char.

> my_char <- c("My","name","is")

| You are really on a roll!

  |=======================================                               |  58%
| Print the contents of my_char to see what it looks like.

> my_char
[1] "My"   "name" "is"
```

```
| You are amazing!

  |=========================================                             |  61%
| Right now, my_char is a character vector of length 3. Let is say we want to
| join the elements of my_char together into one continuous character string
| (i.e. a character vector of length 1). We can do this using the paste()
| function.

...

  |==========================================                            |  63%
| Type paste(my_char, collapse = " ") now. Make sure there's a space between
| the double quotes in the `collapse` argument. You will see why in a second.

> paste(my_char, collapse = " ")
[1] "My name is"

| You are amazing!

  |===========================================                           |  66%
| The `collapse` argument to the paste() function tells R that when we join
| together the elements of the my_char character vector, we'd like to separate
| them with single spaces.

...

  |============================================                          |  68%
| It seems that we're missing something.... Ah, yes! Your name!

...

  |=============================================                         |  71%
| To add (or 'concatenate') your name to the end of my_char, use the c()
| function like this: c(my_char, "your_name_here"). Place your name in double
| quotes where I have put "your_name_here". Try it now, storing the result in a
| new variable called my_name.

> c(my_char, "Janice")
[1] "My"     "name"   "is"     "Janice"

| Not quite right, but keep trying. Or, type info() for more options.

| Tack your name on to the end of the my_char vector using the c() function.
| Be sure to assign the result to a new variable called my_name. If your name
| was "Swirl", you would type my_name <- c(my_char, "Swirl").

> my_name <- c(my_char, "Janice")

| You got it right!

  |==============================================                        |  74%
| Take a look at the contents of my_name.

> my_name
```

```
[1] "My"     "name"   "is"     "Janice"

| Excellent job!

  |=======================================================               |  76%
| Now, use the paste() function once more to join the words in my_name
| together into a single character string. Do not forget to say collapse = " "!

> paste(my_name,"a", collapse = " ")
[1] "My a name a is a Janice a"

| Give it another try. Or, type info() for more options.

| Use paste(my_name, collapse = " ") to join all four words together,
| separated by single spaces.

> paste(c(my_name,"a"), collapse = " ")
[1] "My name is Janice a"

| Nice try, but that is not exactly what I was hoping for. Try again. Or, type
| info() for more options.

| Use paste(my_name, collapse = " ") to join all four words together,
| separated by single spaces.

> paste(my_name, collapse = " ")
[1] "My name is Janice"

| You got it right!

  |=========================================================             |  79%
| In this example, we used the paste() function to collapse the elements of a
| single character vector. paste() can also be used to join the elements of
| multiple character vectors.

...

  |===========================================================           |  82%
| In the simplest case, we can join two character vectors that are each of
| length 1 (i.e. join two words). Try paste("Hello", "world!", sep = " "),
| where the `sep` argument tells R that we want to separate the joined
| elements with a single space.

> paste("Hello","world!", sep = " ")
[1] "Hello world!"

| That is correct!

  |=============================================================         |  84%
| For a slightly more complicated example, we can join two vectors, each of
| length 3. Use paste() to join the integer vector 1:3 with the character
| vector c("X", "Y", "Z"). This time, use sep = "" to leave no space between
| the joined elements.
```

```
> paste(c(1:3,"X","Y","Z"), sep = "")
[1] "1" "2" "3" "X" "Y" "Z"

| Give it another try. Or, type info() for more options.

| Use paste(1:3, c("X", "Y", "Z"), sep = "") to see what happens when we join
| two vectors of equal length using paste().

> paste(1:3,c("X","Y","Z"), sep = "")
[1] "1X" "2Y" "3Z"

| Keep working like that and you will get there!

  |=============================================================          |  87%
| What do you think will happen if our vectors are of different length? (Hint:
| we talked about this in a previous lesson.)

...

  |==============================================================        |  89%
| Vector recycling! Try paste(LETTERS, 1:4, sep = "-"), where LETTERS is a
| predefined variable in R containing a character vector of all 26 letters in
| the English alphabet.

> paste(LETTERS,1:4, sep = "-")
 [1] "A-1" "B-2" "C-3" "D-4" "E-1" "F-2" "G-3" "H-4" "I-1" "J-2" "K-3" "L-4"
[13] "M-1" "N-2" "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4"
[25] "Y-1" "Z-2"

| You are the best!

  |================================================================      |  92%
| Since the character vector LETTERS is longer than the numeric vector 1:4, R
| simply recycles, or repeats, 1:4 until it matches the length of LETTERS.

...

  |=================================================================     |  95%
| Also worth noting is that the numeric vector 1:4 gets 'coerced' into a
| character vector by the paste() function.

...

  |===================================================================   |  97%
| We'll discuss coercion in another lesson, but all it really means is that
| the numbers 1, 2, 3, and 4 in the output above are no longer numbers to R,
| but rather characters "1", "2", "3", and "4".

...

  |======================================================================| 100%
| Would you like to receive credit for completing this course on Coursera.org?

1: No
```

```
2: Yes
```

bnnnh

Swirl 5: Missing Values

Gbsbsr

```
Selection: 5

  |                                                                      |   0%

| Missing values play an important role in statistics and data analysis.
| Often, missing values must not be ignored, but rather they should be
| carefully studied to see if there's an underlying pattern or cause for their
| missingness.

...

  |===                                                                   |   5%
| In R, NA is used to represent any value that is 'not available' or 'missing'
| (in the statistical sense). In this lesson, we'll explore missing values
| further.

...

  |=======                                                               |  10%
| Any operation involving NA generally yields NA as the result. To illustrate,
| let is create a vector c(44, NA, 5, NA) and assign it to a variable x.

> x <- c(44,NA, 5, NA)

| That is a job well done!

  |==========                                                            |  15%
| Now, let is multiply x by 3.

> x*3
[1] 132  NA  15  NA

| Nice work!

  |==============                                                        |  20%
| Notice that the elements of the resulting vector that correspond with the NA
| values in x are also NA.

...

  |=================                                                     |  25%
| To make things a little more interesting, lets create a vector containing
| 1000 draws from a standard normal distribution with y <- rnorm(1000).

> y <- rnorm(1000)
```

```
| All that hard work is paying off!

  |=====================                                           |  30%
| Next, let is create a vector containing 1000 NAs with z <- rep(NA, 1000).

> z <- rep(NA, 1000)

| Nice work!

  |=======================                                         |  35%
| Finally, let is select 100 elements at random from these 2000 values
| (combining y and z) such that we do not know how many NAs we'll wind up with
| or what positions they will occupy in our final vector -- my_data <-
| sample(c(y, z), 100).

> my_data <- sample(c(y,z),100)

| You are the best!

  |=========================                                       |  40%
| Let is first ask the question of where our NAs are located in our data. The
| is.na() function tells us whether each element of a vector is NA. Call
| is.na() on my_data and assign the result to my_na.

> is.na(my_data)
  [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
 [13]  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
 [25] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
 [37] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [49]  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
 [61]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
 [73] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
 [85] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [97] FALSE FALSE  TRUE  TRUE

| You are close...I can feel it! Try it again. Or, type info() for more
| options.

| Assign the result of is.na(my_data) to the variable my_na.

> my_na <- is.na(my_data)

| Your dedication is inspiring!

  |===============================                                 |  45%
| Now, print my_na to see what you came up with.

> my_na
  [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
 [13]  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
 [25] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
 [37] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [49]  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
 [61]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
```

```
 [73] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
 [85] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [97] FALSE FALSE  TRUE  TRUE

| You are quite good my friend!

  |=================================                                     |  50%
| Everywhere you see a TRUE, you know the corresponding element of my_data is
| NA. Likewise, everywhere you see a FALSE, you know the corresponding element
| of my_data is one of our random draws from the standard normal distribution.

...

  |=====================================                                 |  55%
| In our previous discussion of logical operators, we introduced the `==`
| operator as a method of testing for equality between two objects. So, you
| might think the expression my_data == NA yields the same results as is.na().
| Give it a try.

> is.na(my_data ==NA)
  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [29] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [43] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [57] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [71] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [85] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [99] TRUE TRUE

| Keep trying! Or, type info() for more options.

| Try my_data == NA to see what happens.

> my_data ==NA
  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
 [25] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
 [49] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
 [73] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
 [97] NA NA NA NA

| You are doing so well!

  |=======================================                               |  60%
| The reason you got a vector of all NAs is that NA is not really a value, but
| just a placeholder for a quantity that is not available. Therefore the
| logical expression is incomplete and R has no choice but to return a vector
| of the same length as my_data that contains all NAs.

...

  |=========================================                             |  65%
| Don't worry if that's a little confusing. The key takeaway is to be cautious
| when using logical expressions anytime NAs might creep in, since a single NA
| value can derail the entire thing.
```

```
...

  |=================================================                     |  70%
| So, back to the task at hand. Now that we have a vector, my_na, that has a
| TRUE for every NA and FALSE for every numeric value, we can compute the
| total number of NAs in our data.

...

  |=====================================================                 |  75%
| The trick is to recognize that underneath the surface, R represents TRUE as
| the number 1 and FALSE as the number 0. Therefore, if we take the sum of a
| bunch of TRUEs and FALSEs, we get the total number of TRUEs.

...

  |=======================================================               |  80%
| Let is give that a try here. Call the sum() function on my_na to count the
| total number of TRUEs in my_na, and thus the total number of NAs in my_data.
| Don't assign the result to a new variable.

> sum(my_na==TRUE)
[1] 47

| Almost! Try again. Or, type info() for more options.

| Use sum(my_na) to count the number NAs in the data.

> sum(my_data==NA)
[1] NA

| Not quite! Try again. Or, type info() for more options.

| Use sum(my_na) to count the number NAs in the data.

> sum(my_na==NA)
[1] NA

| Not quite! Try again. Or, type info() for more options.

| Use sum(my_na) to count the number NAs in the data.

> sum(my_na)
[1] 47

| You got it right!

  |=========================================================             |  85%
| Pretty cool, huh? Finally, let is take a look at the data to convince
| ourselves that everything 'adds up'. Print my_data to the console.

> my_data
  [1]         NA -1.6311744         NA -1.0257692  0.6059293 -1.0072141
```

```
 [7]  0.4055039  1.9318936  1.0694074 -0.2495153         NA         NA
[13]         NA  0.5103969         NA         NA         NA  0.4003984
[19]         NA -1.7227250         NA -0.3155746 -0.1151023  1.1954787
[25] -0.3044753         NA         NA         NA  0.8955961  0.7285367
[31] -1.7156727  0.7507755         NA         NA -1.4032388         NA
[37] -0.3471497         NA -1.3521822  0.7563810 -0.2561177         NA
[43] -2.0143847         NA         NA         NA         NA         NA
[49]         NA         NA  0.5837084 -0.2768640         NA  0.1524038
[55] -1.2276445  0.9992538         NA -0.2185423 -1.4152395 -1.1586045
[61]         NA  1.3821829  1.4934638  1.8118145 -1.3447612         NA
[67]         NA -0.5245200         NA  1.9451036         NA         NA
[73] -1.6504217 -0.3822693         NA         NA         NA         NA
[79]  0.9045019  0.3354324 -0.8504760         NA -1.2412695         NA
[85] -0.9883835 -1.0464585  0.9176316         NA -0.4186384 -1.2675710
[91]  0.4292441         NA         NA         NA         NA         NA
[97] -0.2460356  1.0581483         NA         NA

| All that hard work is paying off!

  |=======================================================          |  90%
| Now that we have got NAs down pat, let is look at a second type of missing
| value -- NaN, which stands for 'not a number'. To generate NaN, try dividing
| (using a forward slash) 0 by 0 now.

> 0/0
[1] NaN

| You are the best!

  |=================================================================   |  95%
| Let is do one more, just for fun. In R, Inf stands for infinity. What happens
| if you subtract Inf from Inf?

> Inf - Inf
[1] NaN

| You are amazing!

  |=================================================================| 100%
| Would you like to receive credit for completing this course on Coursera.org?

1: No
2: Yes
```

rvrg

Swirl 6: Subsetting and Vectors

Hngv

Swirl 7: Matrices and Data Frames

Swirl 8: Logic

```
Selection: 8

  |                                                                      |   0%

| This lesson is meant to be a short introduction to logical operations in
| R.

...

  |=                                                                     |   2%
| There are two logical values in R, also called boolean values. They are
| TRUE and FALSE. In R you can construct logical expressions which will
| evaluate to either TRUE or FALSE.

...

  |===                                                                   |   4%
| Many of the questions in this lesson will involve evaluating logical
| expressions. It may be useful to open up a second R terminal where you can
| experiment with some of these expressions.

...

  |====                                                                  |   6%
| Creating logical expressions requires logical operators. You are probably
| familiar with arithmetic operators like `+`, `-`, `*`, and `/`. The first
| logical operator we are going to discuss is the equality operator,
| represented by two equals signs `==`. Use the equality operator below to
| find out if TRUE is equal to TRUE.

>
> TRUE==TRUE
[1] TRUE

| Your dedication is inspiring!

  |=====                                                                 |   8%
| Just like arithmetic, logical expressions can be grouped by parenthesis so
| that the entire expression (TRUE == TRUE) == TRUE evaluates to TRUE.

...

  |======                                                                |  10%
| To test out this property, try evaluating (FALSE == TRUE) == FALSE .
```

```
> (FALSE ==TRUE) ==FALSE
[1] TRUE

| Nice work!

  |========                                                              |  12%
| The equality operator can also be used to compare numbers. Use `==` to see
| if 6 is equal to 7.

> 6 == 7
[1] FALSE

| Nice work!

  |=========                                                             |  13%
| The previous expression evaluates to FALSE because 6 is less than 7.
| Thankfully, there are inequality operators that allow us to test if a value
| is less than or greater than another value.

...

  |==========                                                            |  15%
| The less than operator `<` tests whether the number on the left side of the
| operator (called the left operand) is less than the number on the right
| side of the operator (called the right operand). Write an expression to
| test whether 6 is less than 7.

> 6 < 7
[1] TRUE

| Excellent work!

  |============                                                          |  17%
| There is also a less-than-or-equal-to operator `<=` which tests whether the
| left operand is less than or equal to the right operand. Write an
| expression to test whether 10 is less than or equal to 10.

> 10 <= 10
[1] TRUE

| Excellent job!

  |=============                                                         |  19%
| Keep in mind that there are the corresponding greater than `>` and
| greater-than-or-equal-to `>=` operators.

...

  |==============                                                        |  21%
| Which of the following evaluates to FALSE?

1: 7 == 7
2: 6 < 8
3: 9 >= 10
```

```
4: 0 > -36

Selection: 3

| Keep up the great work!

  |===============                                                  |  23%
| Which of the following evaluates to TRUE?

1: 7 == 9
2: 57 < 8
3: 9 >= 10
4: -6 > -7

Selection: 4

| You are really on a roll!

  |=================                                                |  25%
| The next operator we will discuss is the 'not equals' operator represented
| by `!=`. Not equals tests whether two values are unequal, so TRUE != FALSE
| evaluates to TRUE. Like the equality operator, `!=` can also be used with
| numbers. Try writing an expression to see if 5 is not equal to 7.

> 5 != 7
[1] TRUE

| You are amazing!

  |==================                                               |  27%
| In order to negate boolean expressions you can use the NOT operator. An
| exclamation point `!` will cause !TRUE (say: not true) to evaluate to FALSE
| and !FALSE (say: not false) to evaluate to TRUE. Try using the NOT operator
| and the equals operator to find the opposite of whether 5 is equal to 7.

> !(5 == 7)
[1] TRUE

| Keep working like that and you will get there!

  |====================                                             |  29%
| Let is take a moment to review. The equals operator `==` tests whether two
| boolean values or numbers are equal, the not equals operator `!=` tests
| whether two boolean values or numbers are unequal, and the NOT operator `!`
| negates logical expressions so that TRUE expressions become FALSE and FALSE
| expressions become TRUE.

...

  |=====================                                            |  31%
| Which of the following evaluates to FALSE?

1: !FALSE
2: 9 < 10
```

```
3: 7 != 8
4: !(0 >= -1)

Selection: 4

| That is correct!

  |======================                                                |  33%
| What do you think the following expression will evaluate to?: (TRUE !=
| FALSE) == !(6 == 7)

1: %>%
2: TRUE
3: Can there be objective truth when programming?
4: FALSE

Selection: 2

| All that hard work is paying off!

  |=======================                                               |  35%
| At some point you may need to examine relationships between multiple
| logical expressions. This is where the AND operator and the OR operator
| come in.

...

  |========================                                              |  37%
| Let is look at how the AND operator works. There are two AND operators in R,
| `&` and `&&`. Both operators work similarly, if the right and left operands
| of AND are both TRUE the entire expression is TRUE, otherwise it is FALSE.
| For example, TRUE & TRUE evaluates to TRUE. Try typing FALSE & FALSE to how
| it is evaluated.

> FALSE & FALSE
[1] FALSE

| You are the best!

  |=========================                                             |  38%
| You can use the `&` operator to evaluate AND across a vector. The `&&`
| version of AND only evaluates the first member of a vector. Let is test both
| for practice. Type the expression TRUE & c(TRUE, FALSE, FALSE).

> TRUE & c(TRUE, FALSE, FALSE)
[1]  TRUE FALSE FALSE

| All that hard work is paying off!

  |==========================                                            |  40%
| What happens in this case is that the left operand `TRUE` is recycled
| across every element in the vector of the right operand. This is the
| equivalent statement as c(TRUE, TRUE, TRUE) & c(TRUE, FALSE, FALSE).
```

```
...

  |============================                                          |  42%
| Now we'll type the same expression except we'll use the `&&` operator. Type
| the expression TRUE && c(TRUE, FALSE, FALSE).

> TRUE && c(TRUE, FALSE, FALSE)
[1] TRUE

| You are quite good my friend!

  |=============================                                         |  44%
| In this case, the left operand is only evaluated with the first member of
| the right operand (the vector). The rest of the elements in the vector
| are not evaluated at all in this expression.

...

  |==============================                                        |  46%
| The OR operator follows a similar set of rules. The `|` version of OR
| evaluates OR across an entire vector, while the `||` version of OR only
| evaluates the first member of a vector.

...

  |===============================                                       |  48%
| An expression using the OR operator will evaluate to TRUE if the left
| operand or the right operand is TRUE. If both are TRUE, the expression will
| evaluate to TRUE, however if neither are TRUE, then the expression will be
| FALSE.

...

  |================================                                      |  50%
| Let is test out the vectorized version of the OR operator. Type the
| expression TRUE | c(TRUE, FALSE, FALSE).

> TRUE | c(TRUE, FALSE, FALSE)
[1] TRUE TRUE TRUE

| You got it right!

  |=================================                                     |  52%
| Now let s try out the non-vectorized version of the OR operator. Type the
| expression TRUE || c(TRUE, FALSE, FALSE).

> TRUE || c(TRUE, FALSE, FALSE)
[1] TRUE

| You nailed it! Good job!

  |==================================                                    |  54%
| Logical operators can be chained together just like arithmetic operators.
| The expressions: `6 != 10 && FALSE && 1 >= 2` or `TRUE || 5 < 9.3 || FALSE`
```

```
| are perfectly normal to see.

