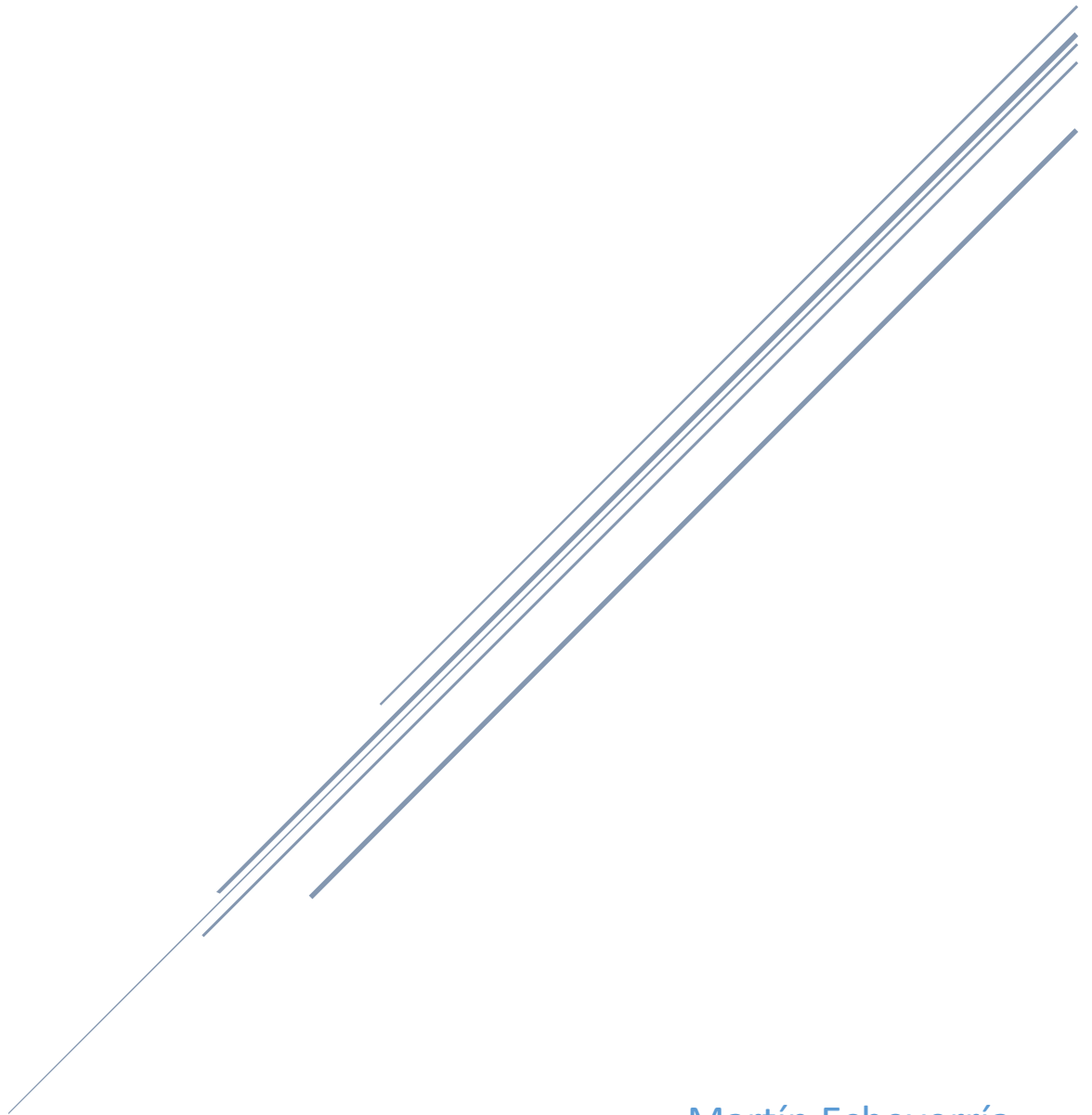


# INTRODUCCION A DATA SCIENCE. PROGRAMACION ESTADÍSTICA CON R

COURSERA





Martín Echeverría  
Inferencia Estadística

## Tabla de contenido

1.- Obtener ayuda. ....	2
2.- Objetos Tipos de Datos y Operaciones Básicas. ....	8
3.- Subconjunto de Datos. ....	24
4.- Leer y Escribir Datos. ....	41
5.- Funciones. ....	52

← → ↺ 🏠 <https://www.coursera.org/learn/intro-data-science-programacion-estadistica-r/home/welcome> ... 📖 ⭐

**coursera** Explorar ▾ 🔍 What do you want to learn?  Martín ▾




**Página de inicio del curso**

- Semana 1
- Semana 2
- Semana 3 📅
- Semana 4
- Calificaciones
- Foros de debate
- Mensajes
- Información del Curso

### Introducción a Data Science: Programación Estadística con R

por Universidad Nacional Autónoma de México



**SEMANA 3**

**Video: Funciones \*apply: apply**  
Tomará alrededor de 7 min. Después de que finalices, continúa e intenta terminar la semana antes de lo previsto. 7 min [Comenzar](#)

- ✓ SEMANA 1
- ✓ SEMANA 2

## 1.- Obtener ayuda.

Selection: 1

| | 0%

| En esta lección conocerás las principales herramientas que R tiene para obtener ayuda.

...

|=====| 6%

| La primera herramienta que puedes usar para obtener ayuda es `help.start()`. En ella encontrarás un

| menú de recursos, entre los cuales se encuentran manuales, referencias y demás material para  
| comenzar a aprender R.

...

|=====| 12%

| Para usar `help.start()` escribe en la línea de comandos `help.start()`. Pruébalo ahora:

> `help.start()`

If nothing happens, you should open

'<http://127.0.0.1:29722/doc/html/index.html>' yourself

| ¡Bien hecho!

|=====| 18%

| R incluye un sistema de ayuda que te facilita obtener información acerca de las funciones de los  
| paquetes instalados. Para obtener información acerca de una función, por ejemplo de la función  
| `print()`, debes escribir `?print` en la línea de comandos.

...

|=====| 24%

| Ahora es tu turno, introduce `?print` en la línea de comandos.

> `?print`

| Perseverancia es la respuesta.

|=====| 29%

| Como puedes observar `?print` te muestra en la ventana Help una breve descripción de la función, de  
cómo

| usarla, así como sus argumentos, etcétera.

...

|=====| 35%

| Asimismo, puedes usar la función help(), la cual es un equivalente de ?. Al utilizar help(), usarás  
| como argumento el nombre de la función entre comillas, por ejemplo, help("print").

...

|=====| 41%

| Para buscar ayuda sobre un operador, éste tiene que encontrarse entre comillas inversas. Por  
| ejemplo, si buscas información del operador +, deberás escribir help(`+`) o ?`+` en la línea de  
| comandos.

...

|=====| 47%

| Otra herramienta disponible es la función apropos(), la cual recibe una cadena entre comillas  
como

| argumento y te muestra una lista de todas las funciones que contengan esa cadena. Inténtalo:  
escribe

| apropos("class") en la línea de comandos.

>

> apropos("class")

```
[1] ".checkMFClasses"      ".classEnv"            ".MFclass"
[4] ".OldClassesList"      ".rs.getR6ClassGeneratorMethod" ".rs.getR6ClassSymbols"
[7] ".rs.getSetRefClassSymbols" ".rs.getSingleClass"      ".rs.objectClass"
[10] ".rs.rnb.engineToCodeClass" ".rs.rpc.get_set_class_slots" ".rs.rpc.get_set_ref_class_call"
[13] ".selectSuperClasses"   ".valueClassTest"        "all.equal.envRefClass"
[16] "assignClassDef"        "class"                   "class<-"
[19] "classesToAM"          "classLabel"              "classMetaName"
[22] "className"            "completeClassDefinition" "completeSubclasses"
[25] "data.class"           "findClass"               "getAllSuperClasses"
[28] "getClass"             "getClassDef"             "getClasses"
[31] "getClassName"         "getClassPackage"         "getRefClass"
```

[34] "getSubclasses"	"insertClassMethods"	"isClass"
[37] "isClassDef"	"isClassUnion"	"isSealedClass"
[40] "isVirtualClass"	"isXS3Class"	"makeClassRepresentation"
[43] "makePrototypeFromClassDef"	"multipleClasses"	"namespaceImportClasses"
[46] "nclass.FD"	"nclass.scott"	"nclass.Sturges"
[49] "newClassRepresentation"	"oldClass"	"oldClass<-"
[52] "promptClass"	"removeClass"	"resetClass"
[55] "S3Class"	"S3Class<-"	"sealClass"
[58] "selectSuperClasses"	"setClass"	"setClassUnion"
[61] "setOldClass"	"setRefClass"	"showClass"
[64] "superClassDepth"	"unclass"	

| ¡Es asombroso!

| ===== | 53%

| También puedes obtener ejemplos del uso de funciones con la función `example()`. Por ejemplo, escribe

| `example("read.table")`.

