Notas de R y GIT

author: CM-274

Shell y bash

El Shell es un programa que toma comandos desde el teclado y los pasa al Sistema Operativo para realizar alguna operación.

Casi todos las distribuciones Linux,proporcionan un shell, desde el proyecto GNU, llamado `bash`, que reemplaza al original shell de Unix, llamado **sh**.

Más información en:

- How to Use Command Line

Emuladores de terminal

Cuando estamos usando una interfaz de usuario gráfica, necesitamos otro programa llamado `emulador de terminal (terminal emulador)` o simplemente **terminal** para interactuar con el shell.

Por ejemplo para interfaces como KDE Gnome

- KDE **konsole**.
- GNOME **gnome-terminal.**

Primeros pasos

Normalmente podemos lanzar el terminal escribiendo `alt + F2`

```
[nombre-usuario@linux~]$
```

Esto puede variar de la distribución y usualmente incluye `nombre-usuario@nombre-maquina`, seguido por el actual directorio de trabajo y un signo de dolar.

Algunas veces en lugar del signo dolar, aparece el signo `#`, esto significa que el terminal tiene `previlegios de super usuario`. Eso significa que tu estas logeado como el `usuario root` o has seleccionado un terminal con permisos de `superusuario`.

Comandos linux

La forma general de los comandos Linux

`<comando><opciones><argumentos>`

- `<comando>` es la acción que queremos que la computadora tome.
- `<opciones>` o *flag* modifica el compartamiento del comando.
- `<argumentos>` son las cosas que queremos sobre que el comando actúe.

Consejos

- Si hay espacios en los nombres de archivo o directorio, utilice el símbolo **“”** para anotar los caracteres de espacio, o simplemente coloque la ruta completa del archivo entre comillas.
- Después de escribir las primeras letras de un nombre de archivo o directorio, puede pulsar **Tab** para completar automáticamente el nombre.
- Utilice las teclas de flecha arriba y abajo para navegar por los comandos previamente escritos.

Ruta de archivos

- Una **ruta de archivo relativa** especifica la ruta de acceso a un archivo, teniendo en cuenta su directorio de trabajo actual. Por ejemplo, si le dieras a alguien instrucciones *relativas* a su casa, le daría direcciones desde su ubicación actual (la ruta relativa es desde donde están ellos hasta donde estás tu).
- Una **ruta de archivo absoluta** especifica la ruta completa a un archivo, haciendo caso omiso del directorio de trabajo actual. Por ejemplo, si diera a alguien direcciones **absolutas** a su casa, empezaría diciéndoles que estuvieran en la tierra, luego vaya a su continente, luego vaya a su país, luego vaya a su región, etc.

Comandos básicos

- **pwd** prints working directory (muestra el directorio actual de trabajo).
- **ls** lista todos los archivos y directorios del actual directorio de trabajo.
- `ls -a`, lista todos los archivos, incluyendo los archivos ocultos.
- `ls -l`, lista todos los archivos en un formato con información extra (permisos, tamaño, fecha de modificación, etc.)
- `ls *`,lista los contenidos de los subdirectorios (un nivel de profundidad) en tu directorio de trabajo.
- `ls <ruta>`, lista los archivos en un directorio específico (sin cambiar tu directorio de trabajo)
- **clear** Borra toda la salida de la consola.

Comandos básicos(1)

- **cd**
- `cd <ruta>` cambia el directorio a la ruta que se especifique, que puede ser una ruta relativa o una ruta absoluta.
- `cd..` mueve hacia un directorio padre.
- `cd` se mueve a tu directorio *home*.
- **mv**
- `mv <nombrearchivo> <nueva ruta>` mueve un archivo desde la localización local a moves a `<nueva ruta>`.
- `mv <nombrearchivo> <nueva ruta >` cambia el nombre de un archivo sin cambiar su ubicación.

Comandos básicos(2)

- **mkdir**
- `mkdir <nombredirectorio>` crea un directorio llamado `<nombredirectorio>`.
- **rm -i**
- `rm <nombrearchivo>` remueve (elimina) un archivo permanentemente.
- `rm -i <nombrearchivo>` remueve archivos en modo interactivo.
- `rm -ir <nombredirectorio>` elimina un directorio y elimina recursivamente todo su contenido.

Comandos básicos(3)

- **cp**
- `cp <nombrearchivo><nueva ruta>` remueve (elimina) un archivo permanentemente.
- `cp <nombrearchivo><nuevoarchivo` remueve archivos en modo interactivo.
- **touch**
- `touch <nombrearchivo>` crea un archivo vacío llamado `<nombrearchivo`.
- Esto es útil para crear archivos vacíos que se editarán posteriormente..
- Puede crear varios archivos vacíos con un solo comando: `touch <nombrearchivo1> <nombrearchivo2>`...

Comandos intermedios

- **head**
- `head <nombrearchivo>`, imprime el encabezado (las primeras 10 lineas) de un archivo.
- `head -n20 <nombrearchivo`, imprime las primeras 20 lineas de un archivo.
- Esto es útil para previsualizar el contenido de un archivo sin abrirlo.
- **tail**
- `tail <nombrearchivo>` imprime las últimas 10 líneas de un archivo .

Comandos intermedios(1)

- **cat**
- `cat <nombrearchivo>`, imprime un archivo.
- **less**
- `less <nombrearchivo>` permite desplazarse por el archivo .
- Pulse la barra de espacio para ir hacia abajo de una página, use las teclas de flecha para desplazarse hacia arriba y hacia abajo y pulse *q* para salir.
- |
- `<comando1>|<comando2>` canaliza los resultados de `<comando 1>` en `<comando 2>` y a continuación, los resultados del `<comando 2>` se imprimen en la consola.

Comandos intermedios(2)

- **wc**
- `wc <nombrearchivo>` devuelve el conteo de líneas, palabras y caracteres en un archivo .
- `wc -l <nombrearchivo`, solo cuenta líneas.
- `wc -w <nombrearchivo` solo cuenta las palabras.
- `wc -c <nombrearchivo>` solo cuenta caracteres.
- Una “palabra” se define como cualquier conjunto de caracteres delimitados por un espacio.
- >

- `<comando> > <nombrearchivo>` toma la salida de y la guarda en `<nombrearchivo>`. Esto sobrescribirá el archivo si ya existe.