...

  |=====================================                                |  56%
| As you may recall, arithmetic has an order of operations and so do logical
| expressions. All AND operators are evaluated before OR operators. Let is
| look at an example of an ambiguous case. Type: 5 > 8 || 6 != 8 && 4 > 3.9

> 5 > 8 || 6 != 8 && 4 > 3.9
[1] TRUE

| That is a job well done!

  |======================================                               |  58%
| Let is walk through the order of operations in the above case. First the
| left and right operands of the AND operator are evaluated. 6 is not equal
| 8, 4 is greater than 3.9, therefore both operands are TRUE so the resulting
| expression `TRUE && TRUE` evaluates to TRUE. Then the left operand of the
| OR operator is evaluated: 5 is not greater than 8 so the entire expression
| is reduced to FALSE || TRUE. Since the right operand of this expression is
| TRUE the entire expression evaluates to TRUE.

...

  |=======================================                              |  60%
| Which one of the following expressions evaluates to TRUE?

1: FALSE || TRUE && FALSE
2: 99.99 > 100 || 45 < 7.3 || 4 != 4.0
3: TRUE && 62 < 62 && 44 >= 44
4: TRUE && FALSE || 9 >= 4 && 3 < 6

Selection: 4

| Keep up the great work!

  |========================================                             |  62%
| Which one of the following expressions evaluates to FALSE?

1: 6 >= -9 && !(6 > 7) && !(!TRUE)
2: FALSE && 6 >= 6 || 7 >= 8 || 50 <= 49.5
3: FALSE || TRUE && 6 != 4 || 9 > 4
4: !(8 > 4) ||  5 == 5.0 && 7.8 >= 7.79

Selection: 1

| Not quite! Try again.

| Try to evaluate each expression in isolation and build up an answer.

1: !(8 > 4) ||  5 == 5.0 && 7.8 >= 7.79
2: FALSE && 6 >= 6 || 7 >= 8 || 50 <= 49.5
3: 6 >= -9 && !(6 > 7) && !(!TRUE)
```

```
4: FALSE || TRUE && 6 != 4 || 9 > 4

Selection: 3

| One more time. You can do it!

| Try to evaluate each expression in isolation and build up an answer.

1: !(8 > 4) ||  5 == 5.0 && 7.8 >= 7.79
2: FALSE && 6 >= 6 || 7 >= 8 || 50 <= 49.5
3: FALSE || TRUE && 6 != 4 || 9 > 4
4: 6 >= -9 && !(6 > 7) && !(!TRUE)

Selection: 2

| Excellent job!

  |=========================================                             |  63%
| Now that you are familiar with R s logical operators you can take advantage
| of a few functions that R provides for dealing with logical expressions.

...

  |=========================================                             |  65%
| The function isTRUE() takes one argument. If that argument evaluates to
| TRUE, the function will return TRUE. Otherwise, the function will return
| FALSE. Try using this function by typing: isTRUE(6 > 4)

> isTRUE(6 > 4)
[1] TRUE

| Keep working like that and you will get there!

  |==========================================                            |  67%
| Which of the following evaluates to TRUE?

1: !isTRUE(8 != 5)
2: isTRUE(!TRUE)
3: !isTRUE(4 < 3)
4: isTRUE(NA)
5: isTRUE(3)

Selection: 3

| Perseverance, that is the answer.

  |==========================================                            |  69%
| The function identical() will return TRUE if the two R objects passed to it
| as arguments are identical. Try out the identical() function by typing:
| identical('twins', 'twins')

> identical('twins', 'twins')
[1] TRUE
```

```
| You got it right!

  |=================================================                     |  71%
| Which of the following evaluates to TRUE?

1: identical(4, 3.1)
2: identical(5 > 4, 3 < 3.1)
3: !identical(7, 7)
4: identical('hello', 'Hello')

Selection: 2

| That is the answer I was looking for.

  |==================================================                    |  73%
| You should also be aware of the xor() function, which takes two arguments.
| The xor() function stands for exclusive OR. If one argument evaluates to
| TRUE and one argument evaluates to FALSE, then this function will return
| TRUE, otherwise it will return FALSE. Try out the xor() function by typing:
| xor(5 == 6, !FALSE)

> xor(5 == 6, !FALSE)
[1] TRUE

| You nailed it! Good job!

  |===================================================                   |  75%
| 5 == 6 evaluates to FALSE, !FALSE evaluates to TRUE, so xor(FALSE, TRUE)
| evaluates to TRUE. On the other hand if the first argument was changed to 5
| == 5 and the second argument was unchanged then both arguments would have
| been TRUE, so xor(TRUE, TRUE) would have evaluated to FALSE.

...

  |====================================================                  |  77%
| Which of the following evaluates to FALSE?

1: xor(!isTRUE(TRUE), 6 > -1)
2: xor(!!TRUE, !!FALSE)
3: xor(identical(xor, 'xor'), 7 == 7.0)
4: xor(4 >= 9, 8 != 8.0)

Selection: 4

| Nice work!

  |=====================================================                 |  79%
| For the next few questions, we're going to need to create a vector of
| integers called ints. Create this vector by typing: ints <- sample(10)

> ints <- sample(10)

| Keep working like that and you will get there!
```

```
  |=================================================        |  81%
| Now simply display the contents of ints.

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

| You are the best!

  |==================================================       |  83%
| The vector `ints` is a random sampling of integers from 1 to 10 without
| replacement. Let is say we wanted to ask some logical questions about
| contents of ints. If we type ints > 5, we will get a logical vector
| corresponding to whether each element of ints is greater than 5. Try
| typing: ints > 5

> ints >5
 [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE

| You got it right!

  |==================================================       |  85%
| We can use the resulting logical vector to ask other questions about ints.
| The which() function takes a logical vector as an argument and returns the
| indices of the vector that are TRUE. For example which(c(TRUE, FALSE,
| TRUE)) would return the vector c(1, 3).

...

  |===================================================      |  87%
| Use the which() function to find the indices of ints that are greater than
| 7.

> which(ints > 7)
[1] 1 6 9

| All that hard work is paying off!

  |===================================================      |  88%
| Which of the following commands would produce the indices of the elements
| in ints that are less than or equal to 2?

1: which(ints <= 2)
2: which(ints < 2)
3: ints < 2
4: ints <= 2

Selection: 1

| That is a job well done!

  |====================================================     |  90%
| Like the which() function, the functions any() and all() take logical
| vectors as their argument. The any() function will return TRUE if one or
| more of the elements in the logical vector is TRUE. The all() function will
```

```
| return TRUE if every element in the logical vector is TRUE.

...

  |=============================================================     |  92%
| Use the any() function to see if any of the elements of ints are less than
| zero.

> any(ints<0)
[1] FALSE

| That is a job well done!

  |==============================================================    |  94%
| Use the all() function to see if all of the elements of ints are greater
| than zero.

> all(ints) > 0
[1] TRUE

| Not quite, but you are learning! Try again. Or, type info() for more
| options.

| Use the all() function on the logical vector produced by: `ints > 0`

> all
function (..., na.rm = FALSE)  .Primitive("all")

| Not quite right, but keep trying. Or, type info() for more options.

| Use the all() function on the logical vector produced by: `ints > 0`

> all(ints>0)
[1] TRUE

| You are amazing!

  |==============================================================    |  96%
| Which of the following evaluates to TRUE?

1: any(ints == 10)
2: all(ints == 10)
3: any(ints == 2.5)
4: all(c(TRUE, FALSE, TRUE))

Selection: 1

| You are really on a roll!

  |=============================================================== |  98%
| That is all for this introduction to logic in R. If you really want to see
| what you can do with logic, check out the control flow lesson!

...
```

```
  |======================================================================| 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: No
2: Yes
```

Swirl 9: Funciones

```
Selection: 9

  |                                                                      |   0%

| Functions are one of the fundamental building blocks of the R language.
| They are small pieces of reusable code that can be treated like any other R
| object.

...

  |=                                                                     |   2%
| If you have worked through any other part of this course, you have probably
| used some functions already. Functions are usually characterized by the
| name of the function followed by parentheses.

...

  |===                                                                   |   4%
| Let is try using a few basic functions just for fun. The Sys.Date() function
| returns a string representing today is date. Type Sys.Date() below and see
| what happens.

> Sys.Date()
[1] "2021-03-10"

| Excellent job!

  |====                                                                  |   6%
| Most functions in R return a value. Functions like Sys.Date() return a
| value based on your computer is environment, while other functions
| manipulate input data in order to compute a return value.

...

  |======                                                                |   8%
| The mean() function takes a vector of numbers as input, and returns the
| average of all of the numbers in the input vector. Inputs to functions are
| often called arguments. Providing arguments to a function is also sometimes
| called passing arguments to that function. Arguments you want to pass to a
| function go inside the function is parentheses. Try passing the argument
| c(2, 4, 5) to the mean() function.
```

```
> mean(c(2, 4, 5))
[1] 3.666667

| You got it!

  |=======                                                  |  10%
| Functions usually take arguments which are variables that the function
| operates on. For example, the mean() function takes a vector as an
| argument, like in the case of mean(c(2,6,8)). The mean() function then adds
| up all of the numbers in the vector and divides that sum by the length of
| the vector.

...

  |========                                                 |  12%
| In the following question you will be asked to modify a script that will
| appear as soon as you move on from this question. When you have finished
| modifying the script, save your changes to the script and type submit() and
| the script will be evaluated. There will be some comments in the script
| that opens up, so be sure to read them!

...

  |==========                                               |  14%
| The last R expression to be evaluated in a function will become the return
| value of that function. We want this function to take one argument, x, and
| return x without modifying it. Delete the pound sign so that x is returned
| without any modification. Make sure to save your script before you type
| submit().

> submit()

| Sourcing your script...


| You are doing so well!

  |===========                                              |  16%
| Now that you have created your first function let is test it! Type:
| boring_function('My first function!'). If your function works, it should
| just return the string: 'My first function!'

> boring_function('My first function!')
[1] "My first function!"

| That is the answer I was looking for.

  |============                                             |  18%
| Congratulations on writing your first function. By writing functions, you
| can gain serious insight into how R works. As John Chambers, the creator of
| R once said:
|
| To understand computations in R, two slogans are helpful: 1. Everything
| that exists is an object. 2. Everything that happens is a function call.
```

```
...

  |==============                                                        |  20%
| If you want to see the source code for any function, just type the function
| name without any arguments or parentheses. Let is try this out with the
| function you just created. Type: boring_function to view its source code.

> boring_function
function(x) {
  x
}
<bytecode: 0x000001c075a40ce0>

| That is the answer I was looking for.

  |===============                                                       |  22%
| Time to make a more useful function! We're going to replicate the
| functionality of the mean() function by creating a function called:
| my_mean(). Remember that to calculate the average of all of the numbers in
| a vector you find the sum of all the numbers in the vector, and then divide
| that sum by the number of numbers in the vector.

...

  |=================                                                     |  24%
| Make sure to save your script before you type submit().

> submit()

| Sourcing your script...


| All that hard work is paying off!

  |==================                                                    |  27%
| Now test out your my_mean() function by finding the mean of the vector c(4,
| 5, 10).

> my_mean
function(my_vector) {
  suma <- sum(my_vector)
  long <- length(my_vector)
  suma/long
  # Write your code here!
  # Remember: the last expression evaluated will be returned!
}
<bytecode: 0x000001c076d80448>

| One more time. You can do it! Or, type info() for more options.

| Run the command my_mean(c(4, 5, 10)) to test out your new function.

> my_mean(c(4,5,10))
```

```
[1] 6.333333

| That is correct!

  |==================                                                    |  29%
| Next, let is try writing a function with default arguments. You can set
| default values for a function is arguments, and this can be useful if you
| think someone who uses your function will set a certain argument to the
| same value most of the time.

...

  |====================                                                  |  31%
| Make sure to save your script before you type submit().

> submit()

| Sourcing your script...


| Try again. Getting it right on the first try is boring anyway!

| Remember to set the appropriate default values!

> submit()

| Sourcing your script...


| That is correct!

  |======================                                                |  33%
| Let is do some testing of the remainder function. Run remainder(5) and see
| what happens.

> remainder(5)
[1] 1

| Excellent job!

  |========================                                              |  35%
| Let is take a moment to examine what just happened. You provided one
| argument to the function, and R matched that argument to 'num' since 'num'
| is the first argument. The default value for 'divisor' is 2, so the
| function used the default value you provided.

...

  |==========================                                            |  37%
| Now let is test the remainder function by providing two arguments. Type:
| remainder(11, 5) and let is see what happens.

> remainder(11, 5)
[1] 1
```

```
| That is correct!

  |=========================                                             |  39%
| Once again, the arguments have been matched appropriately.

...

  |==========================                                            |  41%
| You can also explicitly specify arguments in a function. When you
| explicitly designate argument values by name, the ordering of the arguments
| becomes unimportant. You can try this out by typing: remainder(divisor =
| 11, num = 5).

> remainder(divisor = 11, num = 5)
[1] 5

| All that practice is paying off!

  |===========================                                           |  43%
| As you can see, there is a significant difference between remainder(11, 5)
| and remainder(divisor = 11, num = 5)!

...

  |============================                                          |  45%
| R can also partially match arguments. Try typing remainder(4, div = 2) to
| see this feature in action.

> remainder(4, div = 2)
[1] 0

| You got it!

  |============================                                          |  47%
| A word of warning: in general you want to make your code as easy to
| understand as possible. Switching around the orders of arguments by
| specifying their names or only using partial argument names can be
| confusing, so use these features with caution!

...

  |=============================                                         |  49%
| With all of this talk about arguments, you may be wondering if there is a
| way you can see a function is arguments (besides looking at the
| documentation). Thankfully, you can use the args() function! Type:
| args(remainder) to examine the arguments for the remainder function.

> args(remainder)
function (num, divisor = 2)
NULL

| You are amazing!
```

```
  |=================================                                     |  51%
| You may not realize it but I just tricked you into doing something pretty
| interesting! args() is a function, remainder() is a function, yet remainder
| was an argument for args(). Yes it is true: you can pass functions as
| arguments! This is a very powerful concept. Let is write a script to see how
| it works.

...

  |==================================                                    |  53%
| Make sure to save your script before you type submit().

> submit()

| Sourcing your script...


| That's the answer I was looking for.

  |===================================                                   |  55%
| Let's take your new evaluate() function for a spin! Use evaluate to find
| the standard deviation of the vector c(1.4, 3.6, 7.9, 8.8).

> evaluate(sd, c(1.4, 3.6, 7.9, 8.8))
[1] 3.514138

| Perseverance, that is the answer.

  |====================================                                  |  57%
| The idea of passing functions as arguments to other functions is an
| important and fundamental concept in programming.

...

  |=====================================                                 |  59%
| You may be surprised to learn that you can pass a function as an argument
| without first defining the passed function. Functions that are not named
| are appropriately known as anonymous functions.

...

  |======================================                                |  61%
| Let is use the evaluate function to explore how anonymous functions work.
| For the first argument of the evaluate function we are going to write a tiny
| function that fits on one line. In the second argument we'll pass some data
| to the tiny anonymous function in the first argument.

...

  |=======================================                               |  63%
| Type the following command and then we'll discuss how it works:
| evaluate(function(x){x+1}, 6)

> evaluate(function(x){x+1}, 6)
```

```
[1] 7

| You are the best!

  |=========================================                              |  65%
| The first argument is a tiny anonymous function that takes one argument `x`
| and returns `x+1`. We passed the number 6 into this function so the entire
| expression evaluates to 7.

...

  |=============================================                          |  67%
| Try using evaluate() along with an anonymous function to return the first
| element of the vector c(8, 4, 0). Your anonymous function should only take
| one argument which should be a variable `x`.

> evaluate(function(x){x[0]}, c(8,4,0))
numeric(0)

| Nice try, but that is not exactly what I was hoping for. Try again. Or, type
| info() for more options.

| You may need to recall how to index vector elements. Remember that your
| anonymous function should only have one argument, and that argument should
| be named `x`.

> evaluate(function(x){x[0]}, x = c(8,4,0))
Error in evaluate(function(x) { : unused argument (x = c(8, 4, 0))
> evaluate(function(x){x[0]}, c(8,4,0))
numeric(0)

| You are close...I can feel it! Try it again. Or, type info() for more
| options.

| You may need to recall how to index vector elements. Remember that your
| anonymous function should only have one argument, and that argument should
| be named `x`.

| Try using evaluate() along with an anonymous function to return the first
| element of the vector c(8, 4, 0). Your anonymous function should only take
| one argument which should be a variable `x`.

> evaluate(function(x){x[1]}, c(8,4,0))
[1] 8

| Great job!

  |=============================================                          |  69%
| Now try using evaluate() along with an anonymous function to return the
| last element of the vector c(8, 4, 0). Your anonymous function should only
| take one argument which should be a variable `x`.

> evaluate(function(x){x[length(x)]}, c(8,4,0))
[1] 0
```

```
| All that practice is paying off!

  |=================================================                     |  71%
| For the rest of the course we are going to use the paste() function
| frequently. Type ?paste so we can take a look at the documentation for the
| paste function.

> ?paste

| Keep working like that and you will get there!

  |==================================================                    |  73%
| As you can see the first argument of paste() is `...` which is referred to
| as an ellipsis or simply dot-dot-dot. The ellipsis allows an indefinite
| number of arguments to be passed into a function. In the case of paste()
| any number of strings can be passed as arguments and paste() will return
| all of the strings combined into one string.

...

  |===================================================                   |  76%
| Just to see how paste() works, type paste("Programming", "is", "fun!")

> paste("Programming", "is", "fun!")
[1] "Programming is fun!"

| That is a job well done!

  |====================================================                  |  78%
| Time to write our own modified version of paste().

...

  |=====================================================                 |  80%
| Make sure to save your script before you type submit().

> submit()

| Sourcing your script...


| You almost had it, but not quite. Try again.

| Remember that arguments can come after ...

> submit()

| Sourcing your script...


| All that hard work is paying off!

  |======================================================                |  82%
```

```
| Now let is test out your telegram function. Use your new telegram function
| passing in whatever arguments you wish!

> telegram("hola")
[1] "START hola STOP"

| Great job!

  |=======================================================           |  84%
| Make sure to save your script before you type submit().

> submit()

| Sourcing your script...


| Nice work!

  |=========================================================         |  86%
| Time to use your mad_libs function. Make sure to name the place, adjective,
| and noun arguments in order for your function to work.

> mad_libs(place, adjective, noun)
Error in mad_libs(place, adjective, noun) : objeto 'place' no encontrado
> mad_libs
function(...){
  # Do your argument unpacking here!
  args <- list(...)
  place <- args[["place"]]
  adjective <- args[["adjective"]]
  noun <- args[["noun"]]
  # Don't modify any code below this comment.
  # Notice the variables you'll need to create in order for the code below to
  # be functional!
  paste("News from", place, "today where", adjective, "students took to the str
eets in protest of the new", noun, "being installed on campus.")
}
<bytecode: 0x000001c07945ddd8>

| Not quite! Try again. Or, type info() for more options.

| Use the mad_libs function and be sure to name place, adjective, and noun
| arguments.