> `example("read.table")`

rd.tbl> ## using count.fields to handle unknown maximum number of fields

rd.tbl> ## when fill = TRUE

rd.tbl> test1 <- c(1:5, "6,7", "8,9,10")

rd.tbl> tf <- tempfile()

rd.tbl> writeLines(test1, tf)

rd.tbl> read.csv(tf, fill = TRUE) # 1 column

X1

1 2

2 3

3 4

4 5

5 6

6 7

7 8

8 9

9 10

```
rd.tbl> ncol <- max(count.fields(tf, sep = ","))
```

```
rd.tbl> read.csv(tf, fill = TRUE, header = FALSE,
```

```
rd.tbl+   col.names = paste0("V", seq_len(ncol)))
```

```
  V1 V2 V3
```

```
1 1 NA NA
```

```
2 2 NA NA
```

```
3 3 NA NA
```

```
4 4 NA NA
```

```
5 5 NA NA
```

```
6 6 7 NA
```

```
7 8 9 10
```

```
rd.tbl> unlink(tf)
```

```
rd.tbl> ## "Inline" data set, using text=
```

```
rd.tbl> ## Notice that leading and trailing empty lines are auto-trimmed
```

```
rd.tbl>
```

```
rd.tbl> read.table(header = TRUE, text = "
```

```
rd.tbl+ a b
```

```
rd.tbl+ 1 2
```

```
rd.tbl+ 3 4
```

```
rd.tbl+ ")
```

```
  a b
```

```
1 1 2
```

```
2 3 4
```

```
| ¡Es asombroso!
```

|=====| 59%

| Con eso tendrás una idea de lo que puedes hacer con esta función.

...

|=====|  
65%

| R te permite buscar información sobre un tema usando `??`. Por ejemplo, escribe `??regression` en la

| línea de comandos.

> `??regression`

| ¡Buen trabajo!

|=====|  
71%

| Esta herramienta es muy útil si no recuerdas el nombre de una función, ya que R te mostrará una

| lista de temas relevantes en la ventana Help. Análogamente, puedes usar la función

| `help.search("regression")`.

...

|=====|  
76%

| Otra manera de obtener información de ayuda sobre un paquete es usar la opción `help` para el comando

| `library`, con lo cual tendrás información más completa. Un ejemplo es `library(help="stats")`.

...

|=====|  
82%

| Algunos paquetes incluyen viñetas. Una viñeta es un documento corto que describe cómo se usa un

| paquete. Puedes ver una viñetas usando la función `vignette()`. Pruébalo: escribe `vignette("tests")` en

| la línea de comandos.

>

> vignette("tests")

| ¡Excelente!

|=====

=== | 88%

| Por último si deseas ver la lista de viñetas disponibles puedes hacerlo usando el comando  
vignette()

| con los paréntesis vacíos.

...

|=====

===== | 94%

| Es MUY IMPORTANTE que sepas que durante todo el curso en swirl, puedes hacer uso de las  
funciones

| help() o ? cuando lo desees, incluso si estas en medio de una lección.

...

|=====

===== | 100%

| Has concluido la lección. ¿Te gustaría que se le notificará a Coursera que has completado esta  
| lección?

1: Si

2: No

Selection: 1

¿Cuál es tu nombre de usuario registrado en Coursera (email)? migevi97@gmail.com

¿Cuál es tu token de la tarea? f95BfBTnQi7Cqanx

¿El envío de la calificación fue satisfactorio!



## 2.- Objetos Tipos de Datos y Operaciones Básicas.

Selection: 2 | 0%

| En esta lección conocerás los tipos de datos que existen en el lenguaje R, además de las operaciones básicas que puedes hacer con ellos.

...

| = | 1% | Cuando introduces una expresión en la línea de comandos y das ENTER, R evalúa la expresión y muestra el resultado (si es que existe uno). R puede ser usado como una calculadora, ya que realiza operaciones aritméticas, además de operaciones lógicas.

...

| == | 2% | Pruébalo: ingresa 3 + 7 en la línea de comandos.

> 3+7 [1] 10

| ¡Buen trabajo!

| === | 3% | R simplemente imprime el resultado 10 por defecto. Sin embargo, R es un lenguaje de programación y normalmente la razón por la que usas éstos es para automatizar algún proceso y evitar la repetición innecesaria.

...

| ==== | 4% | En ese caso, tal vez quieras usar el resultado anterior en algún otro cálculo. Así que en lugar de volver a teclear la expresión cada vez que la necesites, puedes crear una variable que guarde el resultado de ésta.

...

| ===== | 6% | La manera de asignar un valor a una variable en R es usar el operador de asignación, el cual es sólo un símbolo de menor que seguido de un signo de menos, mejor conocido como guion alto. El operador se ve así:

| <-

...

| ===== | 7% | Por ejemplo, ahora ingresa en la línea de comandos: mi\_variable <- (180 / 6) - 15

> mi\_variable <- (180 / 6) - 15

| Esa es la respuesta que estaba buscando.

|===== | 8% | Lo que estás haciendo en este caso es asignarle a la variable `mi_variable` el valor de todo lo que se encuentra del lado derecho del operador de asignación, en este caso  $(180 / 6) - 15$ .

... $(180 / 6) - 15$

|===== | 9% | En R también puedes asignar del lado izquierdo:  $(180 / 6) - 15 \rightarrow \text{mi\_variable}$

... $(180 / 6) - 15 \rightarrow \text{mi\_variable}$

|===== | 10% | Como ya te habrás dado cuenta, la asignación ' $<-$ ' no muestra ningún resultado. Antes de ver el contenido de la variable '`mi_variable`', ¿qué crees que contenga la variable '`mi_variable`'?

1: la expresión  $(180 / 6) - 15$  2: la dirección de memoria de la variable '`mi_variable`' 3: la expresión evaluada, es decir un 15

Selection: 3

| ¡Eres bastante bueno!

|===== | 11% | La variable '`mi_variable`' deberá contener el número 15, debido a que  $(180 / 6) - 15 = 15$ . Para revisar el contenido de una variable, basta con escribir el nombre de ésta en la línea de comandos y presionar ENTER. Intentalo: muestra el contenido de la variable '`mi_variable`':

```
> (180 / 6) - 15 = 15 Error in (180/6) - 15 = 15 : target of assignment expands to non-language object
> mi_variable
```

```
[1] 15
```

| ¡Tu dedicación es inspiradora!

|===== | 12% | Nota que el '[1]' acompaña a los valores mostrados al evaluar las expresiones anteriores. Esto se debe a que en R todo número que introduces en la consola es interpretado como un vector.

...

|===== | 13% | Un vector es una colección ordenada de números, por lo cual el '[1]' denota la posición del primer elemento mostrado en el renglón 1. En los casos anteriores sólo existe un único elemento en el vector.

...

|===== | 15% | En R puedes construir vectores más largos usando la función `c()` (combine). Por ejemplo, introduce: `y <- c(561, 1105, 1729, 2465, 2821)`

```
> y <- + | c(561, 1105, 1729, 2465, 2821) Error: unexpected '|' in: "y <- |"
> y <- c(561, 1105, 1729, 2465, 2821)
```

| ¡Eso es trabajo bien hecho!

| ===== | 16% | Ahora observa el contenido de la variable 'y'. Otra manera de ver el contenido de una variable es | imprimirlo con la función print(). Introduce print(y) en la l línea de comandos:

```
> y [1] 561 1105 1729 2465 2821
```

| Estás muy cerca... ¡Puedo sentirlo! Inténtalo de nuevo. O esc ribe info() para más opciones.

```
> > print(y) [1] 561 1105 1729 2465 2821
```

| ¡Es asombroso!

| ===== | 17% | Como puedes notar, la expresión anterior resulta ser un vecto r que contiene los primeros cinco números de | Carmichael. Como ejemplo de un vector que abarque más de una línea, usa el operador de secuencia para | producir un vector con cada uno de los enteros del 1 al 100. Introduce 1:100 en la línea de comandos.

```
> 1:100 [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 [27] 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 [53] 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 [79] 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

| ¡Muy bien!

| ===== | 18% | El vector es el objeto más simple en R. La mayoría de las ope raciones están basadas en vectores.

...

| ===== | 19% | Por ejemplo, puedes realizar operaciones sobre vectores y R a utomáticamente empareja los elementos de los | dos vectores. Introduce c(1.1, 2.2, 3.3, 4.4) - c(1, 1, 1, 1) en la línea de comandos.

```
> c(1.1, 2.2, 3.3, 4.4) - c(1, 1, 1, 1) [1] 0.1 1.2 2.3 3.4
```

| ¡Sigue trabajando de esa manera y llegarás lejos!

| ===== | 20% | Nota: Si los dos vectores son de diferente tamaño, R repetirá la secuencia más pequeña múltiples veces. Por | ejemplo, introduce c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) - c(1, 2) en la línea de comandos.

```
> c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) - c(1, 2) [1] 0 0 2 2 4 4 6 6 8 8
```

| ¡Excelente!

| ===== | 21% | En R casi todo es un objeto. Para ver qué objetos tienes en u n momento determinado, puedes usar la función | ls(). Inténtalo ahora.