Comandos intermedios(3)

- **grep**
- `grep <patron><nombrearchivo>` busca en un archivo un patrón de expresión regular e imprime las líneas coincidentes que debe estar entre comillas para permitir varias palabras. El patrón es sensible a mayúsculas y minúsculas por defecto, pero puede usar la opción **-i** para ignorar el caso.
- `grep -r <patron> <ruta>` hace una búsqueda recursiva de la ruta (verifica los subdirectorios) para encontrar coincidencias dentro de los archivos
- `grep <patron>` realiza una búsqueda global (de todo el equipo) por coincidencias. Pulsa `ctrl + c` si se desea cancelar la búsqueda.
- Pueden usarse patrones mucho más complejos de adaptación de cadenas.

Comandos intermedios(4)

- **find**
- `find <ruta> -name <nombre>` buscará recursivamente la ruta de acceso especificada (y sus subdirectorios) y encontrará archivos y directorios con un `<nombre>`. Usa `.` para especificar el directorio de trabajo.
- Para `<nombre>`, puede buscar una coincidencia exacta o utilizar caracteres comodín para buscar una coincidencia parcial, usando `*` y `?`.
- >>
- `<comando> >> <nombrearchivo>` toma la salida de `<comando>` y lo agrega a `<nombrearchivo>`. Esto creará un archivo si aún no existe.

Comodines

Desde que el shell utiliza nombres de archivo demasiadas veces, proporciona caracteres especiales para ayudar a especificar rápidamente grupos de nombres de archivos. Estos caracteres especiales son llamados comodines.

Los comodines nos permiten seleccionar los nombres de archivo basado en patrones de caracteres. Los principales comodines se muestran a continuación:

- El comodín `*` enlaza uno o más caracteres.
- El comodin `?` enlaza un único caracter.
- `[caracteres]` enlaza algún caracter que es miembro de *caracteres*.

Comodines(1)

- [[:clase:]] enlaza algún caracter que es miembro de una determinada clase. Las clases comunmente usadas son:
- `[:alnum:]` enlaza algún caracter alfa numérico.
- `[:alpha:]` enlaza algún caracter alfabético.
- `[:digit:]` enlaza algún numeral.
- `[:lower:]` enlaza alguna letra minúscula.

- `[:upper:]` enlaza alguna letra mayúscula.

Lecturas

- Cuaderno de notas de comandos Linux para R.
- Unix as IDE: Introduction.
- Unix Power Tool.

¿Por qué aprender un control de versiones?

- Es útil tener un sistema formal de seguimiento de diversas versiones de trabajo.
- Especialmente útil cuando escribe código.
- Permite a los equipos colaborar fácilmente en la misma base de código.
- Permite contribuir a proyectos de código abierto.
- Es una habilidad atractiva para el empleo.

¿ Qué es Git?

- Sistema de control de versiones que le permite realizar un seguimiento de archivos y cambios de archivos en un repositorio (**repo**).
- Principalmente utilizado por los desarrolladores de software.
- Sistema de control de versiones ampliamente utilizado con respecto a otros como **Mercurial**, **Subversion**, **CVS**.
- Se ejecuta desde la línea de comandos (normalmente).
- Se puede usar solo o en equipo.

¿ Qué es Github?

- Es un sitio web, no un sistema de control de versiones.
- Permite color tus repositorios Git línea.
- El servidor de código más grande del mundo.
- Alternativa a Bitbucket.
- Beneficios de GitHub:
- Copia de seguridad de los archivos.
- Interfaz visual para navegar en los repositorios.
- Facilita la colaboración de repositorios.
- GitHub es sólo el Dropbox para Git.
- Git no requiere de GitHub

Git puede ser un reto aprender

- Diseñado (por programadores) para la potencia y la flexibilidad sobre simplicidad.
- Es difícil saber si lo que hiciste fue correcto.
- Difícil de explorar ya que la mayoría de las acciones son **permanentes** (en cierto sentido) y pueden tener serias consecuencias
- Nos enfocaremos en el 10% más importante de Git.

Más información:

- GitRef.
- Mastering Markdown.

Navegando un repositorio de Github(1)

- Ejemplo de repositorio: https://github.com/C-Lara/Curso-R.
- Nombre de cuenta, nombre del repositorio, descripción.
- Estructura de las carpetas.
- Visualización de archivos:
- Vista renderizada (con resaltado de sintaxis).
- Vista sin formato (raw).
- README.md:
- Describe un repositorio.
- Se muestra automáticamente.
- Escrito en Markdown.

Navegando un repositorio de Github(2)

- Commits:
- Uno o más cambios en uno o más archivos.
- Revisión con resaltadps
- Se requieren comentarios para ser commit, puede ser útil la opción **commit -m** .
- Comentario de confirmación más reciente mostrado en el archivo.
- Página de perfil : https://github.com/settings/profile.

Creando un repositorio en Github

- Haga clic en **New repository** :
- Definimos el nombre, descripción, público o privado.
- Inicializa con **README** (si va a clonar).
- Notas:
- No ha sucedido nada a en tu computadora local
- Esto se hizo en GitHub, pero GitHub utilizó Git para agregar el archivo **README.md**.

Markdown básico

- Lenguaje de marcado fácil de leer y fácil de escribir. .
- Markdown permite editar el archivo **README.md** usando GitHub.
- Código HTML válido también se puede utilizar en Markdown.
- Sintaxis común:

- `##` tamaño del encabezado 2 .
- `*Cursivas*` y `**negrita**`
- `[Enlace a gitHub](https://github.com)`
- * indica una viñeta.

Más información: Mastering Markdown.

Avance de lo que se va hacer

- Copia tu nuevo repositorio GitHub en tu computadora.
- Realice algunos cambios de archivos localmente.
- Guardar los cambios localmente (hacer **commit**).
- Actualiza tu repositorio de GitHub con esos cambios.