> mad_libs("place","adjective", "noun")
[1] "News from  today where  students took to the streets in protest of the new
  being installed on campus."

| Keep up the great work!

  |=========================================================         |  88%
| We're coming to the end of this lesson, but there's still one more idea you
| should be made aware of.
```

```
...

  |=========================================================             |  90%
| You are familiar with adding, subtracting, multiplying, and dividing numbers
| in R. To do this you use the +, -, *, and / symbols. These symbols are
| called binary operators because they take two inputs, an input from the
| left and an input from the right.

...

  |==========================================================            |  92%
| In R you can define your own binary operators. In the next script I will show
| you how.

...

  |============================================================          |  94%
| Make sure to save your script before you type submit().

> submit()

| Sourcing your script...


| That is correct!

  |==============================================================        |  96%
| You made your own binary operator! Let is test it out. Paste together the
| strings: 'I', 'love', 'R!' using your new binary operator.

> 'I' %p% 'love' %p% 'R!'
[1] "I love R!"

| You nailed it! Good job!

  |================================================================      |  98%
| We've come to the end of our lesson! Go out there and write some great
| functions!

...

  |==================================================================    | 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: No
2: Yes
```

A

Swirl 10: Lapply and Sapply

k

```
  |================                                                  |  26%
//| The 'l' in 'lapply' stands for 'list'. Type class(cls_list) to confirm
| that lapply() returned a list.

> clas(cls_list)
[1] "list"

| You are amazing!

  |=================                                                 |  28%
| As expected, we got a list of length 30 -- one element for each
| variable/column. The output would be considerably more compact if we
| could represent it as a vector instead of a list.

...

  |====================                                              |  30%
| You may remember from a previous lesson that lists are most helpful for
| storing multiple classes of data. In this case, since every element of
| the list returned by lapply() is a character vector of length one (i.e.
| "integer" and "vector"), cls_list can be simplified to a character
| vector. To do this manually, type as.character(cls_list).

> as.character(cls_list)
 [1] "character" "integer"   "integer"   "integer"   "integer"
 [6] "integer"   "integer"   "integer"   "integer"   "integer"
[11] "integer"   "integer"   "integer"   "integer"   "integer"
[16] "integer"   "integer"   "character" "integer"   "integer"
[21] "integer"   "integer"   "integer"   "integer"   "integer"
[26] "integer"   "integer"   "integer"   "character" "character"

| All that hard work is paying off!

  |=====================                                             |  32%
| sapply() allows you to automate this process by calling lapply() behind
| the scenes, but then attempting to simplify (hence the 's' in 'sapply')
| the result for you. Use sapply() the same way you used lapply() to get
//| the class of each column of the flags dataset and store the result in
//| cls_vect. If you need help, type sapply to bring up the documentation.

//> sapply(flags, clas)
       name    landmass        zone        area  population    language
"character"   "integer"   "integer"   "integer"   "integer"   "integer"
   religion        bars     stripes     colours         red       green
  "integer"   "integer"   "integer"   "integer"   "integer"   "integer"
       blue        gold       white       black      orange     mainhue
  "integer"   "integer"   "integer"   "integer"   "integer" "character"
    circles     crosses    saltires    quarters    sunstars    crescent
  "integer"   "integer"   "integer"   "integer"   "integer"   "integer"
   triangle        icon     animate        text     topleft    botright
  "integer"   "integer"   "integer"   "integer" "character" "character"

//| Nice try, but that is not exactly what I was hoping for. Try again. Or,
////| type info() for more options.
```

```
//| Type cls_vect <- sapply(flags, clas) to store the column classes in a
//| character vector called cls_vect.

//> cls_vect <- sapply(flags, clas)

//| Your dedication is inspiring!

//  |======================                                  |  34%
| Use clas(cls_vect) to confirm that sapply() simplified the result to a
| character vector.

//> clas(cls_vect)
[1] "character"

| That is a job well done!

  |========================                                |  36%
| In general, if the result is a list where every element is of length
| one, then sapply() returns a vector. If the result is a list where every
| element is a vector of the same length (> 1), sapply() returns a matrix.
| If sapply() can not figure things out, then it just returns a list, no
| different from what lapply() would give you.

...

  |=========================                               |  38%
| Let is practice using lapply() and sapply() some more!

...

  |==========================                              |  40%
| Columns 11 through 17 of our dataset are indicator variables, each
| representing a different color. The value of the indicator variable is 1
| if the color is present in a country is flag and 0 otherwise.

...

  |===========================                             |  42%
| Therefore, if we want to know the total number of countries (in our
| dataset) with, for example, the color orange on their flag, we can just
| add up all of the 1s and 0s in the 'orange' column. Try
| sum(flags$orange) to see this.

//> sum(flags$orange)
[1] 26

| That is a job well done!

  |============================                            |  44%
| Now we want to repeat this operation for each of the colors recorded in
| the dataset.

...
```

```
  |=============================                                    |  46%
| First, use flag_colors <- flags[, 11:17] to extract the columns
| containing the color data and store them in a new data frame called
| flag_colors. (Note the comma before 11:17. This subsetting command tells
| R that we want all rows, but only columns 11 through 17.)

> flags_colors <- flags[, 11:17]

| Almost! Try again. Or, type info() for more options.

| flag_colors <- flags[, 11:17] will get the job done!

> flag_colors <- flags[, 11:17]

| You are really on a roll!

  |==============================                                    |  48%
| Use the head() function to look at the first 6 lines of flag_colors.

> head(flag_colors)
  red green blue gold white black orange
1   1     1    0    1     1     1      0
2   1     0    0    1     0     1      0
3   1     1    0    0     1     0      0
4   1     0    1    1     1     0      1
5   1     0    1    1     0     0      0
6   1     0    0    1     0     1      0

| You are doing so well!

  |==============================                                    |  50%
| To get a list containing the sum of each column of flag_colors, call the
| lapply() function with two arguments. The first argument is the object
| over which we are looping (i.e. flag_colors) and the second argument is
| the name of the function we wish to apply to each column (i.e. sum).
| Remember that the second argument is just the name of the function with
| no parentheses, etc.

//> lapply(flag_colors, sum)
$red
[1] 153

$green
[1] 91

$blue
[1] 99

$gold
[1] 91

$white
[1] 146
```

```
$black
[1] 52

$orange
[1] 26


| Excellent job!

  |=================================                                     |  52%
| This tells us that of the 194 flags in our dataset, 153 contain the
| color red, 91 contain green, 99 contain blue, and so on.

...

  |==================================                                    |  54%
| The result is a list, since lapply() always returns a list. Each element
| of this list is of length one, so the result can be simplified to a
| vector by calling sapply() instead of lapply(). Try it now.

//> sapply(flag_colors, sum)
   red  green   blue   gold  white  black orange
   153     91     99     91    146     52     26

| You are amazing!

  |===================================                                   |  56%
| Perhaps it is more informative to find the proportion of flags (out of
| 194) containing each color. Since each column is just a bunch of 1s and
| 0s, the arithmetic mean of each column will give us the proportion of
| 1 s. (If it is not clear why, think of a simpler situation where you have
| three 1s and two 0s -- (1 + 1 + 1 + 0 + 0)/5 = 3/5 = 0.6).

...

  |====================================                                  |  58%
| Use sapply() to apply the mean() function to each column of flag_colors.
| Remember that the second argument to sapply() should just specify the
| name of the function (i.e. mean) that you want to apply.

//> sapply(flag_colors, mean)
      red     green      blue      gold     white     black    orange
0.7886598 0.4690722 0.5103093 0.4690722 0.7525773 0.2680412 0.1340206

| All that practice is paying off!

  |=====================================                                 |  60%
| In the examples we have looked at so far, sapply() has been able to
| simplify the result to vector. That is because each element of the list
| returned by lapply() was a vector of length one. Recall that sapply()
| instead returns a matrix when each element of the list returned by
| lapply() is a vector of the same length (> 1).
```

```
...

  |=======================================                              |  62%
| To illustrate this, let is extract columns 19 through 23 from the flags
| dataset and store the result in a new data frame called flag_shapes.
| flag_shapes <- flags[, 19:23] will do it.

> flag-shapes <- flags[, 19:23]
Error in flag - shapes <- flags[, 19:23] : objeto 'flag' no encontrado
> flag_shapes <- flags[, 19:23]

| You got it!

  |========================================                             |  64%
| Each of these columns (i.e. variables) represents the number of times a
| particular shape or design appears on a country is flag. We are
| interested in the minimum and maximum number of times each shape or
| design appears.

...

  |=========================================                            |  66%
| The range() function returns the minimum and maximum of its first
| argument, which should be a numeric vector. Use lapply() to apply the
| range function to each column of flag_shapes. Do not worry about storing
| the result in a new variable. By now, we know that lapply() always
| returns a list.

> lapply(flag_shapes, range)
$circles
[1] 0 4

$crosses
[1] 0 2

$saltires
[1] 0 1

$quarters
[1] 0 4

$sunstars
[1]  0 50


| That is the answer I was looking for.

  |=========================================                            |  68%
| Do the same operation, but using sapply() and store the result in a
| variable called shape_mat.

> sapply(flag_shapes, range)
     circles crosses saltires quarters sunstars
[1,]       0       0        0        0        0
```

```
[2,]       4       2       1       4      50

| Almost! Try again. Or, type info() for more options.

| shape_mat <- sapply(flag_shapes, range) will apply the range() function
| to each column of flag_shapes and store the result in shape_mat.

> shape_mat <- sapply(flag_shapes, range)

| You are the best!

  |=============================================                     |  70%
| View the contents of shape_mat.

> shape_mat
     circles crosses saltires quarters sunstars
[1,]       0       0        0        0        0
[2,]       4       2        1        4       50

| That is correct!

  |==============================================                    |  72%
| Each column of shape_mat gives the minimum (row 1) and maximum (row 2)
| number of times its respective shape appears in different flags.

...

  |==============================================                    |  74%
| Use the clas() function to confirm that shape_mat is a matrix.

//> class(shape_mat)
[1] "matrix" "array"

| That is a job well done!

  |===============================================                   |  76%
| As we've seen, sapply() always attempts to simplify the result given by
| lapply(). It has been successful in doing so for each of the examples
| we've looked at so far. Let's look at an example where sapply() can't
| figure out how to simplify the result and thus returns a list, no
| different from lapply().

...

  |================================================                  |  78%
| When given a vector, the unique() function returns a vector with all
| duplicate elements removed. In other words, unique() returns a vector of
| only the 'unique' elements. To see how it works, try unique(c(3, 4, 5,
| 5, 5, 6, 6)).

//> unique(c(3, 4, 5, 5, 5, 6, 6 ))
[1] 3 4 5 6

| All that practice is paying off!
```

```
  |=================================================                     |  80%
| We want to know the unique values for each variable in the flags
| dataset. To accomplish this, use lapply() to apply the unique() function
| to each column in the flags dataset, storing the result in a variable
| called unique_vals.

>
> unique_vals <- lapply(flags, unique)

| You are amazing!

  |==================================================                    |  82%
| Print the value of unique_vals to the console.

//> unique_vals
//$name
  [1] "Afghanistan"              "Albania"
  [3] "Algeria"                  "American-Samoa"
  [5] "Andorra"                  "Angola"
  [7] "Anguilla"                 "Antigua-Barbuda"
  [9] "Argentina"                "Argentine"
 [11] "Australia"                "Austria"
 [13] "Bahamas"                  "Bahrain"
 [15] "Bangladesh"               "Barbados"
 [17] "Belgium"                  "Belize"
 [19] "Benin"                    "Bermuda"
 [21] "Bhutan"                   "Bolivia"
 [23] "Botswana"                 "Brazil"
 [25] "British-Virgin-Isles"     "Brunei"
 [27] "Bulgaria"                 "Burkina"
 [29] "Burma"                    "Burundi"
 [31] "Cameroon"                 "Canada"
 [33] "Cape-Verde-Islands"       "Cayman-Islands"
 [35] "Central-African-Republic" "Chad"
 [37] "Chile"                    "China"
 [39] "Colombia"                 "Comorro-Islands"
 [41] "Congo"                    "Cook-Islands"
 [43] "Costa-Rica"               "Cuba"
 [45] "Cyprus"                   "Czechoslovakia"
 [47] "Denmark"                  "Djibouti"
 [49] "Dominica"                 "Dominican-Republic"
 [51] "Ecuador"                  "Egypt"
 [53] "El-Salvador"              "Equatorial-Guinea"
 [55] "Ethiopia"                 "Faeroes"
 [57] "Falklands-Malvinas"       "Fiji"
 [59] "Finland"                  "France"
 [61] "French-Guiana"            "French-Polynesia"
 [63] "Gabon"                    "Gambia"
 [65] "Germany-DDR"              "Germany-FRG"
 [67] "Ghana"                    "Gibraltar"
 [69] "Greece"                   "Greenland"
 [71] "Grenada"                  "Guam"
 [73] "Guatemala"                "Guinea"
```

```
 [75] "Guinea-Bissau"            "Guyana"
 [77] "Haiti"                    "Honduras"
 [79] "Hong-Kong"                 "Hungary"
 [81] "Iceland"                  "India"
 [83] "Indonesia"                "Iran"
 [85] "Iraq"                     "Ireland"
 [87] "Israel"                   "Italy"
 [89] "Ivory-Coast"              "Jamaica"
 [91] "Japan"                    "Jordan"
 [93] "Kampuchea"                "Kenya"
 [95] "Kiribati"                 "Kuwait"
 [97] "Laos"                     "Lebanon"
 [99] "Lesotho"                  "Liberia"
[101] "Libya"                    "Liechtenstein"
[103] "Luxembourg"               "Malagasy"
[105] "Malawi"                   "Malaysia"
[107] "Maldive-Islands"          "Mali"
[109] "Malta"                    "Marianas"
[111] "Mauritania"               "Mauritius"
[113] "Mexico"                   "Micronesia"
[115] "Monaco"                   "Mongolia"
[117] "Montserrat"               "Morocco"
[119] "Mozambique"               "Nauru"
[121] "Nepal"                    "Netherlands"
[123] "Netherlands-Antilles"     "New-Zealand"
[125] "Nicaragua"                "Niger"
[127] "Nigeria"                  "Niue"
[129] "North-Korea"              "North-Yemen"
[131] "Norway"                   "Oman"
[133] "Pakistan"                 "Panama"
[135] "Papua-New-Guinea"         "Parguay"
[137] "Peru"                     "Philippines"
[139] "Poland"                   "Portugal"
[141] "Puerto-Rico"              "Qatar"
[143] "Romania"                  "Rwanda"
[145] "San-Marino"               "Sao-Tome"
[147] "Saudi-Arabia"             "Senegal"
[149] "Seychelles"               "Sierra-Leone"
[151] "Singapore"                "Soloman-Islands"
[153] "Somalia"                  "South-Africa"
[155] "South-Korea"              "South-Yemen"
[157] "Spain"                    "Sri-Lanka"
[159] "St-Helena"                "St-Kitts-Nevis"
[161] "St-Lucia"                 "St-Vincent"
[163] "Sudan"                    "Surinam"
[165] "Swaziland"                "Sweden"
[167] "Switzerland"              "Syria"
[169] "Taiwan"                   "Tanzania"
[171] "Thailand"                 "Togo"
[173] "Tonga"                    "Trinidad-Tobago"
[175] "Tunisia"                  "Turkey"
[177] "Turks-Cocos-Islands"      "Tuvalu"
[179] "UAE"                      "Uganda"
[181] "UK"                       "Uruguay"
```

```
[183] "US-Virgin-Isles"          "USA"
[185] "USSR"                     "Vanuatu"
[187] "Vatican-City"             "Venezuela"
[189] "Vietnam"                  "Western-Samoa"
[191] "Yugoslavia"               "Zaire"
[193] "Zambia"                   "Zimbabwe"

$landmass
[1] 5 3 4 6 1 2

$zone
[1] 1 3 2 4

$area
  [1]   648    29  2388     0  1247  2777  7690    84    19     1   143
 [12]    31    23   113    47  1099   600  8512     6   111   274   678
 [23]    28   474  9976     4   623  1284   757  9561  1139     2   342
 [34]    51   115     9   128    43    22    49   284  1001    21  1222
 [45]    12    18   337   547    91   268    10   108   249   239   132
 [56]  2176   109   246    36   215   112    93   103  3268  1904  1648
 [67]   435    70   301   323    11   372    98   181   583   236    30
 [78]  1760     3   587   118   333  1240  1031  1973  1566   447   783
 [89]   140    41  1267   925   121   195   324   212   804    76   463
[100]   407  1285   300   313    92   237    26  2150   196    72   637
[111]  1221    99   288   505    66  2506    63    17   450   185   945
[122]   514    57     5   164   781   245   178  9363 22402    15   912
[133]   256   905   753   391

$population
 [1]   16    3   20    0    7   28   15    8   90   10    1    6  119    9
[15]   35    4   24    2   11 1008    5   47   31   54   17   61   14  684
[29]  157   39   57  118   13   77   12   56   18   84   48   36   22   29
[43]   38   49   45  231  274   60

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

$religion
[1] 2 6 1 0 5 3 4 7

$bars
[1] 0 2 3 1 5

$stripes
 [1]  3  0  2  1  5  9 11 14  4  6 13  7

$colours
[1] 5 3 2 8 6 4 7 1

$red
[1] 1 0

$green
[1] 1 0
```

```
$blue
[1] 0 1

$gold
[1] 1 0

$white
[1] 1 0

$black
[1] 1 0

$orange
[1] 0 1

$mainhue
[1] "green"  "red"    "blue"   "gold"   "white"  "orange" "black"  "brown"

$circles
[1] 0 1 4 2

$crosses
[1] 0 1 2

$saltires
[1] 0 1

$quarters
[1] 0 1 4

$sunstars
 [1]  1  0  6 22 14  3  4  5 15 10  7  2  9 50

$crescent
[1] 0 1

$triangle
[1] 0 1

$icon
[1] 1 0

$animate
[1] 0 1

$text
[1] 0 1

$topleft
[1] "black"  "red"    "green"  "blue"   "white"  "orange" "gold"

$botright
[1] "green"  "red"    "white"  "black"  "blue"   "gold"   "orange" "brown"
```

```
| Perseverance, that is the answer.

  |=======================================================          |  84%
| Since unique_vals is a list, you can use what you have learned to
| determine the length of each element of unique_vals (i.e. the number of
| unique values for each variable). Simplify the result, if possible.
| Hint: Apply the length() function to each element of unique_vals.

//> sapply(unique_vals, length)
      name   landmass       zone       area population   language
       194          6          4        136         48         10
  religion       bars    stripes    colours        red      green
         8          5         12          8          2          2
      blue       gold      white      black     orange    mainhue
         2          2          2          2          2          8
   circles    crosses   saltires   quarters   sunstars   crescent
         4          3          2          3         14          2
  triangle       icon    animate       text    topleft   botright
         2          2          2          2          7          8

| That is a job well done!

  |=========================================================          |  86%
| The fact that the elements of the unique_vals list are all vectors of
| *different* length poses a problem for sapply(), since there is no
| obvious way of simplifying the result.

...

  |=========================================================          |  88%
| Use sapply() to apply the unique() function to each column of the flags
| dataset to see that you get the same unsimplified list that you got from
| lapply().