```
> ls() [1] "mi_variable" "ncol" "test1" "tf" "y"
```

| ¡Lo estás haciendo muy bien!

|=====| 22% | Como sabes,  
existen otros tipos de objetos, como los caracteres (character).

...(character)

|=====| 24% | Las expresiones  
con caracteres se denotan entre comillas. Por ejemplo, introduce "¡Hola Mundo!" en la línea | de  
comandos.

> "¡Hola Mundo!" [1] "¡Hola Mundo!"

| ¡Muy bien!

|=====| 25% | Esto es mejor  
conocido en R como un vector de caracteres. De hecho, este ejemplo es un vector de longitud |  
uno.

...

|=====| 26% | Ahora crea  
una variable llamada 'colores' que contenga un vector con las cadenas "rojo", "azul", "verde", |  
"azul", "rojo", en ese orden.

> colores <- c("rojo", "azul", "verde", "azul", "rojo")

| ¡Mantén este buen nivel!

|=====| 27% | Ahora  
imprime el vector 'colores' .

> colores [1] "rojo" "azul" "verde" "azul" "rojo"

| ¡Acertaste!

|=====| 28% | En otros  
lenguajes como C, carácter (character) hace referencia a un simple carácter, y cadena (string) se |  
entiende como un conjunto de caracteres ordenados. Una cadena de caracteres es equivalente al  
valor de

| carácter en R.

...

|=====| 29% | Además, hay  
objetos de tipo numérico (numeric) que se dividen en complejos (complex) y enteros (integer). |  
Los últimos ya los conoces, pues has estado trabajando con ellos, además de los vectores y los  
caracteres.

...

===== | 30% | Los  
complejos en R se representan de la siguiente manera:  $a+bi$ , donde 'a' es la parte real y 'b' la parte  
| imaginaria. Pruébalo: guarda el valor de  $2+1i$  en la variable 'complejo'.

```
> complejo <- 2+1i
```

| ¡Eres bastante bueno!

===== | 31% | Al igual  
que los demás objetos de tipo numérico, los complejos pueden hacer uso de los operadores |  
aritméticos más comunes, como '+' (suma), '-' (resta, o negación en el caso unario), '/' (división),  
'\*' | (multiplicación) '^' (donde  $x^2$  significa 'x elevada a la potencia 2'). Para obtener la raíz  
cuadrada, usa | la función `sqrt()`, y para obtener el valor absoluto, la función `abs()`.

...

===== | 33% | También  
hay objetos lógicos (logic) que representan los valores lógicos falso y verdadero.

...

===== | 34% | El valor  
lógico falso puede ser representado por la instrucción FALSE o únicamente por la letra F |  
mayúscula; de la misma manera, el valor lógico verdadero es representado por la instrucción  
TRUE o por la | letra T.

...

===== | 35%  
| Como operadores lógicos están el AND lógico: '&' y '&&' y el OR lógico: '|' y '||'.

...

===== | 36% |  
También existen operadores que devuelven valores lógicos, éstos pueden ser de orden, como: '>'  
(mayor que), '<' (menor que), '>=' (mayor igual) y '<=' (menor igual), o de comparación, como:  
'==' (igualdad) y '!=' | (diferencia). Por ejemplo, introduce en la línea de comandos `mi_variable ==`  
15.

```
> mi_variable == 15 [1] TRUE
```

| ¡Eso es trabajo bien hecho!

===== | 37% | Como  
puedes ver, R te devuelve el valor TRUE, pues si recuerdas, en la variable 'mi\_variable' asignaste el  
| valor de la expresión  $(180 / 6) - 15$ , la cual resultaba en el valor 15. Por lo cual, cuando le  
preguntas a | R si 'mi\_variable' es igual a 15, te devuelve el valor TRUE.

...

|=====| 38% | En R  
existen algunos valores especiales.

...

|=====| 39% | Por  
ejemplo, los valores NA son usados para representar valores faltantes. Supón que cambias el  
tamaño de | un vector a un valor más grande del previamente definido. Recuerda el vector  
'complejo', el cual contenía | el número complejo 2+1i; cambia la longitud de 'complejo'. Ingresa  
length(complejo) <- 3 en la línea de | comandos.

> length(complejo) <- 3

| ¡Acertaste!

|=====| 40% |  
Ahora ve el contenido de 'complejo'.

> complejo [1] 2+1i NA NA

| ¡Es asombroso!

|=====| 42% | Los  
nuevos espacios tendrán el valor NA, el cual quiere decir not available (no disponible).

...

|=====| 43% | Si  
un resultado de la evaluación de alguna expresión aritmética es muy grande, R regresa el valor  
'Inf' | para un valor positivo y '-Inf' para un valor negativo (infinitos positivo y negativo,  
respectivamente). | Por ejemplo, introduce 2^1024 en la línea de comandos.

> 2^1024 [1] Inf

| ¡Excelente trabajo!

|=====| 44%

| Algunas veces la evaluación de alguna expresión no tendrá sentido. En estos casos, R regresará el  
valor Nan (not a number). Por ejemplo,

| divide 0 entre 0.

> 0/0

[1] NaN

| ¡Excelente!

|=====| 45%

| Adicionalmente, en R existe el objeto null y es representado por el símbolo NULL.

...

|=====| 46%

| Nota que NULL no es lo mismo que NA, Inf, -Inf o Nan.

...

|=====| 47%

| Recuerda que R incluye un conjunto de clases para representar fechas y horas. Algunas de ellas son: Date, POSIXct y POSIXlt.

...

|=====| 48%

| Por ejemplo, introduce fecha\_primer\_curso\_R <- Sys.Date() en la línea de comandos.

```
> fecha_primer_curso_R <- Sys.Date()
```

| ¡Bien hecho!

|=====| 49%

| Ahora imprime el contenido de fecha\_primer\_curso\_R.

```
> fecha_primer_curso_R
```

```
[1] "2018-06-25"
```

| ¡Excelente trabajo!

|=====| 51%

| Recuerda que R te permite llevar a cabo operaciones numéricas y estadísticas con las fechas y horas. Además, R incluye funciones para

| manipularlas. Muchas funciones de graficación requieren fechas y horas.

...

|=====| 52%

| Ahora que conoces los objetos más usados en R, debes saber que además de la función c(), en R existe la función vector() para crear

| vectores. Al usar la función vector(), debes especificar el tipo de dato que almacenará el vector y el tamaño.

...

|=====| 53%

| La función vector() crea un vector con los valores por defecto del tipo especificado.

...

|=====| 54%

| Por ejemplo, escribe `vector("numeric", length = 10)` en la línea de comandos.

> `vector("numeric", length = 10)`

[1] 0 0 0 0 0 0 0 0 0 0

| ¡Eso es trabajo bien hecho!

|=====| 55%

| Como puedes observar, el vector fue llenado con ceros.

...

|=====| 56%

| Ahora introduce `vector("character", length = 10)` en la línea de comandos.

> `vector("character", length = 10)`

[1] "" "" "" "" "" "" "" "" "" ""

| ¡Eres el mejor!

|=====| 57%

| Esta vez el vector fue llenado con 10 cadenas vacías.

...

|=====| 58%

| Si le indicaras al vector que el tipo fuera "logical", ¿qué crees que contendría?

1: sólo valores FALSE

2: sólo valores TRUE

3: ninguna de las anteriores

Selection: 3

| No, ¡casi!, pero no. Intenta de nuevo.

1: sólo valores FALSE

2: sólo valores TRUE

3: ninguna de las anteriores

Selection: 1

| ¡Eres bastante bueno!

|=====| 60%



| Crea un vector de tipo "logical" de tamaño 10 usando la función vector().

```
> vector("logical",length = 10)
```

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

| Esa es la respuesta que estaba buscando.

| ===== | 61%

| Nota: Ahora que conoces los objetos más usados en R, es importante que sepas que los vectores sólo pueden contener objetos de la misma

| clase. Para guardar objetos de diferentes clases, puedes usar listas.

...

| ===== | 62%

| Las listas son una colección ordenada de objetos. Para crear una lista, usa la función list() y especifica el contenido de la lista

| separado por comas dentro de los paréntesis. Inténtalo: crea una lista que contenga un 0, la cadena "Hola" y el valor TRUE, en ese

| orden.

```
> list(0,"hola",TRUE)
```

```
[[1]]
```

```
[1] 0
```

```
[[2]]
```

```
[1] "hola"
```

```
[[3]]
```

```
[1] TRUE
```

| Por poco era correcto, sigue intentándolo. O escribe info() para más opciones.

| Introduce list(0,"Hola",TRUE) en la línea de comandos.

```
> list(0,"Hola",TRUE)
```

```
[[1]]
```

```
[1] 0
```

```
[[2]]
```

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

```
[[3]]
```

[1] TRUE

| ¡Es asombroso!

|=====|  
63%

| Anteriormente viste que en R los vectores sólo pueden contener objetos de la misma clase.

...

|=====|  
64%

| Pero, ¿qué pasa si creas un vector `c(T, 19, 1+3i)`? Introduce `c(T, 19, 1+3i)` en la línea de comandos.

> `c(T, 19, 1+3i)`

[1] 1+0i 19+0i 1+3i

| ¡Mantén este buen nivel!

|=====|  
65%

| Como habrás supuesto, el número complejo `1+3i` no puede ser convertido a entero ni a objeto de tipo "logical", entonces los valores `T` y

| `19` son convertidos a los números complejos `1+0i` y `19+0i` respectivamente. Esto no es más que la representación de esos valores en objeto

| tipo "complex".

...

|=====|  
66%

| Esto se llama coerción.

...

|=====|  
67%

| La coerción hace que todos los objetos de un vector sean de una misma clase. Entonces, cuando creas un vector de diferentes tipos, R

| busca un tipo común, y los elementos que no son de ese tipo son convertidos.

...

===== |  
69%

| Otro ejemplo de coerción es cuando usas las funciones `as.*()`.

...

| ===== |  
70%

| Inténtalo: crea un vector de longitud 5 de tipo "numeric" con la función `vector()` y guardarlo en la variable 'c'.

> c <- vector("numeric", length = 5)

| ¡Eso es trabajo bien hecho!

| =====  
71%

| Revisa el contenido de la variable 'c' .

> c

[1] 0 0 0 0 0

| ¡Buen trabajo!

| ===== |  
72%

| Ahora usa la función `as.logical()` con el vector c.

> as.logical(c)

[1] FALSE FALSE FALSE FALSE FALSE

| ¡Lo has logrado! ¡Buen trabajo!

| =====  
73%

| Como puedes imaginar, el vector de tipo "numeric" fue explícitamente convertido a "logical".

...

| =====  
74%

| Este tipo de coerción es mejor conocida como coerción explícita. Además de `as.logical()`, también existe `as.numeric()`, `as.character()`,

| `as.integer()`.

...