Clonando un repositorio de GitHub

- Clonar = copia a una computadora local.
- Es como copiar tus archivos de dropbox a una nueva máquina.
- En primer lugar, cambie el directorio de trabajo en lugar donde se almacene localmente el repositorio: **cd**
- Entonces, clone el repositorio: `git clone <URL>`.
- Consigue el URL desde GitHub (termina en .git).
- Clona en un subdirectorio del directorio de trabajo.
- No hay comentarios visuales cuando escribes tu contraseña.
- Navegamos hasta el repositorio con **cd**, luego lista los archivos **ls**.

Comprobación de repositorios remotos

- **remoto** es una referencia a un repositorio que no se encuentra en una computadora local.
- Como una conexión a una cuenta de dropbox.
- Vemos los repositorios remotos: **git remote -v**.
- El repositorio remoto **origen** fue configurado por **git clone**.
- Nota: Los repositorios remotos son repositorios específicos.

Realizar cambios, comprobar el estado del repositorio

- Haciendo cambios:
- Modificar el **README.md** en cualquier editor de texto.
- Crear un nuevo archivo: `touch <nombrearchivo>`
- Comprueba tu estado: **git status**.
- Estado de los archivos (posiblemente con código de colores):
- Sin seguimiento (rojo).
- Bajo seguimiento y modificado (rojo).
- Preparado para confirmar (verde).
- Confirmado.

Preparar y confirmar cambios

- Preparar cambios para confirmar:
- Añadir un solo archivo: **git add** .
- Añadir todos los archivos *rojos*: **git add -A**.
- Compruebe tu estado:
- Los archivos rojos se han vuelto verdes.
- Confirmar cambios:
- **git commit -m "mensaje para confirmar"**
- ¡Vuelve a comprobar tu estado!.
- Comprueba el registro: **git log**.

Enviando cambios a Github

- Todo lo que has hecho al repositorio clonado (hasta ahora) ha sido local.
- Se ha estado trabajando en la rama principal **master**.
- Envie los cambios confirmados a GitHub:
- Es como sincronizar los cambios de archivos locales en Dropbox.
- **git push** [**nombre-remoto**][**nombre-rama**].
- A menudo: **git push origen master**.
- ¡Actualice tu repositorio GitHub para comprobarlo!.

Cambios realizados de manera remota

- Los cambios de los repositorios se han realizado en una máquina local y luego se han enviado a GitHub.
- ¿Qué sucede si clona el repositorio de GitHub de otras personas y luego estos realizan cambios en él?.
- Git no actualiza automáticamente un repositorio local con cambios remotos.

Recuperando cambios de GitHub

• Git permite recuperar manualmente los cambios de localizaciones remotas. - Es como realizar la sincronización de tus archivos locales desde Dropbox. - **git pull** [**nombre-remoto**][**nombre-rama**]. - A menudo: **git pull origen master**.

Diagrama de flujo de Github

¿Cuándo es necesario realizar git pull ?

- Realizar un **git pull** sólo es necesario cuando se han realizado cambios remotamente pero no localmente.
- El escenario más común: el repositorio es propiedad de otra persona.
- También es común que se realicen cambios en el mismo repositorio desde varios equipos.
- Es un buen hábito realizar un **git pull** antes de empezar a trabajar-
- Ningún daño se produce a un repositorio que no ha cambiado.

Conflictos de fusión

- El problema más común cuando se recuperan datos desde github es un **conflicto de fusión**: existe un conflicto entre los cambios que se están fusionando y los cambios que se han realizado localmente.
- Cómo evitar conflictos de fusion:

- Si desea editar archivos de algún repositorio, realiza copias y edite las copias en su lugar. ? Cómo resolver un conflicto de combinación:
- Descarta tus cambios: `git checkout --<nombrearchivo>`.
- Luego intente realizar un **pull**.

Eliminar o mover un repositorio

- Eliminación de un repositorio en GitHub:
- **Setting** y a continuación **Delete this repository**.
- Eliminación de un repositorio local:
- ¡Borrar la carpeta!
- Mover un repositorio local:
- ¡Sólo mueve la carpeta del repositorio!

Gists: repositorios ligeros

- Se tiene acceso a **Gist**: https://gist.github.com/.
- Se puede incluir uno o más archivos.
- Útil para snippets, presentaciones de tareas.
- Pueden ser públicos o secretos (no privados).
- Soporta edición, clonación, comfirmaciones, comentarios, etc.
- ¡Creemos uno ahora mismo!.

Excluir archivos de un repositorio

- Crea un archivo **.gitignore** en un repo: `touch.gitignore`.
- Especificar exclusiones, una por línea:
- Archivos individuales: `pip-log.txt`.
- Todos los archivos con la extensión correspondiente: `*.pyc`.
- Directorios: `env /`.
- Ejemplos: https://github.com/github/gitignore

Dos formas de inicializar Git

- Inicializando en GitHub:
- Crear un repositorio en GitHub (con un README).
- Clona el repositorio en una máquina local.
- Esto es lo que se hizo en estas notas (recomendado).
- Inicializando localmente:
- Inicializar Git en un directorio local existente: **git init**.
- Crear un repositorio en GitHub (sin un **README**).
- Añadir un repositorio remoto: `git remote add origen <URL>`.

Saliendo de Vim

- Siempre se incluye un mensaje como confirmación: `git commit -m "mensaje"`
- Se le llevará a un editor de texto si intenta hacer una confirmación sin un mensaje: **git commit**.
- El editor de texto por defecto suele ser Vim.
- Cómo salir de Vim:
- Pulse la tecla **Esc** (para entrar en el modo de comando).

- Escribe **:q!** (para salir sin guardar) y **:wq** para guardar y salir
- Pulse la tecla **Enter.

¿Qué es R?

- R es un lenguaje y un entorno de computación y gráficos estadísticos.
- La página del proyecto es: https://www.r-project.org/.
- El entorno de por si, cuenta con un gestor de búsqueda : http://www.rseek.org/.
- R ofrece una amplia variedad de técnicas estadísticas importantes como la regresión lineal, análisis de series temporales, booststrap, estimación de máxima verosimilitud, etc.