//> sapply(flags,unique)
$name
  [1] "Afghanistan"             "Albania"
  [3] "Algeria"                 "American-Samoa"
  [5] "Andorra"                 "Angola"
  [7] "Anguilla"                "Antigua-Barbuda"
  [9] "Argentina"               "Argentine"
 [11] "Australia"               "Austria"
 [13] "Bahamas"                 "Bahrain"
 [15] "Bangladesh"              "Barbados"
 [17] "Belgium"                 "Belize"
 [19] "Benin"                   "Bermuda"
 [21] "Bhutan"                  "Bolivia"
 [23] "Botswana"                "Brazil"
 [25] "British-Virgin-Isles"    "Brunei"
 [27] "Bulgaria"                "Burkina"
 [29] "Burma"                   "Burundi"
 [31] "Cameroon"                "Canada"
```

```
 [33] "Cape-Verde-Islands"       "Cayman-Islands"
 [35] "Central-African-Republic" "Chad"
 [37] "Chile"                    "China"
 [39] "Colombia"                 "Comorro-Islands"
 [41] "Congo"                    "Cook-Islands"
 [43] "Costa-Rica"               "Cuba"
 [45] "Cyprus"                   "Czechoslovakia"
 [47] "Denmark"                  "Djibouti"
 [49] "Dominica"                 "Dominican-Republic"
 [51] "Ecuador"                  "Egypt"
 [53] "El-Salvador"              "Equatorial-Guinea"
 [55] "Ethiopia"                 "Faeroes"
 [57] "Falklands-Malvinas"       "Fiji"
 [59] "Finland"                  "France"
 [61] "French-Guiana"            "French-Polynesia"
 [63] "Gabon"                    "Gambia"
 [65] "Germany-DDR"              "Germany-FRG"
 [67] "Ghana"                    "Gibraltar"
 [69] "Greece"                   "Greenland"
 [71] "Grenada"                  "Guam"
 [73] "Guatemala"                "Guinea"
 [75] "Guinea-Bissau"            "Guyana"
 [77] "Haiti"                    "Honduras"
 [79] "Hong-Kong"                "Hungary"
 [81] "Iceland"                  "India"
 [83] "Indonesia"                "Iran"
 [85] "Iraq"                     "Ireland"
 [87] "Israel"                   "Italy"
 [89] "Ivory-Coast"              "Jamaica"
 [91] "Japan"                    "Jordan"
 [93] "Kampuchea"                "Kenya"
 [95] "Kiribati"                 "Kuwait"
 [97] "Laos"                     "Lebanon"
 [99] "Lesotho"                  "Liberia"
[101] "Libya"                    "Liechtenstein"
[103] "Luxembourg"               "Malagasy"
[105] "Malawi"                   "Malaysia"
[107] "Maldive-Islands"          "Mali"
[109] "Malta"                    "Marianas"
[111] "Mauritania"               "Mauritius"
[113] "Mexico"                   "Micronesia"
[115] "Monaco"                   "Mongolia"
[117] "Montserrat"               "Morocco"
[119] "Mozambique"               "Nauru"
[121] "Nepal"                    "Netherlands"
[123] "Netherlands-Antilles"     "New-Zealand"
[125] "Nicaragua"                "Niger"
[127] "Nigeria"                  "Niue"
[129] "North-Korea"              "North-Yemen"
[131] "Norway"                   "Oman"
[133] "Pakistan"                 "Panama"
[135] "Papua-New-Guinea"         "Parguay"
[137] "Peru"                     "Philippines"
[139] "Poland"                   "Portugal"
```

```
[141] "Puerto-Rico"              "Qatar"
[143] "Romania"                  "Rwanda"
[145] "San-Marino"               "Sao-Tome"
[147] "Saudi-Arabia"             "Senegal"
[149] "Seychelles"               "Sierra-Leone"
[151] "Singapore"                "Soloman-Islands"
[153] "Somalia"                  "South-Africa"
[155] "South-Korea"              "South-Yemen"
[157] "Spain"                    "Sri-Lanka"
[159] "St-Helena"                "St-Kitts-Nevis"
[161] "St-Lucia"                 "St-Vincent"
[163] "Sudan"                    "Surinam"
[165] "Swaziland"                "Sweden"
[167] "Switzerland"              "Syria"
[169] "Taiwan"                   "Tanzania"
[171] "Thailand"                 "Togo"
[173] "Tonga"                    "Trinidad-Tobago"
[175] "Tunisia"                  "Turkey"
[177] "Turks-Cocos-Islands"      "Tuvalu"
[179] "UAE"                      "Uganda"
[181] "UK"                       "Uruguay"
[183] "US-Virgin-Isles"          "USA"
[185] "USSR"                     "Vanuatu"
[187] "Vatican-City"             "Venezuela"
[189] "Vietnam"                  "Western-Samoa"
[191] "Yugoslavia"               "Zaire"
[193] "Zambia"                   "Zimbabwe"

$landmass
[1] 5 3 4 6 1 2

$zone
[1] 1 3 2 4

$area
  [1]   648    29  2388     0  1247  2777  7690    84    19     1   143
 [12]    31    23   113    47  1099   600  8512     6   111   274   678
 [23]    28   474  9976     4   623  1284   757  9561  1139     2   342
 [34]    51   115     9   128    43    22    49   284  1001    21  1222
 [45]    12    18   337   547    91   268    10   108   249   239   132
 [56]  2176   109   246    36   215   112    93   103  3268  1904  1648
 [67]   435    70   301   323    11   372    98   181   583   236    30
 [78]  1760     3   587   118   333  1240  1031  1973  1566   447   783
 [89]   140    41  1267   925   121   195   324   212   804    76   463
[100]   407  1285   300   313    92   237    26  2150   196    72   637
[111]  1221    99   288   505    66  2506    63    17   450   185   945
[122]   514    57     5   164   781   245   178  9363 22402    15   912
[133]   256   905   753   391

$population
 [1]   16    3   20    0    7   28   15    8   90   10    1    6  119    9
[15]   35    4   24    2   11 1008    5   47   31   54   17   61   14  684
[29]  157   39   57  118   13   77   12   56   18   84   48   36   22   29
[43]   38   49   45  231  274   60
```

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

$religion
[1] 2 6 1 0 5 3 4 7

$bars
[1] 0 2 3 1 5

$stripes
 [1]  3  0  2  1  5  9 11 14  4  6 13  7

$colours
[1] 5 3 2 8 6 4 7 1

$red
[1] 1 0

$green
[1] 1 0

$blue
[1] 0 1

$gold
[1] 1 0

$white
[1] 1 0

$black
[1] 1 0

$orange
[1] 0 1

$mainhue
[1] "green"  "red"    "blue"   "gold"   "white"  "orange" "black"  "brown"

$circles
[1] 0 1 4 2

$crosses
[1] 0 1 2

$saltires
[1] 0 1

$quarters
[1] 0 1 4

$sunstars
 [1]  1  0  6 22 14  3  4  5 15 10  7  2  9 50
```

```
$crescent
[1] 0 1

$triangle
[1] 0 1

$icon
[1] 1 0

$animate
[1] 0 1

$text
[1] 0 1

$topleft
[1] "black"  "red"    "green"  "blue"   "white"  "orange" "gold"

$botright
[1] "green"  "red"    "white"  "black"  "blue"   "gold"   "orange" "brown"


| Excellent work!

  |=======================================================       |  90%
| Occasionally, you may need to apply a function that is not yet defined,
| thus requiring you to write your own. Writing functions in R is beyond
| the scope of this lesson, but let is look at a quick example of how you
| might do so in the context of loop functions.

...

  |=========================================================     |  92%
| Pretend you are interested in only the second item from each element of
| the unique_vals list that you just created. Since each element of the
| unique_vals list is a vector and we're not aware of any built-in
| function in R that returns the second element of a vector, we will
| construct our own function.

...

  |===========================================================   |  94%
| lapply(unique_vals, function(elem) elem[2]) will return a list
| containing the second item from each element of the unique_vals list.
| Note that our function takes one argument, elem, which is just a 'dummy
| variable' that takes on the value of each element of unique_vals, in
| turn.

//> lapply(unique_vals, function(elem) elem[2])
$name
[1] "Albania"

$landmass
```

```
[1] 3

$zone
[1] 3

$area
[1] 29

$population
[1] 3

$language
[1] 6

$religion
[1] 6

$bars
[1] 2

$stripes
[1] 0

$colours
[1] 3

$red
[1] 0

$green
[1] 0

$blue
[1] 1

$gold
[1] 0

$white
[1] 0

$black
[1] 0

$orange
[1] 1

$mainhue
[1] "red"

$circles
[1] 1

$crosses
```

```
[1] 1

$saltires
[1] 1

$quarters
[1] 1

$sunstars
[1] 0

$crescent
[1] 1

$triangle
[1] 1

$icon
[1] 0

$animate
[1] 1

$text
[1] 1

$topleft
[1] "red"

$botright
[1] "red"


| You got it right!

  |=======================================================================   |  96%
| The only difference between previous examples and this one is that we
| are defining and using our own function right in the call to lapply().
| Our function has no name and disappears as soon as lapply() is done
| using it. So-called 'anonymous functions' can be very useful when one of
| R is built-in functions is not an option.

...

  |========================================================================= |  98%
| In this lesson, you learned how to use the powerful lapply() and
| sapply() functions to apply an operation over the elements of a list. In
| the next lesson, we'll take a look at some close relatives of lapply()
| and sapply().

...

  |==========================================================================| 100%
| Would you like to receive credit for completing this course on
```

```
| Coursera.org?

1: Yes
2: No
```
V