```
|=====
| 75%

| Si usas la función class(), que te dice la clase a la que pertenece un objeto, obtendrás que class(c)
= "numeric." Pruébalo, ingresa

| class(c) en la línea de comandos.

> class(c)

[1] "numeric"

| ¡Es asombroso!

|=====
| 76%

| Pero si después pruebas la misma función class() enviándole como argumento as.logical(c),
obtendrás que es de tipo logical. Compruébalo:

> as.logical(c)

[1] FALSE FALSE FALSE FALSE FALSE

| No exactamente. Dele otra oportunidad. O escribe info() para más opciones.

| Introduce class(as.logical(c)) en la línea de comandos.

> class(as.logical(c))

[1] "logical"

| ¡Muy bien!

|=====
| 78%

| Además de los vectores y las listas, existen las matrices.

...

|=====
| 79%

| Una matriz es una extensión de un vector de dos dimensiones. Las matrices son usadas para
representar información de un solo tipo de dos

| dimensiones.

...

|=====
| 80%
```

| Una manera de generar una matriz es al usar la función matrix(). Inténtalo, introduce m <- matrix(data=1:12,nrow=4,ncol=3) en la línea

| de comandos.

>

> m <- matrix(data=1:12,nrow=4,ncol=3)

| ¡Es asombroso!

| =====  
| 81%

| Ahora imprime el contenido de 'm'.

> print(m)

    [,1] [,2] [,3]  
[1,]  1  5  9  
[2,]  2  6 10  
[3,]  3  7 11  
[4,]  4  8 12

| ¡Toda esa práctica está rindiendo frutos!

| =====  
| 82%

| Como puedes observar, creaste una matriz con tres columnas (ncol) y cuatro renglones (nrow).

...

| =====  
| 83%

| Recuerda que también puedes crear matrices con las funciones cbind, rbind y as.matrix().

...

| =====  
| 84%

| Los factores son otro tipo especial de vectores usados para representar datos categóricos, éstos pueden ser ordenados o sin orden.

...

| =====  
| 85%

| Recuerda el vector 'colores' que creaste previamente y supón que representa un conjunto de observaciones acerca de cuál es el color

| preferido de las personas.

```
...
|=====
|=      | 87%
```

| Es una representación perfectamente válida, pero puede llegar a ser ineficiente. Ahora representarás los colores como un factor.

| Introduce factor(colores) en la línea de comandos.

```
> factor(colores)
```

```
[1] rojo azul verde azul rojo
```

```
Levels: azul rojo verde
```

```
| ¡Buen trabajo!
|=====
===    | 88%
```

| La impresión de un factor muestra información ligeramente diferente a la de un vector de caracteres. En particular, puedes notar que las

| comillas no son mostradas y que los niveles son explícitamente impresos.

```
...
|=====
====   | 89%
```

| Por último, existen los dataframes, que son una manera muy útil de representar datos tabulares. Son uno de los tipos más importantes.

```
...
|=====
===== | 90%
```

| Un dataframe representa una tabla de datos. Cada columna de éste puede ser de un tipo diferente, pero cada fila debe tener la misma

| longitud.

```
...
|=====
===== | 91%
```

| Ahora crea uno. Introduce data.frame(llave=y, color=colores) en la línea de comandos.

```
> data.frame(llave=y, color=colores)
```

llave color

1 561 rojo

2 1105 azul

3 1729 verde

4 2465 azul

5 2821 rojo

| ¡Eso es correcto!

|=====

===== | 92%

| ¿Recuerdas los vectores 'y' y 'colores'? Pues con ellos creaste un data frame cuya primera columna tiene números de Carmichael y la

| segunda colores.

...

|=====

===== | 93%

| Otra manera de crear dataframes es con las funciones read.table() y read.csv().

...

|=====

===== | 94%

| También puedes usar la función data.matrix() para convertir un data frame en una matriz.

...

|=====

===== | 96%

| Antes de concluir la lección, te mostraré un par de atajos.

...

|=====

===== | 97%

| Al inicio de esta lección introdujiste `mi_variable <- (180 / 6) - 15` en la línea de comandos. Supón que cometiste un error y que querías

| introducir `mi_variable <- (180 / 60) - 15`, es decir, querías escribir 60, pero escribiste 6. Puedes reescribir la expresión o...

...

|=====

===== | 98%

| En muchos entornos de programación, presionar la tecla 'flecha hacia arriba' te mostrará comandos anteriores. Presiona esta tecla hasta

| que llegues al comando (mi\_variable <- (180 / 6) - 15), entonces cambia el número 6 por 60 y presiona ENTER. Si la tecla 'flecha hacia

| arriba' no funciona, sólo escribe el comando correcto.

> mi\_variable <- (180 / 60) - 15

| ¡Sigue trabajando de esa manera y llegarás lejos!

|=====

===== | 99%

| Por último, puedes teclear las dos primeras letras del nombre de la variable y después presionar la tecla Tab (tabulador). La mayoría de

| los entornos de programación muestran una lista de las variables que has creado con el prefijo 'mi\_'. Esta función se llama

| autocompletado y es muy útil para cuando tienes muchas variables en tu espacio de trabajo. Pruébalo, ingresa 'mi\_' y autocompleta. Si

| autocompletar no sirve en tu caso, sólo ingresa mi\_variable en la línea de comandos).

> mi\_variable

[1] -12

| Perseverancia es la respuesta.

|=====

===== | 100%

| Has concluido la lección. ¿Te gustaría que se le notificará a Coursera que has completado esta lección?

1: Si

2: No

Selection: 1

¿Cuál es tu nombre de usuario registrado en Coursera (email)? migevi97@gmail.com

¿Cuál es tu token de la tarea? LTKjlaTsvlbhSokI

¿El envío de la calificación fue satisfactorio!



### 3.- Subconjunto de Datos.

Selection: 3

| 0%

| En esta lección conocerás las maneras de acceder a las estructuras de datos en el lenguaje R.

...

| == | 2%

| R tiene una sintaxis especializada para acceder a las estructuras de datos.

...

| ===== | 4%

| Tú puedes obtener un elemento o múltiples elementos de una estructura de datos usando la notación de indexado de R.

...

| ===== | 5%

| R provee diferentes maneras de referirse a un elemento (o conjunto de elementos) de un vector. Para probar estas diferentes maneras crea

| una variable llamada 'mi\_vector' que contenga un vector con los números enteros del 11 al 30. Recuerda que puedes usar el operador

| secuencia ':'.  
> mi\_vector<- c(11:30)

| ¡Eso es trabajo bien hecho!

| ===== | 7%

| Y ahora ve su contenido.

> mi\_vector

[1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

| ¡Es asombroso!

| ===== | 9%

| La manera más común de buscar un elemento en R es por medio de un vector numérico.

...

| ===== | 11%

| Puedes buscar elementos por posición en un vector usando la siguiente notación: `x[s]`, donde 'x' es un vector del cual deseas obtener

| elementos y 's' es un segundo vector representando el conjunto de índices de elementos que te gustaría consultar.

...

| ===== | 12%

| Debes saber que en R las posiciones de los elementos de un vector comienzan en 1 y no en 0, como en lenguajes de programación como Java

| o C.

...

| ===== | 14%

| Puedes usar un vector entero para buscar un simple elemento o múltiples.

...

| ===== | 16%

| Por ejemplo, obten el tercer elemento de 'mi\_vector'.

```
> mi_vector[3]
```

```
[1] 13
```

| Perseverancia es la respuesta.

| ===== | 18%

| Ahora obten los primeros cinco elementos de 'mi\_vector'.

```
> mi_vector[0:5]
```

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

| No es la respuesta que yo buscaba, pero intenta nuevamente. O escribe info() para más opciones.

| Introduce mi\_vector[1:5] en la línea de comandos.

```
> mi_vector[1:5]
```

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

| ¡Toda esa práctica está rindiendo frutos!

| ===== | 19%

| No necesariamente los índices deben ser consecutivos. Ingresas mi\_vector[c(4,6,13)] en la línea de comandos.

```
>
```

```
> mi_vector[c(4,6,13)]
```

```
[1] 14 16 23
```

| ¡Excelente trabajo!

| ===== | 21%

| Asimismo, no es necesario que los índices se encuentren ordenados. Ingresas mi\_vector[c(6,13,4)] en la línea de comandos.

```
> mi_vector[c(6,13,4)]
```

```
[1] 16 23 14
```

| ¡Eso es correcto!

| ===== |  
23%

| Como un caso especial, puedes usar la notación [[]] para referirte a un solo elemento. Ingresas mi\_vector[[3]] en la línea de comandos.