Características de R

- Como lenguaje de programación incluye estructuras de programación como :condicionales, bucles, funciones recursivas definidas por el usuario, etc. La habilidad para combinar funciones de R, produce una óptima flexibilidad y además resulta de ser una técnica muy útil. Por ejemplo

```
f <- function (x) {
   c(x=x, floor=floor(x), ceiling=ceiling(x), round=round(x,2), signif=signif(x,2))
 }
 t(apply(t(rt(10,4)),2,f))
```

========== - R, puede realizar tareas computacionalmente intensivas, vinculando lenguajes como C, C++ o Fortran en tiempo de ejecución y mejorar el proceso computacional.

- Posee facilidades gráficas para el análisis y visualización de datos ya sea en pantalla o en copia impresa, como lo realiza la libreria **ggplot2** de Hadley Wickham.
- R se maneja por *paquetes*, existe un repositorio oficial de paquetes , organizados por temas:

http://www.cran.r-project.org/web/views/.

========== - Varios editores están disponibles para trabajar con R:

1. [**RStudio**] (http://www.rstudio.com/ide/).
2. [**Eclipse**] (http://www.walware.de/goto/statet).
3. [**Vim-R-Tmux**] (http://manuals.bioinformatics.ucr. edu/home/programming-in-r/vim-r).

- R proporciona una visión general del aprendizaje estadístico, posee un conjunto de herramientas esenciales para dar sentido y resolución a los grandes y complejos conjuntos de datos que han surgido en campos que van desde la biología, las finanzas. el marketing , la astrofísica en los pasados 20 años.

Programación Orientada a Objetos en R

- R hereda de **S**, la programación orientada a objetos, que puede ser útil en problemas de Regresión Lineal.
- R posee **polimorfismo**, lo que significa que una única función puede ser aplicada a diferentes tipos de entradas, procesando la función de una manera apropiada en cada caso (*función genérica*). Ejemplo de esto es la función *plot()*.
- Se maneja este paradigma mediantes las clases **clases S3** y **clases S4**.

```
➜  ~ R

R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R es un software libre y viene sin GARANTIA ALGUNA.
Usted puede redistribuirlo bajo ciertas circunstancias.
Escriba 'license()' o 'licence()' para detalles de distribucion.

R es un proyecto colaborativo con muchos contribuyentes.
Escriba 'contributors()' para obtener más información y
'citation()' para saber cómo citar R o paquetes de R en publicaciones.

Escriba 'demo()' para demostraciones, 'help()' para el sistema on-line de ayuda,
o 'help.start()' para abrir el sistema de ayuda HTML con su navegador.
Escriba 'q()' para salir de R.

[Previously saved workspace restored]

>
```

Figure 1: salida

```
library(pryr)

df <- data.frame(x = 1:10, y = letters[1:10])
otype(df)
```

Instalación de R

La información de como instalar correctamente R, en un entorno de Ubuntu, se encuentra en la página:

How To Set Up R on Ubuntu 14.04.

Si todo ha salido bien, se puede correr el comando **R** y ver lo siguiente:

Instalación de Rstudio

RStudio es un IDE para R. Es software libre con licencia GPLv3 y se puede ejecutar sobre distintas plataformas o incluso desde la web usando RStudio Server.

```
c-lara@Lara:~$ wget https://download1.rstudio.org/rstudio-0.99.893-amd64.deb
c-lara@Lara:~$sudo dpkg -i *.deb
c-lara@Lara:~$rm *.deb
```

Para ver como funciona RStudio escribiendo código se puede revisar: - Programming Part 1 (Writing code in RStudio).

R Markdown

R Markdown es un framework para ciencia para crear reportes dinámicos con R, además de ejecutar y guardar código.

R Markdown soporta formatos de salida estáticos y dinámicos que incluye hTML, pdf, beamer-latex, html5T, shiny,etc.

Más información:

- Lecciones de R Markdown.

R Presentations

R presentations son una característica de RStudio que permiten la creación fácil de presentaciones HTML5 utilizando una combinación de Markdown y R.

El objetivo de R Presentations es crear diapositivas que hagan uso de código R y de ecuaciones de LaTeX tan sencillas como sea posible.

Más información:

- Introduction to Presentations in Rmarkdown.

Knitr

knitr fue diseñado para ser una máquina de generación dinámica de reportes o documentos que son una mezcla de texto y código que se procesa y devuelve respuestas válidas para la ciencia de datos. El diseño de knitr permite no permite sólo código R, sino de otros lenguajes como Python, Java Script o Awk, además de producir resultados en formatos como LaTeX, HTML5, Markdown, AsciiDoc, etc, como se muestran en los ejemplos de knitr.

El paquete **Knitr** se instala en R,

```
install.packages("knitr")
library("knitr")
```

Knitr es libre, además de poseer muchos ejemplos y demostraciones.

Ayuda en R

Escribamos una declaración en R,

```
mean(1:10)
```

Cuando empezamos a escribir código R, es importante saber cómo obtener ayuda y R proporciona muchas formas de hacer esto. Partiendo de que todo es un objeto, podemos hacer lo siguiente:

```
?mean
?"+"
?"if"
??plotting
??"regression model"
```

Ayuda en R

Las funciones **help** y **help.search** hacen las mismas cosas que **?** y **??**, respectivamente, pero se necesita incluir argumentos entre comillas. Los siguientes comandos son equivalentes al código anterior:

```
help("mean")
help("+")
help("if")
help.search("plotting")
help.search("regression model")
```

Ayuda en R

La función **apropos** encuentra variables (incluyendo funciones) que coinciden con su entrada. Esto es realmente útil si sólo puede recordar a medias el nombre de una variable que ha creado o una función que desea utilizar. Por ejemplo, supongamos que creamos una variable *a_vector*:

```
a_vector <- c(1, 3, 6, 10, 15)
apropos("vector")
```

```
[1] "a_vector"             "as.data.frame.vector" "as.vector"
[4] "as.vector.factor"     "is.vector"            "vector"
[7] "Vectorize"
```

Ayuda en R

Encontrar las variables que contienen una cadena en particular está bien, pero también se puede hacer una comparación con **apropos** utilizando expresiones regulares. **Las expresiones regulares** son una sintaxis en varios lenguajes que trata de la coincidencia en las cadenas. Se necesita aprender a usarlos. **Cambiarán tu vida**. Un simple uso de apropos por ejemplo podría ser encontrar todas las variables que terminan en z o encontrar todas las variables que contengan un número entre 4 y 9:

```
apropos("z$")
apropos("[4-9]")
```

Mayor información de expresiones regulares aquí.