Swirl 11: Vapply and Tapply

V
```
  |                                                                      |   0%

| In the last lesson, you learned about the two most fundamental members of
| R is *apply family of functions: lapply() and sapply(). Both take a list as
| input, apply a function to each element of the list, then combine and
| return the result. lapply() always returns a list, whereas sapply()
| attempts to simplify the result.

...

  |===                                                                   |   4%
| In this lesson, you will learn how to use vapply() and tapply(), each of
| which serves a very specific purpose within the Split-Apply-Combine
| methodology. For consistency, we'll use the same dataset we used in the
| 'lapply and sapply' lesson.

...

  |=====                                                                 |   8%
| The Flags dataset from the UCI Machine Learning Repository contains
| details of various nations and their flags. More information may be found
//| here: http://archive.ics.uci.edu/ml/datasets/Flags

...

  |========                                                              |  12%
| I have stored the data in a variable called flags. If it is been a while
| since you completed the 'lapply and sapply' lesson, you may want to
| reacquaint yourself with the data by using functions like dim(), head(),
//| str(), and summary() when you return to the prompt (>). You can also type
| viewinfo() at the prompt to bring up some documentation for the dataset.
| Let is get started!

...

  |===========                                                           |  16%
| As you saw in the last lesson, the unique() function returns a vector of
| the unique values contained in the object passed to it. Therefore,
| sapply(flags, unique) returns a list containing one vector of unique
| values for each column of the flags dataset. Try it again now.

//> sapply(flags, unique)
$name
  [1] "Afghanistan"              "Albania"
```

```
  [3] "Algeria"                   "American-Samoa"
  [5] "Andorra"                   "Angola"
  [7] "Anguilla"                  "Antigua-Barbuda"
  [9] "Argentina"                 "Argentine"
 [11] "Australia"                 "Austria"
 [13] "Bahamas"                   "Bahrain"
 [15] "Bangladesh"                "Barbados"
 [17] "Belgium"                   "Belize"
 [19] "Benin"                     "Bermuda"
 [21] "Bhutan"                    "Bolivia"
 [23] "Botswana"                  "Brazil"
 [25] "British-Virgin-Isles"      "Brunei"
 [27] "Bulgaria"                  "Burkina"
 [29] "Burma"                     "Burundi"
 [31] "Cameroon"                  "Canada"
 [33] "Cape-Verde-Islands"        "Cayman-Islands"
 [35] "Central-African-Republic" "Chad"
 [37] "Chile"                     "China"
 [39] "Colombia"                  "Comorro-Islands"
 [41] "Congo"                     "Cook-Islands"
 [43] "Costa-Rica"                "Cuba"
 [45] "Cyprus"                    "Czechoslovakia"
 [47] "Denmark"                   "Djibouti"
 [49] "Dominica"                  "Dominican-Republic"
 [51] "Ecuador"                   "Egypt"
 [53] "El-Salvador"               "Equatorial-Guinea"
 [55] "Ethiopia"                  "Faeroes"
 [57] "Falklands-Malvinas"        "Fiji"
 [59] "Finland"                   "France"
 [61] "French-Guiana"             "French-Polynesia"
 [63] "Gabon"                     "Gambia"
 [65] "Germany-DDR"               "Germany-FRG"
 [67] "Ghana"                     "Gibraltar"
 [69] "Greece"                    "Greenland"
 [71] "Grenada"                   "Guam"
 [73] "Guatemala"                 "Guinea"
 [75] "Guinea-Bissau"             "Guyana"
 [77] "Haiti"                     "Honduras"
 [79] "Hong-Kong"                 "Hungary"
 [81] "Iceland"                   "India"
 [83] "Indonesia"                 "Iran"
 [85] "Iraq"                      "Ireland"
 [87] "Israel"                    "Italy"
 [89] "Ivory-Coast"               "Jamaica"
 [91] "Japan"                     "Jordan"
 [93] "Kampuchea"                 "Kenya"
 [95] "Kiribati"                  "Kuwait"
 [97] "Laos"                      "Lebanon"
 [99] "Lesotho"                   "Liberia"
[101] "Libya"                     "Liechtenstein"
[103] "Luxembourg"                "Malagasy"
[105] "Malawi"                    "Malaysia"
[107] "Maldive-Islands"           "Mali"
[109] "Malta"                     "Marianas"
```

```
[111] "Mauritania"               "Mauritius"
[113] "Mexico"                   "Micronesia"
[115] "Monaco"                   "Mongolia"
[117] "Montserrat"               "Morocco"
[119] "Mozambique"               "Nauru"
[121] "Nepal"                    "Netherlands"
[123] "Netherlands-Antilles"     "New-Zealand"
[125] "Nicaragua"                "Niger"
[127] "Nigeria"                  "Niue"
[129] "North-Korea"              "North-Yemen"
[131] "Norway"                   "Oman"
[133] "Pakistan"                 "Panama"
[135] "Papua-New-Guinea"         "Parguay"
[137] "Peru"                     "Philippines"
[139] "Poland"                   "Portugal"
[141] "Puerto-Rico"              "Qatar"
[143] "Romania"                  "Rwanda"
[145] "San-Marino"               "Sao-Tome"
[147] "Saudi-Arabia"             "Senegal"
[149] "Seychelles"               "Sierra-Leone"
[151] "Singapore"                "Soloman-Islands"
[153] "Somalia"                  "South-Africa"
[155] "South-Korea"              "South-Yemen"
[157] "Spain"                    "Sri-Lanka"
[159] "St-Helena"                "St-Kitts-Nevis"
[161] "St-Lucia"                 "St-Vincent"
[163] "Sudan"                    "Surinam"
[165] "Swaziland"                "Sweden"
[167] "Switzerland"              "Syria"
[169] "Taiwan"                   "Tanzania"
[171] "Thailand"                 "Togo"
[173] "Tonga"                    "Trinidad-Tobago"
[175] "Tunisia"                  "Turkey"
[177] "Turks-Cocos-Islands"      "Tuvalu"
[179] "UAE"                      "Uganda"
[181] "UK"                       "Uruguay"
[183] "US-Virgin-Isles"          "USA"
[185] "USSR"                     "Vanuatu"
[187] "Vatican-City"             "Venezuela"
[189] "Vietnam"                  "Western-Samoa"
[191] "Yugoslavia"               "Zaire"
[193] "Zambia"                   "Zimbabwe"

$landmass
[1] 5 3 4 6 1 2

$zone
[1] 1 3 2 4

$area
  [1]   648    29  2388     0  1247  2777  7690    84    19     1   143    31
 [13]    23   113    47  1099   600  8512     6   111   274   678    28   474
 [25]  9976     4   623  1284   757  9561  1139     2   342    51   115     9
 [37]   128    43    22    49   284  1001    21  1222    12    18   337   547
```

```
 [49]    91   268    10   108   249   239   132  2176   109   246    36   215
 [61]   112    93   103  3268  1904  1648   435    70   301   323    11   372
 [73]    98   181   583   236    30  1760     3   587   118   333  1240  1031
 [85]  1973  1566   447   783   140    41  1267   925   121   195   324   212
 [97]   804    76   463   407  1285   300   313    92   237    26  2150   196
[109]    72   637  1221    99   288   505    66  2506    63    17   450   185
[121]   945   514    57     5   164   781   245   178  9363 22402    15   912
[133]   256   905   753   391

$population
 [1]   16    3   20    0    7   28   15    8   90   10    1    6  119    9
[15]   35    4   24    2   11 1008    5   47   31   54   17   61   14  684
[29]  157   39   57  118   13   77   12   56   18   84   48   36   22   29
[43]   38   49   45  231  274   60

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

$religion
[1] 2 6 1 0 5 3 4 7

$bars
[1] 0 2 3 1 5

$stripes
 [1]  3  0  2  1  5  9 11 14  4  6 13  7

$colours
[1] 5 3 2 8 6 4 7 1

$red
[1] 1 0

$green
[1] 1 0

$blue
[1] 0 1

$gold
[1] 1 0

$white
[1] 1 0

$black
[1] 1 0

$orange
[1] 0 1

$mainhue
[1] "green"  "red"    "blue"   "gold"   "white"  "orange" "black"  "brown"
```

```
$circles
[1] 0 1 4 2

$crosses
[1] 0 1 2

$saltires
[1] 0 1

$quarters
[1] 0 1 4

$sunstars
 [1]  1  0  6 22 14  3  4  5 15 10  7  2  9 50

$crescent
[1] 0 1

$triangle
[1] 0 1

$icon
[1] 1 0

$animate
[1] 0 1

$text
[1] 0 1

$topleft
[1] "black"  "red"    "green"  "blue"   "white"  "orange" "gold"

$botright
[1] "green"  "red"    "white"  "black"  "blue"   "gold"   "orange" "brown"


| That is a job well done!

  |=============                                                         |  20%
| What if you had forgotten how unique() works and mistakenly thought it
| returns the *number* of unique values contained in the object passed to
| it? Then you might have incorrectly expected sapply(flags, unique) to
| return a numeric vector, since each element of the list returned would
| contain a single number and sapply() could then simplify the result to a
| vector.
...

  |================                                                      |  24%
| When working interactively (at the prompt), this is not much of a problem,
| since you see the result immediately and will quickly recognize your
| mistake. However, when working non-interactively (e.g. writing your own
| functions), a misunderstanding may go undetected and cause incorrect
| results later on. Therefore, you may wish to be more careful and that is
```

```
| where vapply() is useful.

...

  |==================                                                    |  28%
| Whereas sapply() tries to 'guess' the correct format of the result,
| vapply() allows you to specify it explicitly. If the result does not match
| the format you specify, vapply() will throw an error, causing the
| operation to stop. This can prevent significant problems in your code that
| might be caused by getting unexpected return values from sapply().

...

  |=====================                                                 |  32%
| Try vapply(flags, unique, numeric(1)), which says that you expect each
| element of the result to be a numeric vector of length 1. Since this is
| NOT actually the case, YOU WILL GET AN ERROR. Once you get the error, type
| ok() to continue to the next question.

> vapply(flags, unique, numeric(1))
//Error in vapply(flags, unique, numeric(1)) :
//  Los valores deben ser de longitud 1,
//pero el resultado FUN(X [[1]]) es la longitud 194
//> ok()

| That is correct!

  |========================                                              |  36%
| Recall from the previous lesson that sapply(flags, class) will return a
| character vector containing the class of each column in the dataset. Try
| that again now to see the result.

> sapply(flags, class)
       name    landmass        zone        area  population    language
"character"   "integer"   "integer"   "integer"   "integer"   "integer"
   religion        bars     stripes     colours         red       green
  "integer"   "integer"   "integer"   "integer"   "integer"   "integer"
       blue        gold       white       black      orange     mainhue
  "integer"   "integer"   "integer"   "integer"   "integer" "character"
    circles     crosses    saltires    quarters    sunstars    crescent
  "integer"   "integer"   "integer"   "integer"   "integer"   "integer"
   triangle        icon     animate        text     topleft    botright
  "integer"   "integer"   "integer" "character" "character" "character"

| Keep working like that and you will get there!

  |===========================                                           |  40%
| If we wish to be explicit about the format of the result we expect, we can
| use vapply(flags, class, character(1)). The 'character(1)' argument tells
| R that we expect the class function to return a character vector of length
| 1 when applied to EACH column of the flags dataset. Try it now.

> vapply(flags, class, character(1))
       name    landmass        zone        area  population    language
```

```
"character"   "integer"   "integer"   "integer"   "integer"   "integer"
   religion        bars     stripes     colours         red       green
  "integer"   "integer"   "integer"   "integer"   "integer"   "integer"
       blue        gold       white       black      orange     mainhue
  "integer"   "integer"   "integer"   "integer"   "integer" "character"
    circles     crosses    saltires    quarters    sunstars    crescent
  "integer"   "integer"   "integer"   "integer"   "integer"   "integer"
   triangle        icon     animate        text     topleft    botright
  "integer"   "integer"   "integer"   "integer" "character" "character"

| Keep up the great work!

  |============================                                          |  44%
| Note that since our expectation was correct (i.e. character(1)), the
| vapply() result is identical to the sapply() result -- a character vector
| of column classes.

...

  |================================                                      |  48%
| You might think of vapply() as being 'safer' than sapply(), since it
| requires you to specify the format of the output in advance, instead of
| just allowing R to 'guess' what you wanted. In addition, vapply() may
| perform faster than sapply() for large datasets. However, when doing data
| analysis interactively (at the prompt), sapply() saves you some typing and
| will often be good enough.

...

  |====================================                                  |  52%
| As a data analyst, you will often wish to split your data up into groups
| based on the value of some variable, then apply a function to the members
| of each group. The next function we'll look at, tapply(), does exactly
| that.

...

  |========================================                              |  56%
| Use ?tapply to pull up the documentation.

> ?tapply

| That is a job well done!

  |============================================                          |  60%
| The 'landmass' variable in our dataset takes on integer values between 1
| and 6, each of which represents a different part of the world. Use
| table(flags$landmass) to see how many flags/countries fall into each
| group.

> table(flags$landmass)

 1  2  3  4  5  6
31 17 35 52 39 20
```

```
| That is a job well done!

  |=========================================                              |  64%
| The 'animate' variable in our dataset takes the value 1 if a country is
| flag contains an animate image (e.g. an eagle, a tree, a human hand) and 0
| otherwise. Use table(flags$animate) to see how many flags contain an
| animate image.

> table(flags$animate)

  0   1
155  39

| Nice work!

  |==========================================                             |  68%
| This tells us that 39 flags contain an animate object (animate = 1) and
| 155 do not (animate = 0).

...

  |==============================================                         |  72%
| If you take the arithmetic mean of a bunch of 0s and 1s, you get the
| proportion of 1s. Use tapply(flags$animate, flags$landmass, mean) to apply
| the mean function to the 'animate' variable separately for each of the six
| landmass groups, thus giving us the proportion of flags containing an
| animate image WITHIN each landmass group.

> tapply(flags$animate, flags$landmass)
  [1] 5 3 4 6 3 4 1 1 2 2 6 3 1 5 5 1 3 1 4 1 5 2 4 2 1 5 3 4 5 4 4 1 4 1 4 4
 [37] 2 5 2 4 4 6 1 1 3 3 3 4 1 1 2 4 1 4 4 3 2 6 3 3 2 6 4 4 3 3 4 3 3 1 1 6
 [73] 1 4 4 2 1 1 5 3 3 5 6 5 5 3 5 3 4 1 5 5 5 4 6 5 5 5 4 4 4 3 3 4 4 5 5 4
[109] 3 6 4 4 1 6 3 5 1 4 4 6 5 3 1 6 1 4 4 6 5 5 3 5 5 2 6 2 2 6 3 3 1 5 3 4
[145] 3 4 5 4 4 4 5 6 4 4 5 5 3 5 4 1 1 1 4 2 4 3 3 5 5 4 5 4 6 2 4 5 1 6 5 4
[181] 3 2 1 1 5 6 3 2 5 6 3 4 4 4

| You are close...I can feel it! Try it again. Or, type info() for more
| options.

| tapply(flags$animate, flags$landmass, mean) will tell us the proportion of
| flags containing an animate image within each landmass group.

> tapply(flags$animate, flags$landmass, mean)
        1         2         3         4         5         6
0.4193548 0.1764706 0.1142857 0.1346154 0.1538462 0.3000000

| Great job!

  |================================================                       |  76%
| The first landmass group (landmass = 1) corresponds to North America and
| contains the highest proportion of flags with an animate image (0.4194).

...
```

```
  |=======================================================               |  80%
| Similarly, we can look at a summary of population values (in round
| millions) for countries with and without the color red on their flag with
| tapply(flags$population, flags$red, summary).

> tapply(flags$population, flags$red, summary)
$`0`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    0.00    3.00   27.63    9.00  684.00

$`1`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0     0.0     4.0    22.1    15.0  1008.0


| That is correct!

  |=========================================================             |  84%
| What is the median population (in millions) for countries *without* the
| color red on their flag?

1: 22.1
2: 4.0
3: 0.0
4: 9.0
5: 27.6
6: 3.0

//Selection: 1

//| You are close...I can feel it! Try it again.

//| Use your result from the last question. Recall that red = 0 means that the
//| color red is NOT present on a countries flag.

1: 0.0
2: 27.6
3: 9.0
4: 22.1
5: 4.0
6: 3.0

Selection: 1

| Give it another try.

| Use your result from the last question. Recall that red = 0 means that the
| color red is NOT present on a countries flag.

1: 22.1
2: 3.0
3: 0.0
4: 4.0
```

```
5: 27.6
6: 9.0

Selection: 2

| Keep working like that and you will get there!

  |=============================================================        |  88%
| Lastly, use the same approach to look at a summary of population values
| for each of the six landmasses.

//> summary(flags)
     name              landmass         zone            area        
 Length:194         Min.   :1.000   Min.   :1.000   Min.   :    0.0  
 Class :character   1st Qu.:3.000   1st Qu.:1.000   1st Qu.:    9.0  
 Mode  :character   Median :4.000   Median :2.000   Median :  111.0  
                    Mean   :3.572   Mean   :2.211   Mean   :  700.0  
                    3rd Qu.:5.000   3rd Qu.:4.000   3rd Qu.:  471.2  
                    Max.   :6.000   Max.   :4.000   Max.   :22402.0  
   population        language        religion          bars       
 Min.   :   0.00   Min.   : 1.00   Min.   :0.000   Min.   :0.0000  
 1st Qu.:   0.00   1st Qu.: 2.00   1st Qu.:1.000   1st Qu.:0.0000  
 Median :   4.00   Median : 6.00   Median :1.000   Median :0.0000  
 Mean   :  23.27   Mean   : 5.34   Mean   :2.191   Mean   :0.4536  
 3rd Qu.:  14.00   3rd Qu.: 9.00   3rd Qu.:4.000   3rd Qu.:0.0000  
 Max.   :1008.00   Max.   :10.00   Max.   :7.000   Max.   :5.0000  
    stripes          colours           red             green       
 Min.   : 0.000   Min.   :1.000   Min.   :0.0000   Min.   :0.0000  
 1st Qu.: 0.000   1st Qu.:3.000   1st Qu.:1.0000   1st Qu.:0.0000  
 Median : 0.000   Median :3.000   Median :1.0000   Median :0.0000  
 Mean   : 1.552   Mean   :3.464   Mean   :0.7887   Mean   :0.4691  
 3rd Qu.: 3.000   3rd Qu.:4.000   3rd Qu.:1.0000   3rd Qu.:1.0000  
 Max.   :14.000   Max.   :8.000   Max.   :1.0000   Max.   :1.0000  
      blue             gold            white            black       
 Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.000  
 1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:1.0000   1st Qu.:0.000  
 Median :1.0000   Median :0.0000   Median :1.0000   Median :0.000  
 Mean   :0.5103   Mean   :0.4691   Mean   :0.7526   Mean   :0.268  
 3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:1.000  
 Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.000  
     orange         mainhue             circles          crosses       
 Min.   :0.000   Length:194         Min.   :0.0000   Min.   :0.0000  
 1st Qu.:0.000   Class :character   1st Qu.:0.0000   1st Qu.:0.0000  
 Median :0.000   Mode  :character   Median :0.0000   Median :0.0000  
 Mean   :0.134                      Mean   :0.1701   Mean   :0.1495  
 3rd Qu.:0.000                      3rd Qu.:0.0000   3rd Qu.:0.0000  
 Max.   :1.000                      Max.   :4.0000   Max.   :2.0000  
    saltires          quarters         sunstars         crescent       
 Min.   :0.00000   Min.   :0.0000   Min.   : 0.000   Min.   :0.0000  
 1st Qu.:0.00000   1st Qu.:0.0000   1st Qu.: 0.000   1st Qu.:0.0000  
 Median :0.00000   Median :0.0000   Median : 0.000   Median :0.0000  
 Mean   :0.09278   Mean   :0.1495   Mean   : 1.387   Mean   :0.0567  
 3rd Qu.:0.00000   3rd Qu.:0.0000   3rd Qu.: 1.000   3rd Qu.:0.0000  
 Max.   :1.00000   Max.   :4.0000   Max.   :50.000   Max.   :1.0000  
```

```
    triangle           icon           animate           text       
 Min.   :0.0000   Min.   :0.0000   Min.   :0.000   Min.   :0.00000  
 1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.000   1st Qu.:0.00000  
 Median :0.0000   Median :0.0000   Median :0.000   Median :0.00000  
 Mean   :0.1392   Mean   :0.2526   Mean   :0.201   Mean   :0.08247  
 3rd Qu.:0.0000   3rd Qu.:0.7500   3rd Qu.:0.000   3rd Qu.:0.00000  
 Max.   :1.0000   Max.   :1.0000   Max.   :1.000   Max.   :1.00000  
   topleft            botright        
 Length:194         Length:194        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
```

```
| You almost had it, but not quite. Try again. Or, type info() for more
| options.

| You can see a summary of populations for each of the six landmasses by
| calling tapply() with three arguments: flags$population, flags$landmass,
| and summary.

//> tapply(flags$population, flags$landmass, summary)
$`1`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    0.00    0.00   12.29    4.50  231.00 

$`2`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    1.00    6.00   15.71   15.00  119.00 

$`3`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    0.00    8.00   13.86   16.00   61.00 

$`4`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  0.000   1.000   5.000   8.788   9.750  56.000 

$`5`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    2.00   10.00   69.18   39.00 1008.00 

$`6`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    0.00    0.00   11.30    1.25  157.00 


| You are really on a roll!

  |=============================================================     |  92%
| What is the maximum population (in millions) for the fourth landmass group
| (Africa)?
```

```
1: 157.00
2: 56.00
3: 119.0
4: 1010.0
5: 5.00

Selection: 2

| Excellent work!

  |======================================================================   |  96%
| In this lesson, you learned how to use vapply() as a safer alternative to
| sapply(), which is most helpful when writing your own functions. You also
| learned how to use tapply() to split your data into groups based on the
| value of some variable, then apply a function to each group. These
| functions will come in handy on your quest to become a better data
| analyst.

...

  |======================================================================| 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: No
2: Yes
```

V

Swirl 12: Looking at data

A

```
  |                                                                      |   0%

| Whenever you are working with a new dataset, the first thing you should
| do is look at it! What is the format of the data? What are the
| dimensions? What are the variable names? How are the variables stored?
| Are there missing data? Are there any flaws in the data?

...

  |===                                                                   |   4%
| This lesson will teach you how to answer these questions and more using
| R is builtin functions. We will be using a dataset constructed from the
| United States Department of Agriculture's PLANTS Database
| (http://plants.usda.gov/adv_search.html).

...

  |=====                                                                 |   8%
| I have stored the data for you in a variable called plants. Type ls() to
| list the variables in your workspace, among which should be plants.

> ls()
```

```
 [1] "c"               "d"               "d1"
 [4] "d2"              "e"               "e2"
 [7] "f"               "flags"           "grn"
[10] "grn2"            "hi"              "hilbert"
[13] "i"               "iris"            "log.mu"
[16] "m4"              "makeVector"      "mtcars"
[19] "n"               "ok"              "plants"
[22] "printmessage"    "printmessage2"   "r"
[25] "sample.interval" "sm"              "sp2"
[28] "t1"              "t2"              "t3"
[31] "t4"              "viewinfo"        "x"
[34] "x2"              "x3"              "xst"
[37] "y"               "y2"              "y3"

| Keep working like that and you will get there!

  |========                                                              |  12%
| Let is begin by checking the class of the plants variable with
| class(plants). This will give us a clue as to the overall structure of
| the data.

> class(plants)
[1] "data.frame"

| You are doing so well!

  |==========                                                            |  16%
| It is very common for data to be stored in a data frame. It is the
| default class for data read into R using functions like read.csv() and
| read.table(), which you will learn about in another lesson.

...

  |=============                                                         |  20%
| Since the dataset is stored in a data frame, we know it is rectangular.
| In other words, it has two dimensions (rows and columns) and fits neatly
| into a table or spreadsheet. Use dim(plants) to see exactly how many
| rows and columns we're dealing with.

> dim(plants)
[1] 5166   10

| You are quite good my friend!

  |================                                                      |  24%
| The first number you see (5166) is the number of rows (observations) and
| the second number (10) is the number of columns (variables).

...

  |==================                                                    |  28%
| You can also use nrow(plants) to see only the number of rows. Try it
| out.
```

```
> nrow(plants)
[1] 5166

| Your dedication is inspiring!

  |======================                                          |  32%
| ... And ncol(plants) to see only the number of columns.

> ncol(plants)
[1] 10

| You are amazing!

  |========================                                        |  36%
| If you are curious as to how much space the dataset is occupying in
| memory, you can use object.size(plants).

> object.size(plants)
745944 bytes

| You are quite good my friend!

  |============================                                    |  40%
| Now that we have a sense of the shape and size of the dataset, let's get
| a feel for what's inside. names(plants) will return a character vector
| of column (i.e. variable) names. Give it a shot.

> names(plants)
 [1] "Scientific_Name"      "Duration"             "Active_Growth_Period"
 [4] "Foliage_Color"        "pH_Min"               "pH_Max"
 [7] "Precip_Min"           "Precip_Max"           "Shade_Tolerance"
[10] "Temp_Min_F"

| Excellent work!

  |================================                                |  44%
| We've applied fairly descriptive variable names to this dataset, but
| that wont always be the case. A logical next step is to peek at the
| actual data. However, our dataset contains over 5000 observations
| (rows), so it is impractical to view the whole thing all at once.

...

  |==================================                              |  48%
| The head() function allows you to preview the top of the dataset. Give
| it a try with only one argument.

> head(plans)
Error in head(plans) : objeto 'plans' no encontrado
> head(plants)
              Scientific_Name          Duration Active_Growth_Period
1                  Abelmoschus              <NA>                 <NA>
2       Abelmoschus esculentus Annual, Perennial                 <NA>
3                      Abies              <NA>                 <NA>
```

```
4                Abies balsamea         Perennial    Spring and Summer
5 Abies balsamea var. balsamea         Perennial                  <NA>
6                     Abutilon              <NA>                  <NA>
  Foliage_Color pH_Min pH_Max Precip_Min Precip_Max Shade_Tolerance
1          <NA>     NA     NA         NA         NA             <NA>
2          <NA>     NA     NA         NA         NA             <NA>
3          <NA>     NA     NA         NA         NA             <NA>
4         Green      4      6         13         60        Tolerant
5          <NA>     NA     NA         NA         NA             <NA>
6          <NA>     NA     NA         NA         NA             <NA>
  Temp_Min_F
1         NA
2         NA
3         NA
4        -43
5         NA
6         NA

| That is a job well done!

  |=================================                                     |  52%
| Take a minute to look through and understand the output above. Each row
| is labeled with the observation number and each column with the variable
| name. Your screen is probably not wide enough to view all 10 columns
| side-by-side, in which case R displays as many columns as it can on each
| line before continuing on the next.

...

  |==================================                                    |  56%
| By default, head() shows you the first six rows of the data. You can
| alter this behavior by passing as a second argument the number of rows
| you would like to view. Use head() to preview the first 10 rows of plants.

> head(plants,10
+ )
                    Scientific_Name          Duration
1                         Abelmoschus              <NA>
2              Abelmoschus esculentus Annual, Perennial
3                             Abies              <NA>
4                     Abies balsamea         Perennial
5       Abies balsamea var. balsamea         Perennial
6                           Abutilon              <NA>
7               Abutilon theophrasti            Annual
8                            Acacia              <NA>
9                   Acacia constricta         Perennial
10 Acacia constricta var. constricta         Perennial
   Active_Growth_Period Foliage_Color pH_Min pH_Max Precip_Min Precip_Max
1                  <NA>          <NA>     NA     NA         NA         NA
2                  <NA>          <NA>     NA     NA         NA         NA
3                  <NA>          <NA>     NA     NA         NA         NA
4     Spring and Summer         Green      4    6.0         13         60
5                  <NA>          <NA>     NA     NA         NA         NA
6                  <NA>          <NA>     NA     NA         NA         NA
```

```
7                 <NA>          <NA>     NA     NA          NA          NA
8                 <NA>          <NA>     NA     NA          NA          NA
9     Spring and Summer         Green      7    8.5           4          20
10                <NA>          <NA>     NA     NA          NA          NA
   Shade_Tolerance Temp_Min_F
1             <NA>         NA
2             <NA>         NA
3             <NA>         NA
4         Tolerant        -43
5             <NA>         NA
6             <NA>         NA
7             <NA>         NA
8             <NA>         NA
9       Intolerant        -13
10            <NA>         NA

| Excellent work!

  |======================================                                |  60%
| The same applies for using tail() to preview the end of the dataset. Use
| tail() to view the last 15 rows.

> tail(plants,15)
                     Scientific_Name  Duration Active_Growth_Period
5152                         Zizania      <NA>                 <NA>
5153                  Zizania aquatica    Annual               Spring
5154   Zizania aquatica var. aquatica    Annual                 <NA>
5155                 Zizania palustris    Annual                 <NA>
5156 Zizania palustris var. palustris    Annual                 <NA>
5157                        Zizaniopsis      <NA>                 <NA>
5158             Zizaniopsis miliacea Perennial    Spring and Summer
5159                           Zizia      <NA>                 <NA>
5160                     Zizia aptera Perennial                 <NA>
5161                      Zizia aurea Perennial                 <NA>
5162                  Zizia trifoliata Perennial                 <NA>
5163                          Zostera      <NA>                 <NA>
5164                    Zostera marina Perennial                 <NA>
5165                           Zoysia      <NA>                 <NA>
5166                   Zoysia japonica Perennial                 <NA>
     Foliage_Color pH_Min pH_Max Precip_Min Precip_Max Shade_Tolerance
5152          <NA>     NA     NA         NA         NA             <NA>
5153         Green    6.4    7.4         30         50      Intolerant
5154          <NA>     NA     NA         NA         NA             <NA>
5155          <NA>     NA     NA         NA         NA             <NA>
5156          <NA>     NA     NA         NA         NA             <NA>
5157          <NA>     NA     NA         NA         NA             <NA>
5158         Green    4.3    9.0         35         70      Intolerant
5159          <NA>     NA     NA         NA         NA             <NA>
5160          <NA>     NA     NA         NA         NA             <NA>
5161          <NA>     NA     NA         NA         NA             <NA>
5162          <NA>     NA     NA         NA         NA             <NA>
5163          <NA>     NA     NA         NA         NA             <NA>
5164          <NA>     NA     NA         NA         NA             <NA>
5165          <NA>     NA     NA         NA         NA             <NA>
```

```
5166          <NA>      NA      NA          NA          NA             <NA>
     Temp_Min_F
5152         NA
5153         32
5154         NA
5155         NA
5156         NA
5157         NA
5158         12
5159         NA
5160         NA
5161         NA
5162         NA
5163         NA
5164         NA
5165         NA
5166         NA

| You are amazing!

  |=========================================                             |  64%
| After previewing the top and bottom of the data, you probably noticed
| lots of NAs, which are R is placeholders for missing values. Use
| summary(plants) to get a better feel for how each variable is
| distributed and how much of the dataset is missing.

> summary(plants)
 Scientific_Name      Duration         Active_Growth_Period
 Length:5166        Length:5166        Length:5166
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character




 Foliage_Color          pH_Min          pH_Max         Precip_Min
 Length:5166        Min.   :3.000   Min.   : 5.100   Min.   : 4.00
 Class :character   1st Qu.:4.500   1st Qu.: 7.000   1st Qu.:16.75
 Mode  :character   Median :5.000   Median : 7.300   Median :28.00
                    Mean   :4.997   Mean   : 7.344   Mean   :25.57
                    3rd Qu.:5.500   3rd Qu.: 7.800   3rd Qu.:32.00
                    Max.   :7.000   Max.   :10.000   Max.   :60.00
                    NA's   :4327    NA's   :4327     NA's   :4338
   Precip_Max     Shade_Tolerance      Temp_Min_F
 Min.   : 16.00   Length:5166        Min.   :-79.00
 1st Qu.: 55.00   Class :character   1st Qu.:-38.00
 Median : 60.00   Mode  :character   Median :-33.00
 Mean   : 58.73                      Mean   :-22.53
 3rd Qu.: 60.00                      3rd Qu.:-18.00
 Max.   :200.00                      Max.   : 52.00
 NA's   :4338                        NA's   :4328

| You are doing so well!
```

```
  |=========================================                |  68%
| summary() provides different output for each variable, depending on its
| class. For numeric data such as Precip_Min, summary() displays the
| minimum, 1st quartile, median, mean, 3rd quartile, and maximum. These
| values help us understand how the data are distributed.

...

  |=============================================            |  72%
| For categorical variables (called 'factor' variables in R), summary()
| displays the number of times each value (or 'level') occurs in the data.
| For example, each value of Scientific_Name only appears once, since it
| is unique to a specific plant. In contrast, the summary for Duration
| (also a factor variable) tells us that our dataset contains 3031
| Perennial plants, 682 Annual plants, etc.

...

  |=================================================        |  76%
| You can see that R truncated the summary for Active_Growth_Period by
| including a catch-all category called 'Other'. Since it is a
| categorical/factor variable, we can see how many times each value
| actually occurs in the data with table(plants$Active_Growth_Period).

> table(plants$Active_Growth_Period)

Fall, Winter and Spring                  Spring         Spring and Fall
                     15                     144                      10
      Spring and Summer    Spring, Summer, Fall                  Summer
                    447                      95                      92
        Summer and Fall             Year Round
                     24                       5

| That is correct!

  |=====================================================    |  80%
| Each of the functions we've introduced so far has its place in helping
| you to better understand the structure of your data. However, we've left
| the best for last....

...

  |=========================================================|  84%
| Perhaps the most useful and concise function for understanding the
| *str*ucture of your data is str(). Give it a try now.

> str(plants)
'data.frame': 5166 obs. of  10 variables:
 $ Scientific_Name     : chr  "Abelmoschus" "Abelmoschus esculentus" "Abies" "A
bies balsamea" ...
 $ Duration            : chr  NA "Annual, Perennial" NA "Perennial" ...
 $ Active_Growth_Period: chr  NA NA NA "Spring and Summer" ...
 $ Foliage_Color       : chr  NA NA NA "Green" ...
 $ pH_Min              : num  NA NA NA 4 NA NA NA NA 7 NA ...
```

```
 $ pH_Max              : num  NA NA NA 6 NA NA NA NA 8.5 NA ...
 $ Precip_Min          : int  NA NA NA 13 NA NA NA NA 4 NA ...
 $ Precip_Max          : int  NA NA NA 60 NA NA NA NA 20 NA ...
 $ Shade_Tolerance     : chr  NA NA NA "Tolerant" ...
 $ Temp_Min_F          : int  NA NA NA -43 NA NA NA NA -13 NA ...

| Your dedication is inspiring!

  |=============================================================         |  88%
| The beauty of str() is that it combines many of the features of the
| other functions you have already seen, all in a concise and readable
| format. At the very top, it tells us that the class of plants is
| 'data.frame' and that it has 5166 observations and 10 variables. It then
| gives us the name and class of each variable, as well as a preview of
| its contents.

...

  |==============================================================      |  92%
| str() is actually a very general function that you can use on most
| objects in R. Any time you want to understand the structure of something
| (a dataset, function, etc.), str() is a good place to start.

...

  |===============================================================     |  96%
| In this lesson, you learned how to get a feel for the structure and
| contents of a new dataset using a collection of simple and useful
| functions. Taking the time to do this upfront can save you time and
| frustration later on in your analysis.

...

  |================================================================| 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: No
2: Yes

Selection: 2
```