```
> mi_vector[[3]]
```

```
[1] 13
```

| ¡Tu dedicación es inspiradora!

| ===== |  
25%

| La notación [[]] funciona de la misma manera que la notación [] en este caso.

...

|=====

26%

| También puedes usar enteros negativos para obtener un vector que consista en todos los elementos, excepto los elementos especificados.

| Excluye los elementos 9:15, al especificar -9:-15.

> -9:-15

[1] -9 -10 -11 -12 -13 -14 -15

| Estás muy cerca... ¡Puedo sentirlo! Inténtalo de nuevo. O escribe info() para más opciones.

| Introduce mi\_vector[-9:-15] en la línea de comandos.

> mi\_vector[-9:-15]

[1] 11 12 13 14 15 16 17 18 26 27 28 29 30

| ¡Tu dedicación es inspiradora!

|=====

| 28%

| Como alternativa a indexar con un vector de enteros, puedes indexar a través de un vector lógico.

...

|=====

| 30%

| Como ejemplo crea un vector lógico de longitud 10 con valores lógicos alternados, TRUE y FALSE (rep(c(TRUE,FALSE),10)), y consulta con

| él mi\_vector[rep(c(TRUE,FALSE),10)].

> d

Error: object 'd' not found

> mi\_vector[rep(c(TRUE,FALSE),10)]

[1] 11 13 15 17 19 21 23 25 27 29

| ¡Bien hecho!

| =====  
| 32%

| Como podrás notar, lo que ocurrió fue que indexaste únicamente los elementos en las posiciones impares, puesto que creaste un vector con

| elementos TRUE en las posiciones impares y FALSE en las pares.

...

| =====  
| 33%

| El vector índice no necesita ser de la misma longitud que el vector a indexar. R repetirá el vector más corto y regresará los valores

| que cacen. Ingresas mi\_vector[c(FALSE,FALSE,TRUE)] en la línea de comandos.

> mi\_vector[c(FALSE,FALSE,TRUE)]

[1] 13 16 19 22 25 28

| ¡Acertaste!

| =====  
| 35%

| Notarás que ahora indexaste los índices de los elementos múltiplos de 3.

...

| =====  
| 37%

| Es muy útil calcular un vector lógico de un mismo vector. Por ejemplo, busca elementos más grandes que 20. Ingresas en la línea de

| comandos mi\_vector > 20.

> mi\_vector > 20

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

| ¡Es asombroso!

```
| =====
| 39%
```

| Y ahora indexa 'mi\_vector' usando el vector previamente calculado. Ingresas  
mi\_vector[(mi\_vector > 20)] en la línea de comandos.

```
> mi_vector[(mi_vector > 20)]
[1] 21 22 23 24 25 26 27 28 29 30
```

| ¡Eres el mejor!

```
| =====
| 40%
```

| También puedes usar esta notación para extraer partes de una estructura de datos  
multidimensional.

...

```
| =====
| 42%
```

| Un arreglo es un vector multidimensional. Vectores y arreglos se almacenan de la misma manera  
internamente, pero un arreglo se muestra

| diferente y se accede diferente.

...

```
| =====
| 44%
```

| Para crear un arreglo de dimensión 3x3x2 y de contenido los números del 1 al 18 y guardarlo en la variable 'mi\_arreglo', ingresa

| `mi_arreglo <- array(c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18),dim=c(3,3,2))` en la línea de comandos.

> `mi_arreglo <- array(c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18),dim=c(3,3,2))`

| Esa es la respuesta que estaba buscando.

| =====  
| 46%

| Ahora ve el contenido de la variable 'mi\_arreglo'.

> `mi_arreglo`

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

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

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

[1,] 10 13 16

[2,] 11 14 17

[3,] 12 15 18

| ¡Eso es trabajo bien hecho!

| =====  
| 47%

| R tiene una manera muy limpia de referirse a parte de un arreglo. Se especifican índices para cada dimensión, separados por comas.

| Ingresa `mi_arreglo[1,3,2]` en la línea de comandos.

> `mi_arreglo[1,3,2]`

[1] 16

| ¡Eres el mejor!

| =====  
| 49%

| Asimismo, puedes ingresar `mi_arreglo[1:2,1:2,1]` en la línea de comandos. ¡Inténtalo!

```
> mi_arreglo[1:2,1:2,1]
```

```
  [,1] [,2]
```

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

```
[2,]  2  5
```

| Perseverancia es la respuesta.

```
|=====
```

```
| 51%
```

| Una matriz es simplemente un arreglo bidimensional. Ahora crea una matriz con 3 renglones y 3 columnas con los números enteros del 1 al

| 9 y guárdala en la variable 'mi\_matriz'.

```
> mi_matriz <- array(c(1:9), dim(3,3,1))
```

```
Error in dim(3, 3, 1) : 3 arguments passed to 'dim' which requires 1
```

```
> mi_matriz <- array(c(1:9), dim=c(3,3,1))
```

| Una vez más. ¡Tú puedes hacerlo! O escribe `info()` para más opciones.

| Ingresa `mi_matriz <- matrix(data=1:9,nrow=3,ncol=3)` en la línea de comandos.

```
> mi_matriz <- matrix(data=1:9,nrow=3,ncol=3)
```

| Esa es la respuesta que estaba buscando.

```
|=====
```

```
| 53%
```



| Al igual que con los arreglos, para obtener todos los renglones o columnas de una dimensión de una matriz, simplemente omite los

| índices.

...

| =====  
| 54%

| Por ejemplo, si quisieras solo el primer renglón de 'mi\_matriz', basta con ingresar mi\_matriz[1,] en la línea de comandos. ¡Inténtalo!

> mi\_matriz[1,]

[1] 1 4 7

| ¡Excelente!

| =====  
| 56%

| ¡Ahora obtén solo la primera columna!

> mi\_matriz[,1]

[1] 1 2 3

| ¡Es asombroso!

| =====  
| 58%

| También puedes referirte a un rango de renglones. Ingresa mi\_matriz[2:3,] en la línea de comandos.

> Ingresa mi\_matriz[2:3,]

Error: unexpected symbol in "Ingresa mi\_matriz"

> mi\_matriz[2:3,]

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

[1,] 2 5 8

[2,] 3 6 9

| ¡Sigue trabajando de esa manera y llegarás lejos!

| =====  
| 60%

| O referirte a un conjunto no contiguo de renglones. Ingresa mi\_matriz[c(1,3),] en la línea de comandos.

```
> mi_matriz[c(1,3),]
```

```
  [,1] [,2] [,3]
```

```
[1,]  1   4   7
```

```
[2,]  3   6   9
```

| ¡Eso es trabajo bien hecho!

| =====  
| 61%

| En los ejemplos de arriba solo has visto estructuras de datos basadas en un solo tipo. Recuerda que R tiene un tipo de datos incorporado

| para la mezcla de objetos de diferentes tipos, llamados listas.

...  
| =====  
== | 63%

| Debes de saber que en R las listas son sutilmente diferentes de las listas en muchos otros lenguajes. Las listas en R contienen una

| selección heterogénea de objetos. Puedes nombrar cada componente en una lista.

...  
| =====  
==== | 65%

| Los elementos en una lista pueden ser referidos por su ubicación o por su nombre.

..  
| =====  
===== | 67%

| Ingresa este ejemplo de una lista con cuatro componentes nombrados carro <- list(color="rojo", nllantas=4, marca= "Renault",

| ncilindros=4).

```
> carro <- list(color="rojo", nllantas=4, marca= "Renault",ncilindros=4)
```

| ¡Buen trabajo!

| =====  
===== | 68%

| Tú puedes acceder a los elementos de una lista de múltiples formas. Puedes usar la misma notación que usaste con los vectores.

```
...
|=====
===== | 70%
```

| Y además puedes indexar un elemento por nombre usando la notación \$. Por ejemplo, ingresa carro\$color en la línea de comandos.

```
> carro$colors
```

```
NULL
```

| ¡Sigue intentando! O escribe info() para más opciones.

```
> carro$colors
```

```
NULL
```

| Eso no es precisamente lo que buscaba. Trata otra vez. O escribe info() para más opciones.

```
> carro$color
```

```
[1] "rojo"
```

| ¡Todo ese trabajo está rindiendo frutos!

```
|=====
===== | 72%
```

| Además, puedes usar la notación [] para indexar un conjunto de elementos por nombre. Ingresa carro[c("ncilindros","nllantas")] en la

| línea de comandos.

```
> carro[c("ncilindros","nllantas")]
```

```
$ncilindros
```

```
[1] 4
```

```
$nllantas
```

```
[1] 4
```

| ¡Traes una muy buena racha!

```
|=====
===== | 74%
```

| También puedes indexar por nombre usando la notación [[]] cuando seleccionas un simple elemento. Por ejemplo, ingresa carro[["marca"]]

| en la línea de comandos.

```
> carro[["marca"]]
```

```
[1] "Renault"
```

```
| ¡Eso es trabajo bien hecho!  
|=====
```

```
===== | 75%  
  
| Hasta puedes indexar por nombre parcial usando la opción exact=FALSE. Ingresar  
carro[["mar",exact=FALSE]] en la línea de comandos.
```

```
> carro[["mar",exact=FALSE]]
```

```
[1] "Renault"
```

```
| ¡Eres bastante bueno!  
|=====
```

```
===== | 77%  
  
| Ahora crea la siguiente lista: camioneta <- list(color="azul", nllantas=4, marca= "BMW",  
ncilindros=6).
```

```
> camioneta <- list(color="azul", nllantas=4, marca= "BMW", ncilindros=6)
```

```
| ¡Muy bien!  
|=====
```

```
===== | 79%  
  
| Algunas veces una lista será una lista de listas. Ingresar cochera <- list(carro, camioneta).
```

```
> cochera <- list(carro, camioneta)
```

```
| ¡Muy bien!  
|=====
```

```
===== | 81%  
  
| Ahora ve el contenido de 'cochera'.
```

```
> cochera
```

```
[[1]]
```

```
[[1]]$color
```

```
[1] "rojo"
```

```
[[1]]$nllantas
```

```
[1] 4
```

```
[[1]]$marca
```

```
[1] "Renault"
```

```
[[1]]$ncilindros
```

```
[1] 4
```

```
[[2]]
```

```
[[2]]$color
```

```
[1] "azul"
```

```
[[2]]$nllantas
```

```
[1] 4
```

```
[[2]]$marca
```

```
[1] "BMW"
```

```
[[2]]$ncilindros
```

```
[1] 6
```

```
| Perseverancia es la respuesta.
```

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

| Tú puedes usar la notación `[[ ]]` para referirte a un elemento en este tipo de estructura de datos. Para hacer esto usa un vector como

| argumento. R iterará a través de los elementos en el vector referenciando sublistas.