Ayuda en R

La mayoría de las funciones tienen ejemplos que puedes ejecutar para tener una mejor idea de cómo funcionan. Se utiliza la función **example** para ejecutarlos. También hay demostraciones de conceptos que son accesibles con la función **demo**:

```
example("plot")
demo(Japanese)
```

R es modular y se divide en paquetes, algunos de los cuales contienen vignettes, que son documentos cortos sobre cómo usar los paquetes. Se puede explorar todos los vignettes usando **browseVignettes**:

```
browseVignettes()
```

Ayuda en R

También puede acceder a un vignettes específico mediante la función **vignette** (pero si su memoria es tan mala como la mía, usar browseVignettes combinado con una búsqueda de página es más fácil que tratar de recordar el nombre de un vignette y en qué paquete está):

```
vignette("Sweave", package = "utils")
```

El operador de búsqueda de ayuda **??** y **browseVignettes** sólo encuentran paquetes que hayan sido instalados. Si se desea buscar cualquier paquete, puede utilizar **RSiteSearch**, que ejecuta una consulta en http://search.r-project.org. Los términos que tenga varias palabras necesitan ser escritos entre llaves.

```
RSiteSearch("{Bayesian regression}")
```

Algunos Recursos en Línea de R

- http://www.r-bloggers.com/. Todo lo referente al ecosistema R, esta aquí, gracias a la contribucion de más o menos 450 bloggers.
- **Stack Overflow**, Es un sitio para encontrar soluciones a problemas informáticos. http://stackoverflow.com/questions/tagged/r.
- http://zoonek2.free.fr/UNIX/48_R/02.html Uno de los mejores cursos sobre R. Hecho por Vincent Zoonekynd.
- Recursos de UCLA para aprender R: http://www.ats.ucla.edu/stat/r/. Operaciones matématicas y vectores ==========
- El operador + realiza la adición y se puede usar para añadir dos vectores. Un vector es un conjunto ordenado de valores
- El operador dos puntos **:** crea una secuencia desde un número al próximo y la función **c** concatena valores, en este caso crea vectores.
- Los nombres de variables son sensibles a mayúsculas y minúsculas en R, por lo que necesitamos tener un poco de cuidado. La función **C** hace algo completamente diferente a **c**.

```
1:5 + 6:10
c(1, 3, 6, 10, 15) + c(0, 1, 3, 6, 10)
```

Vectorización

Si estamos escribiendo en un lenguaje como C o Fortran, tendríamos que escribir un bucle para realizar la adición de los elementos a los vectores. La naturaleza vectorizada de la adición en R hace las cosas fáciles, evitando los bucle .

La vectorización tiene varios significados en R, el más común de los cuales es que un operador o una función actuará sobre cada elemento de un vector sin la necesidad de que podamos escribir un bucle.

```
sum(1:5)
```

```
[1] 15
```

Vectorización(1)

Un segundo significado de la vectorización es cuando una función toma un vector como entrada y calcula un resumen estadístico.

```
sd(1:5)
```

```
[1] 1.581139
```

Un tercer caso mucho menos común sobre la vectorización es la vectorización sobre argumentos. Esto es cuando una función calcula un resumen estadístico de varios de sus argumentos de entrada. La función **median** devuelve un resultado diferente o error por ejemplo:

```
median(1, 2, 3, 4, 5)
```

```
[1] 1
```

Vectorización(2)

Todos los operadores aritméticos en R, no sólo más (+) son vectorizados.

```
c(2, 3, 5, 7, 11, 13) - 2
```

```
[1]  0  1  3  5  9 11
```

```
1:2 ** 3
```

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

```
1:10 %/% 3
```

```
 [1] 0 0 1 1 1 2 2 2 3 3
```

```
1:10 %% 3
```

```
 [1] 1 2 0 1 2 0 1 2 0 1
```

Funciones matématicas

left: 60% R también contiene una amplia selección de funciones matemáticas, **sin**, **cos**, **tan** y sus inversas, logaritmos y exponentes, **log1p** y **expm1** que calculan **log(1 + x)** y exp (x - 1)** con más precisión para valores muy pequeños, por ejemplo:

```
cos(c(0, pi/5, pi/3, pi))
```

```
[1]  1.000000  0.809017  0.500000 -1.000000
```

```
exp(pi * 1i) + 1
```

```
[1] 0+1.224647e-16i
```

```
factorial(7)
```

```
[1] 5040
```

Operadores relacionales

left: 60% Para comparar valores enteros por igualdad, se utiliza ==. Los otros operadores relacionales son vectorizados. Para comprobar la desigualdad, el operador no es igual **!=**. Los operadores mayor y menor que son como se puede esperar: > y < (o > = y <= si se permite la igualdad).

```
c(3, 4 - 1, 1 + 1 + 1) == 3
```

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

```
1:3 != 3:1
```

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

```
(1:5) ^ 2 >= 16
```

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

Comparación de no enteros

La comparación de no enteros usando == es problemática. Todos los números que hemos tratado hasta ahora son números de coma flotante. Esto significa que se almacenan en la forma **a*2^ b**, para dos números a y b. Puesto que esta forma entera tiene que ser almacenada en 32 bits, el número resultante es solamente una aproximación, lo que las respuestas que se esperaba pueden estar equivocadas.

```
sqrt(2) ^ 2 == 2
```

```
[1] FALSE
```

```
sqrt(2) ^ 2 - 2
```

```
[1] 4.440892e-16
```

Comparación de no enteros(1)

R proporciona la función **all.equal** para comprobar la igualdad de números. Esto proporciona un nivel de tolerancia (por defecto, aproximadamente de 1.5e-8), de modo que se ignoran los errores de redondeo inferiores a esa tolerancia:

```
all.equal(sqrt(2) ^ 2, 2)
```

```
[1] TRUE
```

Si los valores a comparar no son los mismos, all.equal devuelve un informe sobre las diferencias.