C

Swirl 13: Simulation

F

```
Selection: 13

  |                                                                      |   0%

| One of the great advantages of using a statistical programming language
| like R is its vast collection of tools for simulating random numbers.
```

```
...

  |==                                                                |   3%
| This lesson assumes familiarity with a few common probability
| distributions, but these topics will only be discussed with respect to
| random number generation. Even if you have no prior experience with
| these concepts, you should be able to complete the lesson and understand
| the main ideas.

...

  |====                                                              |   6%
| The first function we'll use to generate random numbers is sample(). Use
| ?sample to pull up the documentation.

> ?sample

| That's correct!

  |======                                                            |   9%
| Let's simulate rolling four six-sided dice: sample(1:6, 4, replace =
| TRUE).

> sample(1:6, 4, replace = TRUE)
[1] 1 4 1 1

| That is correct!

  |========                                                          |  12%
| Now repeat the command to see how your result differs. (The probability
| of rolling the exact same result is (1/6)^4 = 0.00077, which is pretty
| small!)

> sample(1:6, 4, replace = FALSE)
[1] 5 4 6 2

| Nice try, but that is not exactly what I was hoping for. Try again. Or,
| type info() for more options.

| Type sample(1:6, 4, replace = TRUE) to simulate rolling four six-sided
| dice again.

> sample(1:6, 4, replace = TRUE)
[1] 4 4 1 6

| Your dedication is inspiring!

  |==========                                                        |  15%
| sample(1:6, 4, replace = TRUE) instructs R to randomly select four
| numbers between 1 and 6, WITH replacement. Sampling with replacement
| simply means that each number is "replaced" after it is selected, so
| that the same number can show up more than once. This is what we want
| here, since what you roll on one die should not affect what you roll on
| any of the others.
```

```
...

  |============                                                          |  18%
| Now sample 10 numbers between 1 and 20, WITHOUT replacement. To sample
| without replacement, simply leave off the 'replace' argument.

> salple(1:20)
Error in salple(1:20) : no se pudo encontrar la función "salple"
> sample(1:20)
 [1] 11 10  5  4 16 13 14  8 20 19 17  6 12  3  2  7  9 15 18  1

| Not exactly. Give it another go. Or, type info() for more options.

| Type sample(1:20, 10) to sample 10 numbers between 1 and 20, without
| replacement.

> sample(1:20, 10)
 [1]  6  4  3 17 12 20  9  1 11  8

| Keep working like that and you will get there!

  |==============                                                        |  21%
| Since the last command sampled without replacement, no number appears
| more than once in the output.

...

  |================                                                      |  24%
| LETTERS is a predefined variable in R containing a vector of all 26
| letters of the English alphabet. Take a look at it now.

> sample(a:z, 26)
Error in sample(a:z, 26) : objeto 'a' no encontrado
> sample('a':'z', 26)
Error in "a":"z" : Argumento NA/NaN
Además: Warning messages:
1: In sample("a":"z", 26) : NAs introducidos por coerción
2: In sample("a":"z", 26) : NAs introducidos por coerción
> sample(letters)
 [1] "c" "f" "p" "l" "k" "g" "w" "n" "h" "d" "r" "v" "m" "t" "u" "s" "x"
[18] "z" "i" "e" "b" "y" "a" "q" "j" "o"

| Not quite right, but keep trying. Or, type info() for more options.

| Type LETTERS to print its contents to the console.

> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
[18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

| Keep up the great work!

  |==================                                                    |  27%
```

```
| The sample() function can also be used to permute, or rearrange, the
| elements of a vector. For example, try sample(LETTERS) to permute all 26
| letters of the English alphabet.

> sample(LETTERS)
 [1] "H" "T" "M" "E" "J" "W" "D" "Y" "N" "S" "O" "K" "I" "L" "P" "Q" "G"
[18] "Z" "B" "F" "V" "A" "X" "R" "C" "U"

| You are the best!

  |====================                                                  |  30%
| This is identical to taking a sample of size 26 from LETTERS, without
| replacement. When the 'size' argument to sample() is not specified, R
| takes a sample equal in size to the vector from which you are sampling.

...

  |======================                                                |  33%
| Now, suppose we want to simulate 100 flips of an unfair two-sided coin.
| This particular coin has a 0.3 probability of landing 'tails' and a 0.7
| probability of landing 'heads'.

...

  |========================                                              |  36%
| Let the value 0 represent tails and the value 1 represent heads. Use
| sample() to draw a sample of size 100 from the vector c(0,1), with
| replacement. Since the coin is unfair, we must attach specific
| probabilities to the values 0 (tails) and 1 (heads) with a fourth
| argument, prob = c(0.3, 0.7). Assign the result to a new variable called
| flips.

>
> sample(c(0,1), 100, replace = TRUE, prob = c(0.3,0.7))
  [1] 1 1 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1
 [36] 1 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 1 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0
 [71] 1 0 1 0 0 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1

| Keep trying! Or, type info() for more options.

| The following command will produce 100 flips of an unfair coin and
| assign the result: flips <- sample(c(0,1), 100, replace = TRUE, prob =
| c(0.3, 0.7))

> flips <- sample(c(0,1), 100, replace = TRUE, prob = c(0.3,0.7))

| Perseverance, that is the answer.

  |==========================                                            |  39%
| View the contents of the flips variable.

> flips
  [1] 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1
 [36] 0 1 0 1 1 0 1 1 1 1 0 1 1 1 1 0 0 0 1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 0
```

```
 [71] 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 1

| You nailed it! Good job!

  |============================                                          |  42%
| Since we set the probability of landing heads on any given flip to be
| 0.7, we'd expect approximately 70 of our coin flips to have the value 1.
| Count the actual number of 1s contained in flips using the sum()
| function.

> sum(flips)
[1] 77

| You are doing so well!

  |==============================                                        |  45%
| A coin flip is a binary outcome (0 or 1) and we are performing 100
| independent trials (coin flips), so we can use rbinom() to simulate a
| binomial random variable. Pull up the documentation for rbinom() using
| ?rbinom.

> ?rbinom

| All that hard work is paying off!

  |================================                                      |  48%
| Each probability distribution in R has an r*** function (for "random"),
| a d*** function (for "density"), a p*** (for "probability"), and q***
| (for "quantile"). We are most interested in the r*** functions in this
| lesson, but I encourage you to explore the others on your own.

...

  |================================                                      |  52%
| A binomial random variable represents the number of 'successes' (heads)
| in a given number of independent 'trials' (coin flips). Therefore, we
| can generate a single random variable that represents the number of
| heads in 100 flips of our unfair coin using rbinom(1, size = 100, prob =
| 0.7). Note that you only specify the probability of 'success' (heads)
| and NOT the probability of 'failure' (tails). Try it now.

>
> rbinom(1, size = 100, prob = 0.7)
[1] 79

| You are quite good my friend!

  |==================================                                    |  55%
| Equivalently, if we want to see all of the 0s and 1s, we can request 100
| observations, each of size 1, with success probability of 0.7. Give it a
| try, assigning the result to a new variable called flips2.

> flips2 <- rbinom(1, size = 100, prob = 0.7)
```

```
| Not quite right, but keep trying. Or, type info() for more options.

| Call rbinom() with n = 100, size = 1, and prob = 0.7 and assign the
| result to flips2.

> flips2 <- rbinom(100, size = 1, prob = 0.7)

| Great job!

  |=====================================                                |  58%
| View the contents of flips2.

> flips2
  [1] 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 0
 [36] 1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1
 [71] 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1

| Nice work!

  |======================================                               |  61%
| Now use sum() to count the number of 1s (heads) in flips2. It should be
| close to 70!

> sum(flips)
[1] 77

| Not quite! Try again. Or, type info() for more options.

| Use sum(flips2) to count the number of 1s.

> sum(flips2)
[1] 76

| You are really on a roll!

  |=======================================                              |  64%
| Similar to rbinom(), we can use R to simulate random numbers from many
| other probability distributions. Pull up the documentation for rnorm()
| now.

> ?rnorm

| Perseverance, that is the answer.

  |========================================                             |  67%
| The standard normal distribution has mean 0 and standard deviation 1. As
| you can see under the 'Usage' section in the documentation, the default
| values for the 'mean' and 'sd' arguments to rnorm() are 0 and 1,
| respectively. Thus, rnorm(10) will generate 10 random numbers from a
| standard normal distribution. Give it a try.

> rnorm(10)
 [1] -0.02251028 -1.16492184 -0.68953187  0.04545480 -0.71316877
 [6] -0.80028453 -0.80750088 -1.26945338 -0.42451060  0.61641393
```

```
| You are the best!

  |=========================================                |  70%
| Now do the same, except with a mean of 100 and a standard deviation of
| 25.

> rnorm(10, mean = 100, sd = 25)
 [1]  87.14927 105.39969  90.81056 150.58953  98.29308 132.36737  78.78415
 [8]  90.85789  74.93612 111.46217

| All that practice is paying off!

  |=============================================            |  73%
| Finally, what if we want to simulate 100 *groups* of random numbers,
| each containing 5 values generated from a Poisson distribution with mean
| 10? Let is start with one group of 5 numbers, then I will show you how to
| repeat the operation 100 times in a convenient and compact way.

...

  |=================================================        |  76%
| Generate 5 random values from a Poisson distribution with mean 10. Check
| out the documentation for rpois() if you need help.

> rpois(5)
Error in rpois(5) :
  el argumento "lambda" está ausente, sin valor por omisión
> rpois(5, 10)
[1] 14 12  5  8 11

| Keep up the great work!

  |=================================================        |  79%
| Now use replicate(100, rpois(5, 10)) to perform this operation 100
| times. Store the result in a new variable called my_pois.

> my_pois <- rpois(5, rpois(5,10))

| One more time. You can do it! Or, type info() for more options.

| my_pois <- replicate(100, rpois(5, 10)) will repeat the operation 100
| times and store the result.

> my_pois <- replicate(5, rpois(5,10))

| You are close...I can feel it! Try it again. Or, type info() for more
| options.

| my_pois <- replicate(100, rpois(5, 10)) will repeat the operation 100
| times and store the result.

> my_pois <- replicate(100, rpois(5,10))
```

| You are amazing!