```
...
```

```
|=====
===== | 84%
```

| Ingresar `cochera[[c(2, 1)]]` en la línea de comandos.

```
>
```

```
> cochera[[c(2, 1)]]
```

```
[1] "azul"
```

```
| ¡Sigue trabajando de esa manera y llegarás lejos!
```

```
|=====
===== | 86%
```

| Recuerda que los data frames son una lista que contiene múltiples vectores nombrados que tienen la misma longitud. A partir de este

| momento usarás el data frame `cars` del paquete `datasets`. No te preocupes, este paquete viene cargado por defecto.

```
...
```

```
|=====
===== | 88%
```

| Los datos que conforman al data frame cars son un conjunto de observaciones tomadas en la década de 1920; estas observaciones describen

| la velocidad (mph) de algunos carros y la distancia (ft) que les tomó parar.

...

```
|=====
===== | 89%
```

| Ve el contenido del data frame cars. Ingresa cars en la línea de comandos.

```
> 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

| ¡Eso es correcto!

|=====

===== | 91%

| Te puedes referir a los elementos de un data frame (o a los elementos de una lista) por nombre usando el operador \$. Ingresa cars\$speed

| en la línea de comandos.

> cars\$speed

[1] 4 4 7 7 8 9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 16 16 17 17 17 18  
18 18 18 19 19 19 20 20 20 20 20 22 23

[46] 24 24 24 24 25

| ¡Buen trabajo!

|=====

===== | 93%

| Supón que deseas saber a qué velocidad iban los carros a los que les tomó más de 100 pies (ft) frenar.

...

|=====

===== | 95%

| Una manera de encontrar valores específicos en un data frame es al usar un vector de valores booleanos para especificar cuál o cuáles

| elementos regresar de la lista. La manera de calcular el vector apropiado es así: cars\$dist>100. ¡Inténtalo!

> cars\$dist>100

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[45] FALSE FALSE FALSE FALSE TRUE FALSE

| ¡Excelente!

|=====

===== | 96%

| Entonces puedes usar ese vector para referirte al elemento correcto. Ingresa cars\$speed[cars\$dist>100] en la línea de comandos.

> cars\$speed[cars\$dist>100]



[1] 24

| ¡Lo has logrado! ¡Buen trabajo!

|=====

===== | 98%

| Ahora ya sabes cómo acceder a las estructuras de datos.

...

|=====

===== | 100%

| Has concluido la lección. ¿Te gustaría que se le notificará a Coursera que has completado esta lección?

1: No

2: Si

Selection: 2

¿Cuál es tu nombre de usuario registrado en Coursera (email)? migevi97@gmail.com

¿Cuál es tu token de la tarea? gxqV0wQd4cr0LohR

¿El envío de la calificación fue satisfactorio!

## 4.- Leer y Escribir Datos.

Selection: 4

| 0%

| En esta lección conocerás cómo cargar conjuntos de datos en R y guardar estos conjuntos desde R.

...

|=== 2%

| Una de las mejores cosas acerca de R es lo fácil que es añadir información desde otros programas.

...

|===== 4%

| R puede importar conjuntos de datos desde archivos de texto, otros softwares de estadística y hasta hojas de cálculo. No es necesario

| tener una copia local del archivo. Tú puedes especificar la ubicación del archivo desde una url y R buscará el archivo en Internet.

...

|===== 7%

| La mayoría de los archivos que contienen información tienen un formato similar. Generalmente cada línea del archivo representa una

| observación o registro, por lo que cada línea contiene un conjunto de diferentes variables asociadas con la observación.

...

|===== 9%

| Algunas veces, diferentes variables son separadas por un carácter especial, llamado delimitador. Otra veces las variables son

| diferenciadas por su ubicación en cada línea.

...

|===== 11%

| En esta lección trabajarás con el archivo inmigtntalpry.csv el cual contiene la estimación de personas provenientes de otros países que

| llegan a cada uno de los estados de México. Si tienes suerte, el archivo se mostrará en algún editor; de lo contrario búscalo en el

| subdirectorio swirl\_temp de tu directorio de trabajo y velo en una aplicación separada.

(Se ha copiado el archivo inmigintnalpry.csv a la ruta  
C:/Users/Migevi/Documents/swirl\_temp/inmigintnalpry.csv ).

...

|===== | 13%

| Como podrás notar el primer renglón del archivo contiene los nombres de las columnas, en este caso los nombres de cada una de las

| variables de la observación; además, el archivo tiene delimitada cada variable de la observación por una coma.

...

|===== | 15%

| Para cargar este archivo a R, debes especificar que el primer renglón contiene los nombres de las columnas y que el delimitador es una

| coma.

...

|===== | 17%

| Para hacer esto necesitarás especificar los argumentos header y sep en la función read.table. Header para especificar que el primer

| renglón contiene los nombres de la columnas (header=TRUE) y sep para especificar el delimitador (sep=",").

...

|===== | 20%

| ¡Importa el archivo inmigintnalpry.csv! Ingresa datos <-  
read.table("swirl\_temp/inmigintnalpry.csv", header=TRUE, sep=",", fileEncoding

| = "latin1") en la línea de comandos.

>

> datos <- read.table("swirl\_temp/inmigintnalpry.csv", header=TRUE, sep=",", fileEncoding

+ | = "latin1")

Error: unexpected '=' in:

"datos <- read.table("swirl\_temp/inmigintnalpry.csv", header=TRUE, sep=",", fileEncoding

| ="

```
> datos <- read.table("swirl_temp/inmigintnalpry.csv", header=TRUE, sep=",", fileEncoding =  
"latin1")
```

| ¡Eres bastante bueno!

| =====

22%

| Como podrás notar usaste el argumento fileEncoding; esto debido a que de no usarlo R no  
podría importar el archivo, puesto que la

| segunda cadena del archivo: año, no es una cadena válida para el tipo de codificación que  
read.table usa por defecto. Para poder leer el

| archivo basta con especificar el argumento fileEncoding. De no especificarlo R te indicará que  
hay un error.

...

| =====

24%

| Intenta usar datos\_2 <- read.table("swirl\_temp/inmigintnalpry.csv", header=TRUE, sep=",").  
Debido a que el archivo inmigintnalpry.csv

| contiene caracteres especiales como la ñ, R PUEDE MOSTRARTE UN ERROR. Si R te muestra el  
error, ingresa ok() en la línea de comandos

| para continuar.

```
> ok()
```

| ¡Eres el mejor!

| =====

26%

| Este error es muy común cuando intentas leer archivos que su contenido está en español; esto  
se debe a que usa otra codificación para

| poder abarcar más símbolos que no usan otros idiomas, como en este caso la ñ. Para poder leer  
archivos que contengan ñ, basta con

| especificar el argumento fileEncoding, el cual indica la codificación del archivo a importar; en  
este caso, usarás fileEncoding =

| "latin1".

...

```
|=====
| 28%
```

| Comúnmente las opciones más importantes son sep y header. Casi siempre debes saber el campo separador y si hay un campo header.

...

```
|=====
| 30%
```

| Ahora ve lo que contiene 'datos'. Para hacer esto usarás la función View(). Si te encuentras en Rstudio simplemente puedes presionar el

| nombre de la variable datos en el apartado Entorno ('Environment') y te mostrará su contenido. Presiona la variable datos en Rstudio o

| ingresa View(datos) en la línea de comandos.

```
> View(datos)
```

| ¡Traes una muy buena racha!

```
|=====
| 33%
```

| ¡Como podrás notar el archivo contiene 302060 observaciones!

...

```
|=====
| 35%
```

| Es importante saber que no solo existe read.table(). R además incluye un conjunto de funciones que llaman a read.table() con diferentes

| opciones por defecto para valores como sep y header, y algunos otros. En la mayoría de los casos encontrarás que puedes usar read.csv()

| para archivos separados por comas o read.delim() para archivos delimitados por TAB sin especificar otras opciones.

...

```
|=====
| 37%
```

| La mayoría de las veces deberías ser capaz de cargar archivos de texto en R con la función read.table(). Pero algunas veces serás

| proveído con un archivo de texto que no pueda ser leído correctamente con esta función.

...

|=====

| 39%

| Si estás en Europa y usas comas para indicar punto decimal en los números, entonces puedes usar `read.csv2()` y `read.delim2()`.

...

|=====

| 41%

| Una manera de agilizar la lectura de datos es usando el parámetro `colClasses` de la función `read.table()`.