Asignación de variables

La mayoría de veces queremos almacenar los resultados para su reutilización. Podemos asignar una variable (local) usando <- or =, aunque por razones históricas, se prefiere <- :

```
x <- 1:5
y = 6:10

# Podemos reutilizar esos valores
x + 2 * y - 3
```

También podemos hacer una asignación global usando <<-. Por ahora tenemos una variable disponible en cualquier lugar:

```
y <<- exp(exp(1))
```

Números especiales

R admite cuatro valores numéricos especiales: **Inf**, **-Inf**, **NaN** y **NA**.

NaN es la abreviatura de **not a number** y significa que nuestro cálculo no tiene sentido matemático o no se puede realizar correctamente. **NA** es la abreviatura de **not available** y representa un valor perdido, un problema común en análisis de datos.

En general, si nuestro cálculo implica un valor **NA**, en los resultados aparecerá un **NA**:

```
c(NA + 1, NA * 5, NA + Inf)
```

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

Números especiales(1)

Cuando la aritmética implica **NA** y **NaN** la respuesta es uno de esos dos valores, pero cuál de esos dos es, depende del sistema:

```
c(NA + NA, NaN + NaN, NaN + NA, NA + NaN)
```

```
[1]  NA NaN NaN  NA
```

```
c(sqrt(Inf), sin(Inf))
```

```
[1] Inf NaN
```

Hay funciones disponibles para comprobar estos valores especiales. Se debe observar que **NaN** y **NA** no son ni finitos ni infinitos, y **NA** por ejemplo es un número:

Números especiales(2)

Las funciones **is.infinite**, **is.nan** y **is.na** verifican si una elemento es de los tipos mencionados:

```
x <- c(0, Inf, -Inf, NaN, NA)
is.finite(x)
```

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

```
is.infinite(x)
```

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

```
is.nan(x)
```

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

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

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

Operadores lógicos

Muchos lenguajes de programación utilizan lógica booleana, donde los valores pueden ser **TRUE** o **FALSE**. En R, la situación es un poco más complicada, ya que también podemos tener valores faltantes **NA**. Hay tres operadores lógicos vectorizados en R:

- **!** es usado para **not**.
- **&** es usado para **and**.
- **|** es usado para **or**

Dos funciones útiles para tratar con vectores lógicos son **any** y **all** los cuales devuelven **TRUE** si el vector de entrada contiene al menos un valor **TRUE** o todos los valores son **TRUE**, respectivamente:

Operadores lógicos(1)

Podemos escribir algunas tablas de verdad, con el siguiente código:

```
x <- c(TRUE, FALSE, NA)
xy <- expand.grid(x = x, y = x) # todas las combinaciones de x e y
within(
  xy,
```

```
  {
    and <- x & y
    or <- x | y
    not.y <- !y
    not.x <- !x
      }
  )
```

Estructuras de datos básicas de R

- Existen ciertas estructuras 1d y 2d dimensionales , para almacenar distintos objetos de R

Vectores

Un vector numérico es una lista de números. La función *c()* se utiliza para coleccionar cosas en un vector. Podemos asignar esto a un objeto con nombre:

```
x <- c(0, 7, 8)
x
```

- El operador **:** es usado para crear secuencias de valores crecientes (decrecientes).

```
numb5a20 <- 5:20
numb5a20
```

```
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Concatenación de vectores

Los vectores pueden ser **concatenados** con la función *c*.

```
c(numb5a20, x)
```

- Un ejemplo de la función *c()*

```
algunos.numeros <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 59, 67, 71, 73, 79, 83, 89
```

Podemos agregar *numb5a20* al final de *algunos.numeros* y entonces agregar la secuencia decreciente desde 4 al 1:

Un ejemplo

```
a.concatenacion <- c(algunos.numeros, numb5a20, 4:1)
a.concatenacion
```

```
 [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  59  67
[18]  71  73  79  83  89  97 103 107 109 113 119   5   6   7   8   9  10
[35]  11  12  13  14  15  16  17  18  19  20   4   3   2   1
```

- Los números en los corchetes dan el índice de los elementos a la derecha.

Extrayendo elementos de un vector

- Una forma de mostrar el elemento 22 de *a.concatenacion* es utilizar corchetes para extraer un sólo elemento:

```
a.concatenacion[22]
```

```
[1] 89
```

- Podemos extraer más de un elemento a la vez.

```
algunos.numeros[c(3, 6, 7)]
```

```
[1]  5 13 17
```

Usando el operador : e índices negativos

- Para obtener el tercero al séptimo elemento de *numb5a20*, escribe

```
numb5a20[3:7]
```

```
[1]  7  8  9 10 11
```

- Los índices negativos se pueden utilizar para evitar ciertos elementos. Por ejemplo, podemos seleccionar todo excepto el segundo elemento de x como sigue

```
x[-2]
```

Extracción de elementos de un vector avanzada

- Utilizar un índice cero no devuelve nada. Esto no es algo que normalmente se escribe, pero puede ser útil en expresiones más complicadas.

```
numb5a20[c(0, 3:7)]
```

```
[1]  7  8  9 10 11
```

- No se puede mezclar índices positivos y negativos. Para ver qué pasa, consideramos

```
x <- c(0,  7, 8)
x[c(-2, 3)]
```

Aritmética de vectores

Las operaciones básicas de la arítmética se pueden hacer con vectores de R

```
x<- c(0,7,8)
x - 4
```

```
[1] -4  3  4
```

```
x*3
```

```
[1]  0 21 24
```

Además

```
x^3
```

```
[1]   0 343 512
```

Las operaciones son elemento a elemento.

Arítmética entre dos vectores

Las operaciones binarias trabajan elemento a elemento también cuando son aplicadas a un par de vectores.

Por ejemplo, podemos calcular $y_i^{x_i}$, para $i = 1, 2, 3$, como sigue:

```
x<- c(0,4,6)
y <- -x - 3
y
```

```
[1] -3 -7 -9
```

---

además

```
y^x
```

```
[1]      1   2401 531441
```

Reciclado

Cuando los vectores tienen diferentes longitudes, la longitud más corta se extiende por **reciclado**: los valores se repiten, comenzando desde el principio.

Por ejemplo, para ver el patrón de restos de los números del *1* al *10* módulo *2* y *3*, sólo necesitamos dar el vector *2: 3* una vez:

```
c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5,6, 6, 7, 7, 8, 8, 9, 9, 10, 10) %% 2:3
```

```
 [1] 1 1 0 2 1 0 0 1 1 2 0 0 1 1 0 2 1 0 0 1
```

Cuando el reciclado muestra advertencias

R dará una advertencia si la longitud del vector más grande no es un múltiplo de la longitud del vector más pequeño.

Por ejemplo, si queríamos los restos modulo *2*, *3* y *4*, esta es la manera incorrecta de hacerlo:

```
c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10) %% 2:4
```

```
Warning in c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10)%
%2:4: longer object length is not a multiple of shorter object length
```

```
 [1] 1 1 2 0 0 3 0 1 1 1 0 2 1 1 0 0 0 1 0 1
```

Patrones simples de vectores

La construcción de patrón de vectores se pueden hacer usando las funciones **seq()**, así como la función **rep()**.

Por ejemplo, la secuencia de números impares menores o iguales a 21, pueden ser obtenido usando

```
help(seq)
seq(1, 21, by =2)
```

Debes notar que aquí usamos **by =2**.

La función **seq()** tiene varios paramétros opcionales, incluyendo *by*. Si *by* no es especificado, el valor por defecto de 1 puede ser usado.

Patrones repetidos

Si queremos encontrar patrones repetidos, podemos utilizar **rep()**.

```
rep(3, 12)
```

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

Esta función puede tener más opciones. Por ejemplo que resulta de escribir?

```
rep(seq(2,20, by = 2), 2)
```

Más ejemplos

Las funciones anteriores se pueden combinar

```
rep(c(1, 4), c(3, 2))
```

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

```
rep(c(1, 4), each=3)
```

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

```
rep(seq(2, 20, 2), rep(2, 10))
```

```
 [1]  2  2  4  4  6  6  8  8 10 10 12 12 14 14 16 16 18 18 20 20
```

Revisar la ayuda de la función puede ser útil.

Valores pérdidos: NA

El símbolo de valores pérdidos es **NA**. Valores pérdidos a menudo surgen en problemas reales, pero también pueden surgir debido a la forma en que se realizan los cálculos.

```
algunos.pares <- NULL # creamos un vector sin elementos

algunos.pares[seq(2, 20, 2)] <- seq(2, 20, 2)
algunos.pares
```

```
 [1] NA  2 NA  4 NA  6 NA  8 NA 10 NA 12 NA 14 NA 16 NA 18 NA 20
```

Otros valores especiales

Lo que sucedió aquí es que asignamos valores a los elementos *2, 4,. . . , 20* pero nunca se asignó nada a los elementos *1, 3,. . . , 19*, por lo que R usa **NA** para indicar que el valor es desconocido.

Si tenemos el vector *x <- c(0,7, 8)*. Consideramos

```
x <- c(0, 7, 8)
x/x
```

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

El símbolo **NaN** indica un valor que es *no un número*, que surge como resultado de intentar calcular el indeterminado 0/0. Este símbolo se utiliza a veces cuando un cálculo no tiene sentido.

Sobre los índices vectoriales

En otros casos, se pueden mostrar valores especiales o puede recibir un mensaje de error o de advertencia:

```
1/x
```

```
[1]       Inf 0.1428571 0.1250000
```

Los índices vectoriales sean enteros. Cuando se usan valores fraccionarios, se truncarán hacia 0. Así, 0.4 se convierte en 0, por ejemplo

```
x[0.4] # vector de longitud de cero
```

```
numeric(0)
```

Matrices y arrays

Para ordenar los valores en una matriz, usamos la función **matrix()**:

```
m <- matrix(1:6, nrow=2, ncol=3)
m
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Indexado de matrices

Entonces podemos acceder a los elementos usando dos índices. Por ejemplo, el valor en la primera fila, la segunda columna es

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

R también permite que una matriz sea indexada como un vector, usando sólo un valor:

```
m[4]
```

Aquí los elementos se seleccionan en el orden en que se almacenan internamente: en la primera columna, luego en la segunda y así sucesivamente.

Formas de almacenamiento de los elementos

Esto se conoce como orden de almacenamiento de **columna principal**. Algunos lenguajes usan el orden de almacenamiento de la **fila principal**, donde los valores se almacenan en orden de izquierda a derecha en la primera fila, luego de izquierda a derecha sobre la segunda, y así sucesivamente.

Las filas o columnas enteras de matrices se pueden seleccionar dejando en blanco el índice correspondiente:

```
m[1,]
m[,1]
```

Ejemplos

Podemos controlar como R completa los datos usando el argumento **byrow**,

```
matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,
       byrow=FALSE)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Podemos repetir la misma línea de código colocando **byrow=TRUE** .

Continuación. . .

```
matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,
       byrow=TRUE)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Enlazando filas

Si tiene varios vectores de igual longitud, podemos construir una matriz enlazando estos vectores utilizando las funciones, **rbind** y **cbind**.

Puedes tratar cada vector como una fila (mediante el comando **rbind**)

```
rbind(1:3,4:6)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Enlazando columnas

Podemos tratar cada vector como una columna (utilizando el comando **cbind**).

La misma matriz del ejemplo se puede escribir

```
cbind(c(1,4),c(2,5),c(3,6))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Dimensiones de una matriz

Otra función útil, **dim**, proporciona las dimensiones de una matriz almacenada en su espacio de trabajo.

```
mymat <- rbind(c(1,3,4),5:3,c(100,20,90),11:13)
dim(mymat)
```

```
[1] 4 3
```

```
nrow(mymat)
```

```
[1] 4
```

Otras funciones

Utilizamos las funciones : **nrow** (que proporciona el número de filas) y **ncol** (que proporciona el número de columnas).