  |=== | 82%
| Take a look at the contents of my_pois.

```
> my_pois
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
[1,]    9    9    8   10    7    6   12    6   11    11     5     6    15
[2,]   12    3    8    6   12   11   10   14    7     9    14    13     8
[3,]   12    3   11   13   12    8   15   12   14     7    16    11     9
[4,]    9    6   11   10   10    3   12   13    9    13    14    13    10
[5,]   10   10    9    9    9    7   14   15   17     8     8    14    13
     [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
[1,]    13    12    10    16    12    10     5    11    14     8     9
[2,]    16     5     8     9    13     5    10    10     3    16    10
[3,]     5    12    11     6    12    15     8    17     9    11    21
[4,]    10    10     6    11     9    11    11     7    13    10     5
[5,]    16     6    10     7    18    11     9    11     9    10    14
     [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
[1,]    13     8    17    14     8    16     8     9    10    12    11
[2,]    10    14    10    12     8    11    14    12    17    11     7
[3,]     3     9    17    11     9     8     9    17     5    14     7
[4,]    10    10     8     3    10     7    15     7    12     5    18
[5,]     4     8     7    18     7     8     7     9     6     9     9
     [,36] [,37] [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46]
[1,]    12     7    13     9     9     7    10    11    12    11    12
[2,]     9    15    11     8    14    14    10    11     8     8     7
[3,]    12     6     7    13     7    12    11     9     9     7    11
[4,]     8    12     8    12    11    13     8     6    10     9    10
[5,]    12    11    12     9     8    11     9     7     7    16    14
     [,47] [,48] [,49] [,50] [,51] [,52] [,53] [,54] [,55] [,56] [,57]
[1,]     7    11    17     4    13    18     5     3    13    10    12
[2,]    13    14     8    15    11     9    12     9    10    10    10
[3,]     8    12    11     6     9    12    11     6     7    10    11
[4,]     8     8     6     8     9     9     5    20    14     4    11
[5,]    11    16    13    10     9    17     7    10     9     8     5
     [,58] [,59] [,60] [,61] [,62] [,63] [,64] [,65] [,66] [,67] [,68]
[1,]     8     9     8     4    13    11    10    15    11     8    13
[2,]     5     9    11    19     9     8    12    12     6     9     9
[3,]    14     9    14    15    13    12     5    11    12    15    13
[4,]    12     8    13    11    11    12     7     7    12     8     8
[5,]    11     7     7     8    12    15    11    12    13     9     9
     [,69] [,70] [,71] [,72] [,73] [,74] [,75] [,76] [,77] [,78] [,79]
[1,]    11     8    12    13    10    13     9    11     7    14    10
[2,]    11     6     8     9    10    10    10    12     8     4     8
[3,]    18    13     8     6    12    13     9    11    12    11     9
[4,]     7     6    13     5    11     9     6     7    14    13     9
[5,]    11     9    12    10     8    18    11    11     6    15    10
     [,80] [,81] [,82] [,83] [,84] [,85] [,86] [,87] [,88] [,89] [,90]
[1,]     7    12     9    10     8    12    11     7    19     9     7
[2,]    14     9     6    13    12    10     9     6    12    13     6
[3,]     7    14    13    15     9    10    10     6    13    14     9
[4,]     8    10    10     5    14    14     8    11     9     9     8
[5,]     7    16    10    10     7     9    12    12     5    12    12
```

```
     [,91] [,92] [,93] [,94] [,95] [,96] [,97] [,98] [,99] [,100]
[1,]     6    12     8     7     7    12     9     9    14     10
[2,]    13     7    15    14    10     8     5     8     5      8
[3,]    11    10     9    14    11     6     3     7    12     11
[4,]     9    16    12     4     9    11    10    10     7      7
[5,]     8    14     5     8    10    23    16    10     5      7

| That is the answer I was looking for.

  |=======================================================           |  85%
| replicate() created a matrix, each column of which contains 5 random
| numbers generated from a Poisson distribution with mean 10. Now we can
| find the mean of each column in my_pois using the colMeans() function.
| Store the result in a variable called cm.

> cm <- colMeans(my_pois)

| Great job!

  |=========================================================         |  88%
| And let is take a look at the distribution of our column means by
| plotting a histogram with hist(cm).

> hist(cm)

| You are really on a roll!

  |===========================================================       |  91%
| Looks like our column means are almost normally distributed, right?
| That is the Central Limit Theorem at work, but that is a lesson for
| another day!

...

  |=============================================================     |  94%
| All of the standard probability distributions are built into R,
| including exponential (rexp()), chi-squared (rchisq()), gamma
| (rgamma()), .... Well, you see the pattern.

...

  |===============================================================   |  97%
| Simulation is practically a field of its own and we've only skimmed the
| surface of what is possible. I encourage you to explore these and other
| functions further on your own.

...

  |=================================================================| 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: Yes
2: No
```

```
Selection: 1
```

f

Swirl 14: Fechas y tiempos en R

```
Selection: 14

  |                                                                      |   0%

| R has a special way of representing dates and times, which can be helpful
| if you are working with data that show how something changes over time (i.e.
| time-series data) or if your data contain some other temporal information,
| like dates of birth.

...

  |==                                                                    |   3%
| Dates are represented by the 'Date' class and times are represented by the
| 'POSIXct' and 'POSIXlt' classes. Internally, dates are stored as the number
| of days since 1970-01-01 and times are stored as either the number of
| seconds since 1970-01-01 (for 'POSIXct') or a list of seconds, minutes,
| hours, etc. (for 'POSIXlt').

...

  |====                                                                  |   6%
| Let's start by using d1 <- Sys.Date() to get the current date and store it
| in the variable d1. (That's the letter 'd' and the number 1.)

> d1 <- Sys.Date()

| Your dedication is inspiring!

  |======                                                                |   8%
| Use the clas() function to confirm d1 is a Date object.

> clas(d1)
[1] "Date"

| You are really on a roll!

  |========                                                              |  11%
| We can use the unclass() function to see what d1 looks like internally. Try
| it out.

> unclass()
Error in unclass() : 0 arguments passed to 'unclass' which requires 1
> unclass(d1)
```

```
[1] 18697

| You are the best!

  |=========                                                             |  14%
| That is the exact number of days since 1970-01-01!

...

  |===========                                                           |  17%
| However, if you print d1 to the console, you'll get today's date --
| YEAR-MONTH-DAY. Give it a try.

> d1
[1] "2021-03-11"

| You are really on a roll!

  |=============                                                         |  19%
| What if we need to reference a date prior to 1970-01-01? Create a variable
| d2 containing as.Date("1969-01-01").

> d2 <- as.Date("1969-01-01")

| Your dedication is inspiring!

  |===============                                                       |  22%
| Now use unclass() again to see what d2 looks like internally.

> unclass()
Error in unclass() : 0 arguments passed to 'unclass' which requires 1
> unclass(d2)
[1] -365

| Excellent job!

  |=================                                                     |  25%
| As you may have anticipated, you get a negative number. In this case, it is
| -365, since 1969-01-01 is exactly one calendar year (i.e. 365 days) BEFORE
| 1970-01-01.

...

  |===================                                                   |  28%
| Now, let is take a look at how R stores times. You can access the current
| date and time using the Sys.time() function with no arguments. Do this and
| store the result in a variable called t1.

> t1 <- Sys.time()

| Keep working like that and you will get there!

  |=====================                                                 |  31%
| View the contents of t1.
```

```
> t1
[1] "2021-03-11 21:36:37 -05"

| You are the best!

  |======================                                          |  33%
| And check the clas() of t1.

> clas(t1)
[1] "POSIXct" "POSIXt"

| You are amazing!

  |=========================                                       |  36%
| As mentioned earlier, POSIXct is just one of two ways that R represents
| time information. (You can ignore the second value above, POSIXt, which
| just functions as a common language between POSIXct and POSIXlt.) Use
| unclass() to see what t1 looks like internally -- the (large) number of
| seconds since the beginning of 1970.

> unclass(t1)
[1] 1615516597

| You are really on a roll!

  |============================                                    |  39%
//| By default, Sys.time() returns an object of class POSIXct, but we can
| coerce the result to POSIXlt with as.POSIXlt(Sys.time()). Give it a try and
| store the result in t2.

//> t2 <- as.POSIXlt(Sys.time())

| That is a job well done!

  |==============================                                  |  42%
| Check the clas of t2.

//> class(t2)
[1] "POSIXlt" "POSIXt"

| All that hard work is paying off!

  |================================                                |  44%
| Now view its //contents.

> t2
[1] "2021-03-11 21:37:43 -05"

| Keep working like that and you will get there!

  |==================================                              |  47%
| The printed format of t2 is identical to that of t1. Now unclass() t2 to
| see how it is different internally.
```

```
> unclass(t2)
$sec
[1] 43.61978

$min
[1] 37

$hour
[1] 21

$mday
[1] 11

$mon
[1] 2

$year
[1] 121

$wday
[1] 4

$yday
[1] 69

$isdst
[1] 0

$zone
[1] "-05"

$gmtoff
[1] -18000

attr(,"tzone")
[1] ""    "-05" "-04"

| You are doing so well!

  |=================================                                     |  50%
| t2, like all POSIXlt objects, is just a list of values that make up the
| date and time. Use str(unclass(t2)) to have a more compact view.

> str(unclass(t2))
List of 11
 $ sec   : num 43.6
 $ min   : int 37
 $ hour  : int 21
 $ mday  : int 11
 $ mon   : int 2
 $ year  : int 121
 $ wday  : int 4
 $ yday  : int 69
```

```
 $ isdst : int 0
 $ zone  : chr "-05"
 $ gmtoff: int -18000
 - attr(*, "tzone")= chr [1:3] "" "-05" "-04"

| That is correct!

  |====================================                                  |  53%
| If, for example, we want just the minutes from the time stored in t2, we
| can access them with t2$min. Give it a try.

> t2$min
[1] 37

| Perseverance, that is the answer.

  |=======================================                               |  56%
| Now that we have explored all three types of date and time objects, let is
| look at a few functions that extract useful information from any of these
| objects -- weekdays(), months(), and quarters().

...

  |=========================================                             |  58%
| The weekdays() function will return the day of week from any date or time
| object. Try it out on d1, which is the Date object that contains today is
| date.

> weekdays(d1)
[1] "jueves"

| Excellent work!

  |============================================                          |  61%
| The months() function also works on any date or time object. Try it on t1,
| which is the POSIXct object that contains the current time (well, it was
| the current time when you created it).

> months(t1)
[1] "Marzo"

| You got it right!

  |==============================================                        |  64%
| The quarters() function returns the quarter of the year (Q1-Q4) from any
| date or time object. Try it on t2, which is the POSIXlt object that
| contains the time at which you created it.

> months(t2)
[1] "Marzo"

| Give it another try. Or, type info() for more options.

| quarters(t2) will give you the current quarter.
```

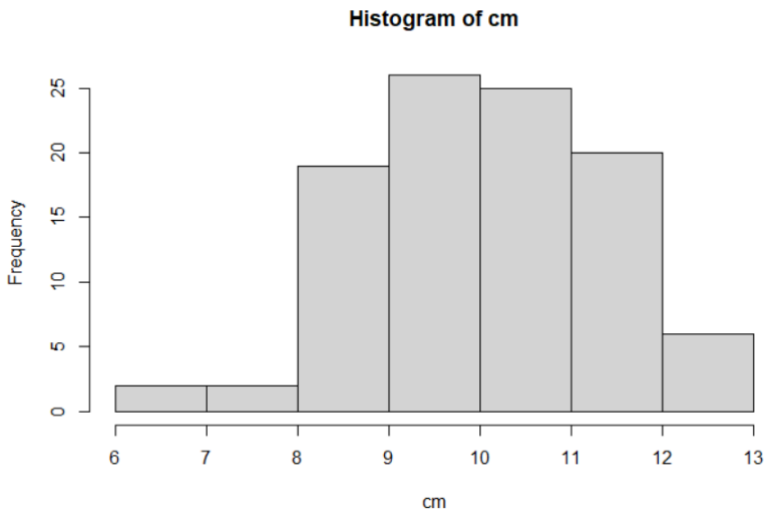
```
> quarters(t2)
[1] "Q1"

| That is correct!

  |=========================================                             |  67%
| Often, the dates and times in a dataset will be in a format that R does not
| recognize. The strptime() function can be helpful in this situation.

...

  |=============================================                         |  69%
| strptime() converts character vectors to POSIXlt. In that sense, it is
| similar to as.POSIXlt(), except that the input does not have to be in a
| particular format (YYYY-MM-DD).

...

  |=================================================                     |  72%
| To see how it works, store the following character string in a variable
| called t3: "October 17, 1986 08:24" (with the quotes).

> t3 <- "October 17, 1986 08:24"

| That is correct!

  |=====================================================                 |  75%
| Now, use strptime(t3, "%B %d, %Y %H:%M") to help R convert our date/time
| object to a format that it understands. Assign the result to a new variable
| called t4. (You should pull up the documentation for strptime() if you had
| like to know more about how it works.)

> strptime(t3, "%B %d, %Y %H:%M")
[1] NA

| Nice try, but that is not exactly what I was hoping for. Try again. Or, type
| info() for more options.

| t4 <- strptime(t3, "%B %d, %Y %H:%M") will convert our date/time object to
| a format that R understands.

> t4 <- strptime(t3, "%B %d, %Y %H:%M")

| That is the answer I was looking for.

  |=========================================================             |  78%
| Print the contents of t4.

> t4
[1] NA

| Great job!
```

```
  |=======================================================               |  81%
| That is the format we've come to expect. Now, let is check its clas().

//> clas(t4)
[1] "POSIXlt" "POSIXt"

| Excellent work!

  |=========================================================             |  83%
| Finally, there are a number of operations that you can perform on dates and
| times, including arithmetic operations (+ and -) and comparisons (<, ==,
//| etc.)

...

  |===========================================================           |  86%
| The variable t1 contains the time at which you created it (recall you used
| Sys.time()). Confirm that some time has passed since you created t1 by
| using the 'greater than' operator to compare it to the current time:
| Sys.time() > t1

//> Sys.time() > t1
[1] TRUE

| Your dedication is inspiring!

  |=============================================================         |  89%
| So we know that some time has passed, but how much? Try subtracting t1 from
| the current time using Sys.time() - t1. Do not forget the parentheses at the
| end of Sys.time(), since it is a function.

> Sys.time() - t1
Time difference of 8.429773 mins

| That is the answer I was looking for.

  |===============================================================       |  92%
| The same line of thinking applies to addition and the other comparison
| operators. If you want more control over the units when finding the above
| difference in times, you can use difftime(), which allows you to specify a
| 'units' parameter.

...

  |=================================================================     |  94%
| Use difftime(Sys.time(), t1, units = 'days') to find the amount of time in
| DAYS that has passed since you created t1.

> difftime(Sys.time(), t1, units= 'days')
Time difference of 0.006381585 days

| You nailed it! Good job!

  |===================================================================   |  97%
```

```
| In this lesson, you learned how to work with dates and times in R. While it
| is important to understand the basics, if you find yourself working with
| dates and times often, you may want to check out the lubridate package by
| Hadley Wickham.

...

  |======================================================================| 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: Yes
2: No
```

SWIRL 13:

Casi ultima figura:



Swirl 15: Base graphics

G

```
Selection: 15

  |                                                                      |   0%

| One of the greatest strengths of R, relative to other programming
| languages, is the ease with which we can create publication-quality
| graphics. In this lesson, you will learn about base graphics in R.

...
```

```
  |=                                                                |   2%
| We do not cover the more advanced portions of graphics in R in this
| lesson. These include lattice, ggplot2 and ggvis.

...

  |===                                                              |   4%
| There is a school of thought that this approach is backwards, that we
| should teach ggplot2 first. See
| http://varianceexplained.org/r/teach_ggplot2_to_beginners/ for an
| outline of this view.

...

  |====                                                             |   7%
| Load the included data frame cars with data(cars).

> data(cars)

| Yo are the best!

  |======                                                           |   9%
| To fix ideas, we will work with simple data frames. Our main goal is to
| introduce various plotting functions and their arguments. All the output
| would look more interesting with larger, more complex data sets.

...force(cars)

  |=======                                                          |  11%
| Pull up the help page for cars.

> force(cars)
   speed dist
1      4    2
2      4   10
3      7    4
4      7   22
5      8   16
6      9   10
7     10   18
8     10   26
9     10   34
10    11   17
11    11   28
12    12   14
13    12   20
14    12   24
15    12   28
16    13   26
17    13   34
18    13   34
19    13   46
20    14   26
21    14   36
```

```
22    14   60
23    14   80
24    15   20
25    15   26
26    15   54
27    16   32
28    16   40
29    17   32
30    17   40
31    17   50
32    18   42
33    18   56
34    18   76
35    18   84
36    19   36
37    19   46
38    19   68
39    20   32
40    20   48
41    20   52
42    20   56
43    20   64
44    22   66
45    23   54
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85

| You are close...I can feel it! Try it again. Or, type info() for more
| options.

| Type ?cars or help(cars) to view a help page with details on the car
| data frame.

> ?cars

| That is a job well done!

  |========                                                              |  13%
| As you can see in the help page, the cars data set has only two
| variables: speed and stopping distance. Note that the data is from the
| 1920s.

...

  |==========                                                            |  15%
| Run head() on the cars data.

> head(cars)
  speed dist
1     4    2
2     4   10
```

```
3     7    4
4     7   22
5     8   16
6     9   10

| Great job!

  |===========                                                     |  17%
| Before plotting, it is always a good idea to get a sense of the data.
| Key R commands for doing so include, dim(), names(), head(), tail() and
| summary().

...

  |=============                                                   |  20%
| Run the plot() command on the cars data frame.

> plot(cars)

| All that hard work is paying off!

  |==============                                                  |  22%
| As always, R tries very hard to give you something sensible given the
| information that you have provided to it. First, R notes that the data
| frame you have given it has just two columns, so it assumes that you
| want to plot one column versus the other.

...

  |================                                                |  24%
| Second, since we do not provide labels for either axis, R uses the names
| of the columns. Third, it creates axis tick marks at nice round numbers
| and labels them accordingly. Fourth, it uses the other defaults supplied
| in plot().

...

  |=================                                               |  26%
| We will now spend some time exploring plot, but many of the topics
| covered here will apply to most other R graphics functions. Note that
| 'plot' is short for scatterplot.

...

  |===================                                             |  28%
| Look up the help page for plot().

> _plot
Error: unexpected input in "_"
> ?plot

| You are amazing!

  |====================                                            |  30%
```

```
| The help page for plot() highlights the different arguments that the
| function can take. The two most important are x and y, the variables
| that will be plotted. For the next set of questions, include the
| argument names in your answers. That is, do not type plot(cars$speed,
| cars$dist), although that will work. Instead, use plot(x = cars$speed, y
| = cars$dist).

...

  |=====================                                                 |  33%
| Use plot() command to show speed on the x-axis and dist on the y-axis
| from the cars data frame. Use the form of the plot command in which
| vectors are explicitly passed in as arguments for x and y.

> plot(x,y)
Error in xy.coords(x, y, xlabel, ylabel, log) :
  'x' and 'y' lengths differ
> plot(cars)

| Not exactly. Give it another go. Or, type info() for more options.

| Type plot(x = cars$speed, y = cars$dist) to create the plot.

> plot(cars$speed, cars$dist)

| That is correct!

  |=======================                                               |  35%
| Note that this produces a slightly different answer than plot(cars). In
| this case, R is not sure what you want to use as the labels on the axes,
| so it just uses the arguments which you pass in, data frame name and
| dollar signs included.

...

  |========================                                              |  37%
| Note that there are other ways to call the plot command, i.e., using the
| "formula" interface. For example, we get a similar plot to the above
| with plot(dist ~ speed, cars). However, we will wait till later in the
| lesson before using the formula interface.

...

  |=========================                                             |  39%
| Use plot() command to show dist on the x-axis and speed on the y-axis
| from the cars data frame. This is the opposite of what we did above.

> plot(cars$dist, cars$speed)

| Keep up the great work!

  |==========================                                            |  41%
| It probably makes more sense for speed to go on the x-axis since
| stopping distance is a function of speed more than the other way around.
```

```
| So, for the rest of the questions in this portion of the lesson, always
| assign the arguments accordingly.

...

  |============================                                          |  43%
| In fact, you can assume that the answers to the next few questions are
| all of the form plot(x = cars$speed, y = cars$dist, ...) but with
| various arguments used in place of the ...

...

  |=============================                                         |  46%
| Recreate the plot with the label of the x-axis set to "Speed".

> plot(cars$speed, cars$dist, xlabel = "Speed")
Warning messages:
1: In plot.window(...) : "xlabel" is not a graphical parameter
2: In plot.xy(xy, type, ...) : "xlabel" is not a graphical parameter
3: In axis(side = side, at = at, labels = labels, ...) :
  "xlabel" is not a graphical parameter
4: In axis(side = side, at = at, labels = labels, ...) :
  "xlabel" is not a graphical parameter
5: In box(...) : "xlabel" is not a graphical parameter
6: In title(...) : "xlabel" is not a graphical parameter

| That's not exactly what I'm looking for. Try again. Or, type info() for
| more options.

| Type plot(x = cars$speed, y = cars$dist, xlab = "Speed") to create the
| plot.

> plot(cars$speed, cars$dist, xlab = "Speed")
There were 12 warnings (use warnings() to see them)

| You are quite good my friend!

  |=============================                                         |  48%
| Recreate the plot with the label of the y-axis set to "Stopping
| Distance".

> plot(cars$speed, cars$dist, ylab = "Stopping Distance")

| All that practice is paying off!

  |==============================                                        |  50%
| Recreate the plot with "Speed" and "Stopping Distance" as axis labels.

> plot(cars$speed, cars$dist, xlab = "Speed", ylab = "Stopping Distance")

| Excellent job!

  |==============================                                        |  52%
| The reason that plots(cars) worked at the beginning of the lesson was
```