...

|=====

| 43%

| Este parámetro recibe un vector, el cual describe cada uno de los tipos por columna que va a leer. Esto agiliza la lectura debido a que

| `read.table()` normalmente lee toda la información y después revisa cada una de las columnas, y decide conforme a lo que vio de qué tipo

| es cada columna, y al indicar el parámetro `colClasses` le dices a la función `read.table()` de qué tipo son los datos que va a ver, con lo

| que te evitas el chequeo para saber el tipo de cada columna.

...

|=====

| 46%

| Puedes averiguar la clase de las columnas de manera fácil cuando tienes archivos grandes.

...

|=====

| 48%

| Lo que puedes hacer es indicarle a `read.table()` que solo lea los primeros 100 renglones del archivo; esto lo haces indicando el

| parámetro `nrow`. Cabe recordar que debes especificar la codificación del archivo, debido a que usa caracteres especiales, también que el

| primer renglón son los nombres de la columnas y que el delimitador es una coma. Ingresas inicial <-

| `read.table("swirl_temp/inmigintnalpry.csv", header=TRUE, sep=",", fileEncoding = "latin1", nrow = 100)` en la línea de comandos.

```
> inicial <- read.table("swirl_temp/inmigintnalpry.csv", header=TRUE, sep=",", fileEncoding =  
"latin1", nrow = 100)
```

| ¡Lo has logrado! ¡Buen trabajo!

| =====  
| 50%

| Con esto has conseguido leer las primeras 100 observaciones.

...

| =====  
| 52%

| Después usa la función `sapply` mandándole como parámetros el objeto inicial (el cual contiene las 100 observaciones) y la función

| `class()`. Ingresa `clases <- sapply(inicial, class)` en la línea de comandos.

```
> clases <- sapply(inicial, class)
```

| ¡Traes una muy buena racha!

| =====  
| 54%

| Con esto lo que conseguiste fue aplicar la función `class()` a cada una de las columnas del objeto inicial. La función `class()` es una

| función que determina la clase o tipo de un objeto. Entonces los tipos de cada una de las columnas fueron guardados en el objeto `clases`.

...

| =====  
| 57%

| Para ver el contenido del objeto 'clases', basta con escribir `clases` en la línea de comandos.

```
> clases
```

```
  renglon   año   ent  id_ent  cvegeo   sexo  edad inmigintnal  
"integer" "integer" "factor" "integer" "integer" "factor" "integer" "numeric"
```

| ¡Todo ese trabajo está rindiendo frutos!

| =====  
| 59%

| Por último, con este vector de clases, leerás todo el archivo usando la función `read.table`, pero pasándole el argumento `colClasses`.

| Ingresa datos <- read.table("swirl\_temp/inmigintnalpry.csv", header=TRUE, sep=";", fileEncoding = "latin1", colClasses=clases) en la

| línea de comandos.

```
> datos <- read.table("swirl_temp/inmigintnalpry.csv", header=TRUE, sep=";", fileEncoding = "latin1", colClasses=clases)
```

| ¡Lo estás haciendo muy bien!

```
|=====
| 61%
```

| Como podrás notar el tiempo de lectura mejoró significativamente usando este truco.

```
...
|=====
==                                     | 63%
```

| Si deseas guardar objetos, la manera más simple es usando la función save(). Por ejemplo, puedes usar el siguiente comando para salvar

| el objeto 'datos' y el objeto 'clases' en el archivo swirl\_temp/datos\_inmigrates.RData. Ingresa save(datos, clases,

| file="swirl\_temp/datos\_inmigrates.RData") en la línea de comandos.

```
> save(datos, clases, file="swirl_temp/datos_inmigrates.RData")
```

| ¡Toda esa práctica está rindiendo frutos!

```
|=====
=====                               | 65%
```

| La función save() escribe una representación externa de los objetos especificados a un archivo señalado. Además, como ya te habrás dado

| cuenta, tú puedes guardar múltiples objetos en el mismo archivo, tan solo al listarlos en la función save().

```
...
|=====
=====                               | 67%
```

| Es importante notar que en R, las rutas de archivo siempre son especificadas con diagonales ("/"), aun estando en Microsoft Windows. Así

| que para salvar este archivo al directorio "C:\Documents and Settings\Mi Usuario\Mis Documentos\datos\_inmigrates.RData, solo usarías el

| siguiente comando: save(datos,file="C:/Documents and Settings/Mi Usuario/Mis Documentos/datos\_inmigrates.RData").

...



```
|=====
===== | 70%
```

| También es importante notar que el argumento file debe ser explícitamente nombrado.

...

```
|=====
===== | 72%
```

| Ahora que has guardado los objetos 'datos' y 'clases' en un archivo, puedes borrarlos. Introduce rm(datos,clases) en la línea de

| comandos.

```
> rm(datos,clases)
```

| ¡Es asombroso!

```
|=====
===== | 74%
```

| Y si ahora usas la función ls(), la cual como recordarás muestra qué conjuntos de datos y funciones un usuario ha definido, verás que no

| están presentes los objetos datos y clases. Ingresas ls() en la línea de comandos.

```
>
```

```
> ls()
```

```
[1] "c"           "camioneta"    "carro"        "cochera"      "colores"
[6] "colores"     "complejo"     "display_swirl_file" "empleados"
"fecha_primer_curso_R"
[11] "find_course" "inicial"      "m"            "mi_arreglo"   "mi_matriz"
[16] "mi_variable" "mi_vector"    "mod"          "num"          "ok"
[21] "parametro"   "salario"      "verosimilitud" "y"
```

| ¡Buen trabajo!

```
|=====
===== | 76%
```

| Ahora, puedes fácilmente cargar los objetos 'datos' y 'clases' devuelta a R con la función load(). Solo debes especificar el nombre del

| archivo donde los guardaste. Ingresas load("swirl\_temp/datos\_inmigrantes.RData") en la línea de comandos.

```
> load("swirl_temp/datos_inmigrantes.RData")
```

```
| ¡Eso es trabajo bien hecho!  
|=====
```

```
=====| 78%  
  
| Y si ahora usas la función ls(), verás que están presentes los objetos 'datos' y 'clases'. Ingresa ls()  
en la línea de comandos.
```

```
> ls()
```

```
[1] "c"          "camioneta"  "carro"      "clases"    "cochera"  
[6] "colores"    "colores"    "complejo"    "datos"      "display_swirl_file"  
[11] "empleados"  "fecha_primer_curso_R" "find_course" "inicial"    "m"  
[16] "mi_arreglo" "mi_matriz"  "mi_variable" "mi_vector"  "mod"  
[21] "num"        "ok"         "parametro"  "salario"    "verosimilitud"  
[26] "y"
```

```
| ¡Acertaste!  
|=====
```

```
=====| 80%  
  
| Es importante saber que los archivos guardados en R funcionarán en todas las plataformas; es  
decir, archivos guardados en Linux
```

```
| funcionarán si son cargados desde Windows o Mac OS X.
```

```
...  
|=====
```

```
=====| 83%  
  
| Si deseas guardar cada uno de los objetos de tu espacio de trabajo (workspace), puedes hacerlo  
usando la función save.image(). De hecho,
```

```
| cuando salgas de la session de R, se te preguntará si deseas salvar tu actual espacio de trabajo  
(workspace). Si señalas que sí lo
```

```
| deseas, tu espacio de trabajo será guardado de la misma manera que usar esta función.
```

```
...  
|=====
```

```
=====| 85%  
  
| Por último, al igual que para importar datos existe la función read.table(), para exportar datos a  
un archivo de texto existe la función
```

```
| write.table().
```

...  
|=====

---

===== | 87%

| Normalmente los datos a exportar son data frames y matrices.

...  
|=====

---

===== | 89%

| Para exportar un objeto a un archivo basta con escribir la función `write.table()` y como argumento el nombre del objeto, además del

| nombre del archivo donde se guardará. Ingresa `write.table(datos, file="swirl_temp/datos.txt")` en la línea de comandos.

> `write.table(datos, file="swirl_temp/datos.txt")`

| Esa es la respuesta que estaba buscando.

|=====

---

===== | 91%

| Si tienes suerte, te mostraré el archivo `datos.txt` en algún editor; de lo contrario, búscalo en el subdirectorio `swirl_temp` de tu

| directorio de trabajo y vélo en una aplicación separada.

...  
|=====

---

===== | 93%

| Como podrás notar el archivo `datos.txt` no es igual al archivo `inmigintnalpry.csv` que al inicio de esta lección te mostré. Una de las

| principales razones es que para escribir el objeto `datos` no especificaste un delimitador (`sep`) y por defecto R delimitó con espacios.

...  
|=====

---

===== | 96%

| Al igual que con la función `read.table()`, R incluye un conjunto de funciones que llaman a `write.table()` con diferentes opciones por

| defecto, como lo son `write.csv()` y `write.csv2()`.

...  
|=====

---

===== | 98%

| Si deseas, puedes jugar con las funciones `write.*()` para lograr que `datos.txt` sea idéntico a `inmigintnalpry.csv`. Recuerda que para ver

| los parámetros de `write.*()` puedes usar `help()`; por ejemplo, `help(write.csv)`.

...

|=====

=====| 100%

| Has concluido la lección. ¿Te gustaría que se le notificará a Coursera que has completado esta lección?

1: Si

2: No

Selection: 1

¿Cuál es tu nombre de usuario registrado en Coursera (email)? migevi97@gmail.com

¿Cuál es tu token de la tarea? GEQAGUXy8KzoGSbi

¡El envío de la calificación fue satisfactorio!

## 5.- Funciones.

Selection: 5

| | 0%

| En esta lección conocerás las funciones del lenguaje R.

...

|===| 3%

| En R las operaciones que hacen todo el trabajo son llamadas funciones.

...

|=====| 6%

| Una función es un objeto en R, que puede tomar como entrada algunos objetos (llamados argumentos de

| función) y puede regresar un objeto de salida.

...

|=====| 9%

| Las mayoría de las funciones son de la siguiente forma: `f(argumento_1, argumento_2, ...)`. Donde `f`

| es el nombre de la función y `argumento_1, argumento_2, ...` son argumentos para la función.

...

|=====| 12%

| Has usado alguna función anteriormente, ya que no se puede hacer nada interesante sin ellas. Todo

| el trabajo en R es hecho por funciones.

...

|=====| 16%

| Una función que has estado usando a lo largo del curso es la función `c()`, la cual crea un vector de  
| los elementos que le sean pasados como argumentos. Por ejemplo introduce `c(1, 03, 2016)` en la  
línea

| de comandos.

> `c(1, 03, 2016)`

[1] 1 3 2016

| ¡Excelente!

|=====

| 19%

| La mayoría de las funciones en R regresan un valor; este valor puede ser calculado con base en el ambiente de la computadora o con base en la entrada (argumentos), como en este caso, en donde el valor regresado es el vector que contiene a 1, 3 y 2016.

...

|=====

| 22%

| Cada inicialización de variables en R, operaciones aritméticas, hasta repetir código en un loop, puede ser escrita como una función.

...

|=====

| 25%

| Las funciones son creadas usando la función especial function() y una vez creadas son guardadas como objetos de R de clase tipo function.

...

|=====

| 28%

| En la siguiente pregunta se te pedirá que modifiques un script. Las instrucciones de lo que debes hacer se encontrarán en el script. Una vez que hayas acabado de modificar el script, guarda tus cambios e ingresa submit() en la línea de comandos y así el script será evaluado. Si después de hacer esto la línea de comandos te dice que lo vuelvas a intentar y el script nuevamente aparece, esto se debe a que debes corregir tu script, siéntete libre de hacerlo, solo no olvides ingresar submit() cada vez que guardes tus cambios.

...

|=====

| 31%

| Generalmente el cuerpo de la función es encerrado entre llaves {}, pero no es necesario si el cuerpo es una simple expresión. Por ejemplo, la expresión sucesor <- function(x) x+1 es equivalente

| a la que se encuentra en el script.

> sucesor <- function(x) x+1

```
> sucesor <- function(x) {x+1}
```

```
> sucesor <- function(x)
```

```
+ x+1
```

```
> sucesor <- function(x) {
```

```
+   x + 1
```

```
+ }
```

```
> submit()
```

| Leyendo tu script...

| ¡Casi! Vuelve a intentar de nuevo.

| Asegúrate de haber borrado el símbolo # enfrente de la x, para que la última expresión sea x + 1.

```
> sucesor <- function(x) {
```

```
+   x + 1
```

```
+ }
```

```
> submit()
```

| Leyendo tu script...

| ¡Lo has logrado! ¡Buen trabajo!

| =====

| 34%

| ¡Ahora que has creado tu primera función ¡pruébala! Ingresa sucesor(5) en la línea de comandos.  
Si tu función

| funciona, debería de regresar únicamente el valor 6.

```
> sucesor(5)
```

```
[1] 6
```

| Perseverancia es la respuesta.

| =====

| 38%

| ¡Felicidades!, has escrito tu primera función.

...

| =====

| 41%

| Es importante que sepas que si deseas ver el código fuente de cualquier función, solo debes de  
teclear el

| nombre de la función sin argumentos ni paréntesis. Ahora ve el código fuente de la función que acabas de

| crear. Ingresa sucesor en la línea de comandos.

> sucesor

```
function(x) {  
  x + 1  
}
```

<bytecode: 0x000000001a954528>

| ¡Excelente!

|=====| 44%

| La definición de una función en R incluye los nombres de los argumentos, como en el caso anterior que

| nombraste a 'x'. Si especificas un valor por defecto para un argumento, entonces el argumento será considerado

| opcional.

...

|=====| 47%

| Ahora harás una función ligeramente más complicada, donde usarás argumentos por defecto. Crearás una función

| llamada diferencia\_cuadrada(). Recuerda que para elevar un número a cierta potencia se usa el operador binario

| `^`. Asegúrate de guardar tus cambios antes de ingresar submit() en la línea de comandos.

> submit()

| Leyendo tu script...

| Por poco era correcto, sigue intentándolo.

| Recuerda establecer el valor por defecto adecuado.

> submit()

| Leyendo tu script...

| Intenta de nuevo. ¡De cualquier forma hacerlo bien a la primera es aburrido!

| Recuerda establecer el valor por defecto adecuado.

> submit()



| Leyendo tu script...

| ¡Es asombroso!

| ===== | 50%

| Ahora prueba tu función `diferencia_cuadrada()`. Ingresas `diferencia_cuadrada(3)` en la línea de comandos.

```
> diferencia_cuadrada(3)
```

```
[1] 5
```

| ¡Es asombroso!

| ===== | 53%

| ¿Qué ha pasado? Como proveíste un solo argumento a la función, R cazó ese argumento a 'x', debido a que 'x' es

| el primer argumento. Por lo que 'y' usó el valor por defecto que definiste (2).

...

| ===== | 56%

| Recordarás que en una llamada a función puedes sobrescribir los valores por defecto. Así que ahora prueba

| `diferencia_cuadrada()` con dos argumentos. Ingresas `diferencia_cuadrada(10, 5)` en la línea de comandos.

```
> diferencia_cuadrada(10, 5)
```

```
[1] 75
```

| ¡Traes una muy buena racha!

| ===== | 59%

| En R puedes explícitamente nombrar a los argumentos. Por ejemplo ingresa `diferencia_cuadrada(y = 10, x = 5)` en

| la línea de comandos.

```
> diferencia_cuadrada(y = 10, x = 5)
```

```
[1] -75
```

| ¡Es asombroso!

| ===== | 62%

| Como podrás notar es diferente ingresar `diferencia_cuadrada(10, 5)` a `diferencia_cuadrada(y = 10, x = 5)`.

...

|=====|  
66%

| R también caza parcialmente los argumentos; es decir, ingresar diferencia\_cuadrada(10, y = 5) resulta en lo

| mismo que ingresar diferencia\_cuadrada(x = 10, y = 5) o diferencia\_cuadrada(10, 5).

...

|=====|  
69%

| Si no especificas un valor por defecto para un argumento, y si no especificas el valor de ese argumento cuando

| llamas a la función, obtendrás un error si la función intenta usar ese argumento.

...

|=====|  
72%

| Si deseas escribir una función que acepte un número variable de argumentos, en R puedes usar '...'; para hacer

| esto se especifica '...' en los argumentos de la función.

...

|=====|  
75%

| Ahora escribirás una función usando '...'. Cerciórate de guardar tus cambios en el script antes de que

| introduzcas submit().

> submit()

| Leyendo tu script...

| ¡Eres el mejor!

|=====|  
78%

| Ahora prueba tu función numeros\_por\_vocales. Usa la función numeros\_por\_vocales pasándole como argumentos las

| cadenas que desees.

>

> numeros\_por\_vocales("A","E")

[1] "4 3"

| ¡Lo has logrado! ¡Buen trabajo!

| =====  
| 81%

| Muchas funciones en R pueden recibir otras funciones como argumentos. Por ejemplo, si deseas saber los

| argumentos de una función puedes hacer uso de las funciones args() o formals(), las cuales reciben como

| argumento el nombre de la función de la que deseas conocer los argumentos.

...

| =====  
| 84%

| Ahora muestra los argumentos de la función mean(), la cual regresa el promedio de los elementos que recibe

| como argumentos. Usa cualquiera de la funciones antes mencionadas.

> args(mean)

function (x, ...)

NULL

| ¡Excelente!

| =====  
= | 88%

| Es importante que sepas que la función args() es usada principalmente de modo interactivo para imprimir los

| argumentos de una función. Para uso en programación considera mejor usar formals().

...

| =====  
==== | 91%

| El concepto de pasar funciones como argumentos es muy poderoso. Completa la función operador\_binario() para

| ver cómo funciona. Recuerda guardar tus cambios en el script antes de que introduzcas submit().

```
> submit()
```

```
| Leyendo tu script...
```

```
| ¡Mantén este buen nivel!
```

```
|=====
```

```
===== | 94%
```

```
| Ahora prueba tu función operador_binario(). Ingresá operador_binario(`%/`, 7, 3) en la línea de comandos.
```

```
| Recuerda que el operador `%/` no es más que la división entera en R.
```

```
> operador_binario(`%/`, 7, 3)
```

```
[1] 2
```

```
| ¡Bien hecho!
```

```
|=====
```

```
===== | 97%
```

```
| Por último, recuerda que todas las funciones en R regresan un valor. Algunas funciones en R además hacen otras
```

```
| cosas, como cambiar el estado de las variables, graficar, cargar o guardar archivos, o hasta acceder a la red.
```

```
...
```

```
|=====
```

```
===== | 100%
```

```
| Has concluido la lección. ¿Te gustaría que se le notificará a Coursera que has completado esta lección?
```

```
1: No
```

```
2: Si
```

```
Selection: 2
```

```
¿Cuál es tu nombre de usuario registrado en Coursera (email)? migevi97@gmail.com
```

```
¿Cuál es tu token de la tarea? k1YB7BEclwS2YbOJ
```

```
¡El envío de la calificación fue satisfactorio!
```