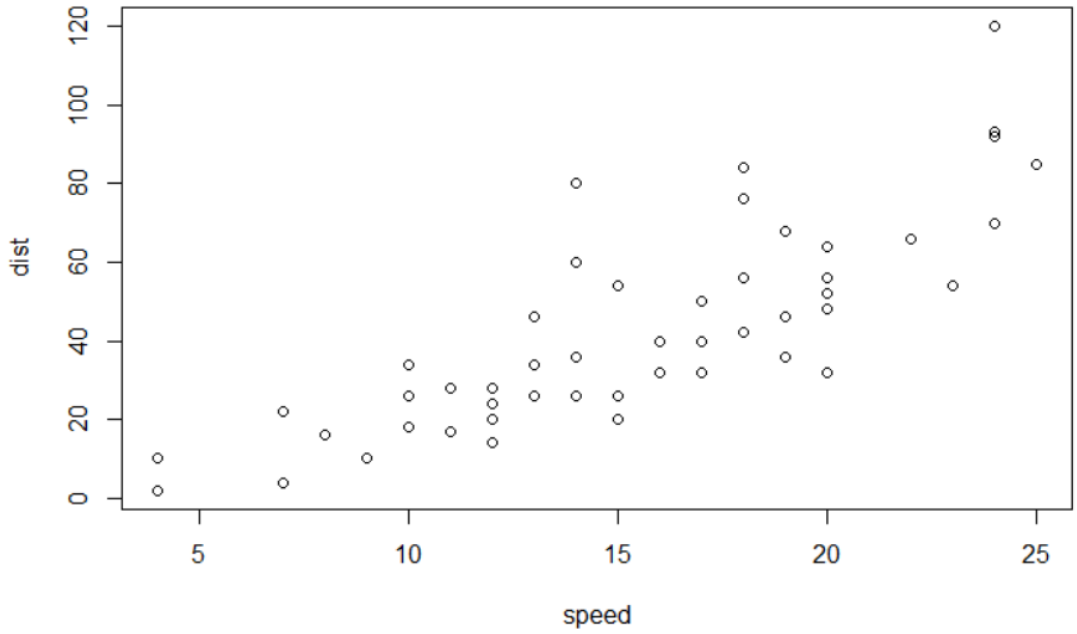
```
| that R was smart enough to know that the first element (i.e., the first
| column) in cars should be assigned to the x argument and the second
| element to the y argument. To save on typing, the next set of answers
| will all be of the form, plot(cars, ...) with various arguments added.

...

  |==================================                                    |  54%
| For each question, we will only want one additional argument at a time.
| Of course, you can pass in more than one argument when doing a real
| project.

...

  |====================================                                  |  57%
| Plot cars with a main title of "My Plot". Note that the argument for the
| main title is "main" not "title".

> plot(cars$speed, cars$dist, main = "My Plot")

| Keep trying! Or, type info() for more options.

| Type plot(cars, main = "My Plot") to create the plot.

> plot(cars, main = "My Plot")

| You nailed it! Good job!

  |======================================                                |  59%
| Plot cars with a sub title of "My Plot Subtitle".

> plot(cars, title = "My Plot Subtitle")
Warning messages:
1: In plot.window(...) : "title" is not a graphical parameter
2: In plot.xy(xy, type, ...) : "title" is not a graphical parameter
3: In axis(side = side, at = at, labels = labels, ...) :
  "title" is not a graphical parameter
4: In axis(side = side, at = at, labels = labels, ...) :
  "title" is not a graphical parameter
5: In box(...) : "title" is not a graphical parameter
6: In title(...) : "title" is not a graphical parameter

| Not quite! Try again. Or, type info() for more options.

| Type plot(cars, sub = "My Plot Subtitle") to create the plot.

> plot(cars, sub = "My Plot Subtitle")
There were 12 warnings (use warnings() to see them)

| You are the best!

  |======================================                                |  61%
| The plot help page (?plot) only covers a small number of the many
| arguments that can be passed in to plot() and to other graphical
```

```
| functions. To begin to explore the many other options, look at ?par.
| Let is look at some of the more commonly used ones. Continue using
| plot(cars, ...) as the base answer to these questions.

...

  |=======================================                          |  63%
| Plot cars so that the plotted points are colored red. (Use col = 2 to
| achieve this effect.)

> plot(cars, col = 2)

| Perseverance, that is the answer.

  |========================================                         |  65%
| Plot cars while limiting the x-axis to 10 through 15.  (Use xlim = c(10,
| 15) to achieve this effect.)

> plot(cars, xlim = c(10,15))

| Your dedication is inspiring!

  |=========================================                        |  67%
| You can also change the shape of the symbols in the plot. The help page
| for points (?points) provides the details.

...

  |==========================================                       |  70%
| Plot cars using triangles.  (Use pch = 2 to achieve this effect.)
> plot(cars, pch = 2)

| You are the best!

  |===========================================                      |  72%
| Arguments like "col" and "pch" may not seem very intuitive. And that is
| because they are not! So, many/most people use more modern packages, like
| ggplot2, for creating their graphics in R.

...

  |============================================                     |  74%
| It is, however, useful to have an introduction to base graphics because
| many of the idioms in lattice and ggplot2 are modeled on them.

...

  |=============================================                    |  76%
| Let is now look at some other functions in base graphics that may be
| useful, starting with boxplots.

...

  |==============================================                   |  78%
```

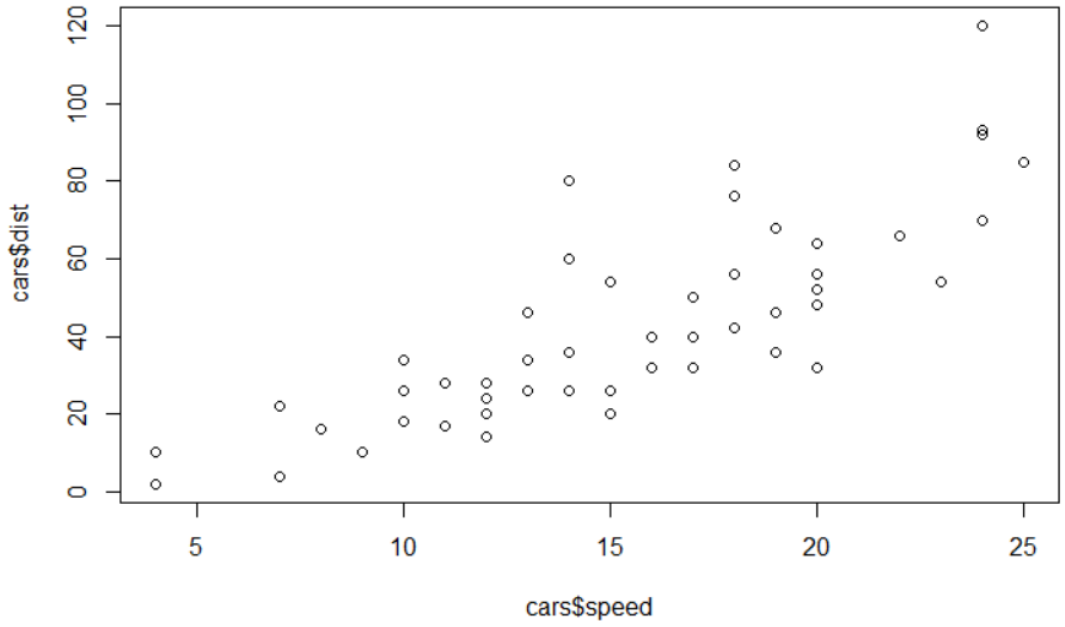
```
| Load the mtcars data frame.

> data(mtcars)

| You are quite good my friend!

  |=================================================        |  80%
| Anytime that you load up a new data frame, you should explore it before
| using it. In the middle of a swirl lesson, just type play(). This
| temporarily suspends the lesson (without losing the work you have
| already done) and allows you to issue commands like dim(mtcars) and
| head(mtcars). Once you are done examining the data, just type nxt() and
| the lesson will pick up where it left off.

...

| You are quite good my friend!

  |=================================================        |  80%
| Anytime that you load up a new data frame, you should explore it before
| using it. In the middle of a swirl lesson, just type play(). This
| temporarily suspends the lesson (without losing the work you have
| already done) and allows you to issue commands like dim(mtcars) and
| head(mtcars). Once you are done examining the data, just type nxt() and
| the lesson will pick up where it left off.

...

  |==================================================       |  83%
| Look up the help page for boxplot().

> ?boxplot

| You got it!

  |==================================================       |  85%
| Instead of adding data columns directly as input arguments, as we did
| with plot(), it is often handy to pass in the entire data frame. This is
| what the "data" argument in boxplot() allows.

...

  |==================================================       |  87%
| boxplot(), like many R functions, also takes a "formula" argument,
| generally an expression with a tilde ("~") which indicates the
| relationship between the input variables. This allows you to enter
| something like mpg ~ cyl to plot the relationship between cyl (number of
| cylinders) on the x-axis and mpg (miles per gallon) on the y-axis.

...

  |==================================================       |  89%
| Use boxplot() with formula = mpg ~ cyl and data = mtcars to create a box
| plot.
```

```
> boxplot(formula = mpg ~ cyl, data = mtcars)

| Great job!

  |=============================================================       | 91%
| The plot shows that mpg is much lower for cars with more cylinders. Note
| that we can use the same set of arguments that we explored with plot()
| above to add axis labels, titles and so on.

...

  |=================================================================     | 93%
| When looking at a single variable, histograms are a useful tool. hist()
| is the associated R function. Like plot(), hist() is best used by just
| passing in a single vector.

...

  |===================================================================   | 96%
| Use hist() with the vector mtcars$mpg to create a histogram.

> hist(mtcars$mpg)

| You are doing so well!

  |===================================================================== | 98%
| In this lesson, you learned how to work with base graphics in R. The
| best place to go from here is to study the ggplot2 package. If you want
| to explore other elements of base graphics, then this web page
| (http://www.ling.upenn.edu/~joseff/rstudy/week4.html) provides a useful
| overview.

...

  |======================================================================| 100%
| Would you like to receive credit for completing this course on
| Coursera.org?

1: No
2: Yes

Selection: 2
```

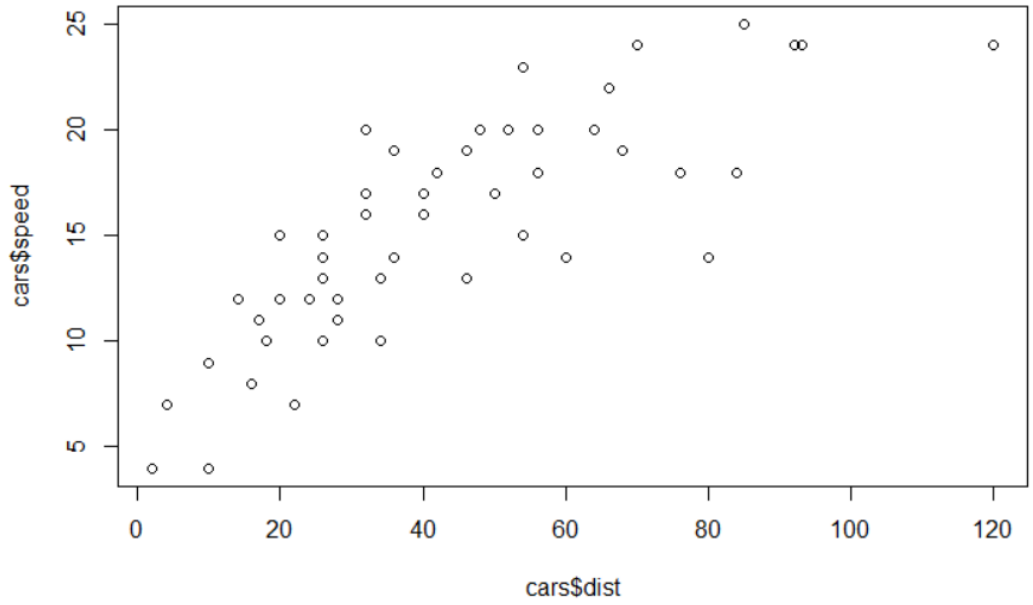
Figura 20%


dist
0
20
40
60
80
100
120
5
10
15
20
25
speed

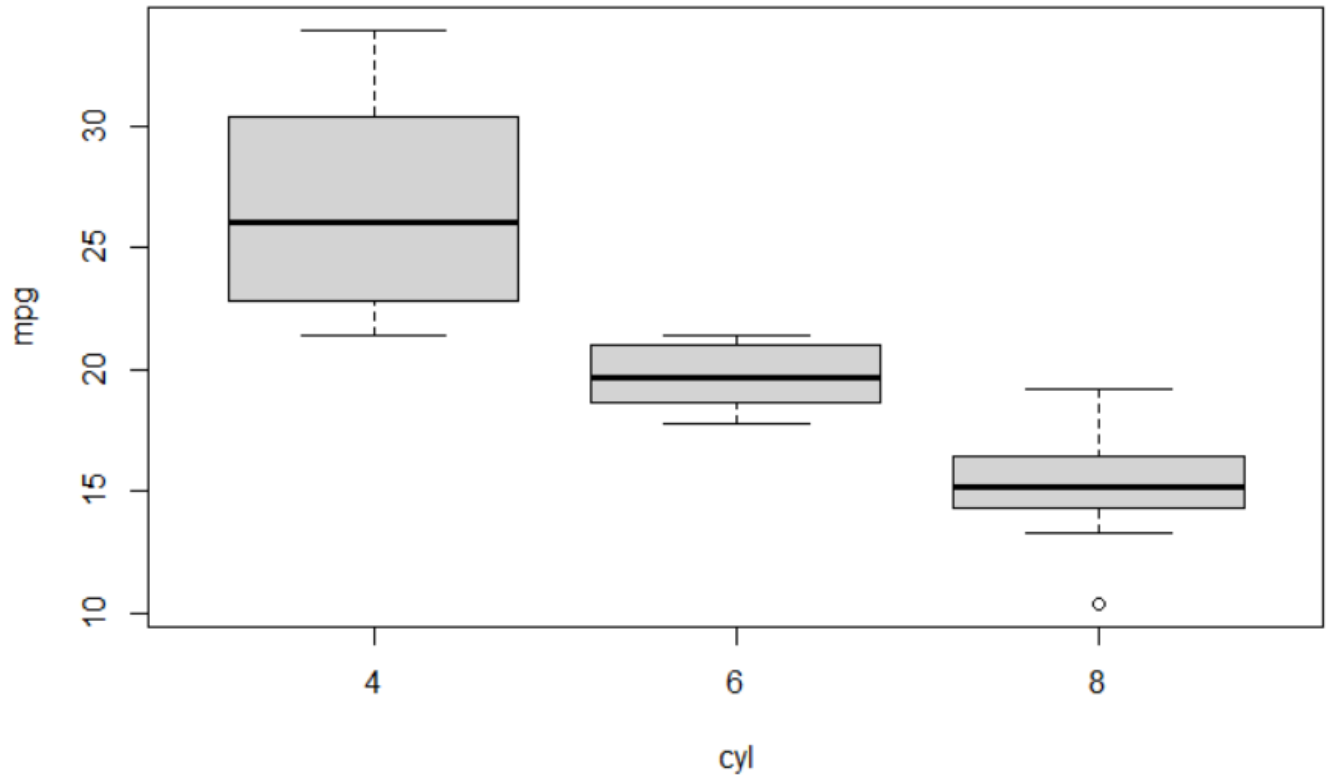
35%


cars$dist
0
20
40
60
80
100
120
5
10
15
20
25
cars$speed

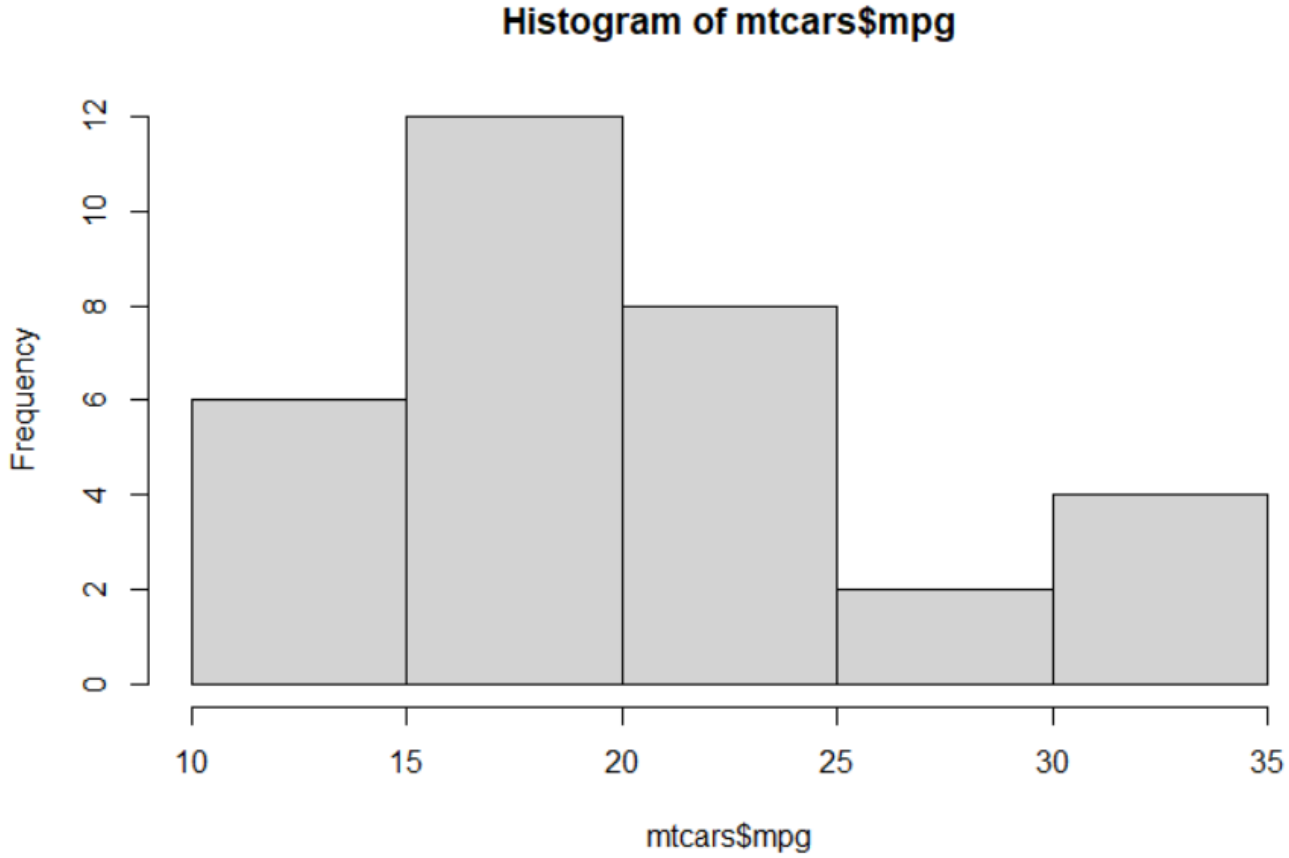
41%



Swirl 15 85%



95% por ahí


Histogram of mtcars$mpg
Frequency
0
2
4
6
8
10
12
10
15
20
25
30
35
mtcars$mpg