En el último comando mostrado, usamos **dim** y el conocimiento de como extraer un elemento de un vector para extraer el mismo resultado que **ncol** daría.

```
ncol(mymat)
```

```
[1] 3
```

```
dim(mymat)[2]
```

```
[1] 3
```

Extrayendo filas

Sea la siguiente matriz

```
A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9)
            ,nrow=3,ncol=3)
A[2:3, ]
```

```
     [,1]  [,2] [,3]
[1,]  4.5   0.1  8.2
[2,] 55.3 105.5 27.9
```

Este comando retorna la segunda y tercera filas de **A**.

Extrayendo columnas

El siguiente comando retorna la tercera y primera columna de **A**

```
A[,c(3,1)]
```

```
     [,1] [,2]
[1,] -4.2  0.3
[2,]  8.2  4.5
[3,] 27.9 55.3
```

Para casos un poco más complejos, el siguiente comando accede a la tercera y primera filas de A, en ese orden, y de esas filas devuelve los elementos de columna segundo y tercero.

```
A[c(3,1),2:3]
```

Extrayendo elementos de la diagonal

También puede identificar los valores a lo largo de la diagonal de una matriz cuadrada (es decir, una matriz con un número igual de filas y columnas) usando el comando **diag**.

```
diag(x=A)
```

```
[1]  0.3  0.1 27.9
```

Omitir elementos de una matriz

Para eliminar u omitir elementos de una matriz, se vuelven a usar corchetes, pero esta vez con índices negativos. En el siguiente código eliminamos la primera fila de **A** y recuperamos la tercera y segunda columna en ese orden.

```
A[-1,3:2]
```

```
     [,1]  [,2]
[1,]  8.2   0.1
[2,] 27.9 105.5
```

Ejemplos

El siguiente ejemplo produce **A** sin la primera fila y segunda columna:

```
A[-1,-2]
```

Este ejemplo elimina la primera fila y luego elimina la segunda y tercera columnas del resultado:

```
A[-1,-c(2,3)]
```

Sobreescribiendo elementos de una matriz

Para sobrescribir elementos particulares, o filas o columnas enteras, debes identificar los elementos que se van a reemplazar y luego asignar los nuevos valores.

Los nuevos elementos pueden ser un solo valor, un vector de la misma longitud que el número de elementos a sustituir, o un vector cuya longitud divida uniformemente el número de elementos a reemplazar.

Creamos una copia

```
B <- A
```

Ejemplos

Lo siguiente sobrescribe la segunda fila de **B** con la secuencia 1, 2 y 3:

```
B[2,] <- 1:3
B
```

```
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  1.0   2.0  3.0
[3,] 55.3 105.5 27.9
```

Lo siguiente sobrescribe los elementos de la segunda columna de la primera y tercera filas con 900:

```
B[c(1,3),2] <- 900
```

Utilizando el reciclado

Para probar el reciclaje de vectores de R, vamos a sobrescribir los elementos de la primera y tercera columna de las filas 1 y 3 (un total de cuatro elementos) con los dos valores -7 y 7.

```
B[c(1,3),c(1,3)] <- c(-7,7)
B
```

```
     [,1]  [,2] [,3]
[1,]   -7  91.0   -7
[2,]    1   2.0    3
[3,]    7 105.5    7
```

Operaciones con matrices

- Transpuesta de una matriz
- Matriz identidad
- Multiplicación de un escalar por una matriz
- Suma y resta de matrices
- Multiplicación matricial
- Inversión de una matriz

Transpuesta de una matriz

En R, la transpuesta de una matriz es encontrada usanda la función **t**.

```
A <- rbind(c(2,5,2),c(6,1,4))
t(A)
```

```
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
```

Si tu realizas *la traspuesta de la traspuesta* recuperamos la matriz original.

```
t(t(A))
```

Matriz identidad

Puedes crear una matriz identidad usando lo función **matrix**, pero hay una forma utilizando la función **diag**

```
A <- diag(x =3)
A
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Multiplicación de un escalar por una matriz

R realizará esta multiplicación componente a componente. La multiplicación escalar de una matriz se realiza utilizando el operador aritmético *.

```
A <- rbind(c(2,5,7),c(6,3,4))
a<- 3
a*A
```

```
     [,1] [,2] [,3]
[1,]    6   15   21
[2,]   18    9   12
```

Suma y resta de matrices

Puede sumar o restar dos matrices de igual tamaño con los símbolos estándar + y -.

```
A <- cbind(c(2,5,7),c(6,1,4))
B <- cbind(c(-2,3,9),c(8.1,8.2,-9.8))
A + B
```

```
     [,1] [,2]
[1,]    0 14.1
[2,]    8  9.2
[3,]   16 -5.8
```

Multiplicación de matrices

A diferencia de la suma, resta y la multiplicación escalar, la multiplicación matricial no es un simple cálculo componente a componente, y no se puede utilizar el operador *.

En su lugar, debe utilizar el operador de producto de matriz de R, escrito con símbolos de porcentaje como %*%.

```
A <- rbind(c(2,5,2),c(6,1,4))
B <- cbind(c(3,-1,1),c(-3,1,5))
A%*% B  # Las matrices son compatibles
```

```
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

Inversión de una matriz

Hay varios enfoques diferentes para obtener la inversión de una matriz, y estos cálculos pueden llegar a ser computacionalmente costosos a medida que aumenta el tamaño de una matriz

Por ahora, se usará función **solve** como una opción para invertir una matriz.

```
A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
solve(A)
```

```
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

Arrays

Así como una matriz (un **rectángulo** de elementos) es el resultado de incrementar la dimensión de un vector (una **línea** de elementos), la dimensión de una matriz puede aumentarse para obtener estructuras de datos más complejas.

En R, los vectores y las matrices pueden ser considerados casos especiales del un arreglo más general llamado **array**, que es cómo se refiere a este tipo de estructuras cuando tienen más de dos dimensiones.

Para crear estas estructuras usamos la función **array** y el argumento **dim**

```
AR <- array(data=1:24,dim=c(3,4,2))
```

Descripción gráfica de un array

Un diagrama conceptual de un array 3 x 4 x 2.

En este ejemplo, observamos el orden de las dimensiones suministradas a **dim: c(filas, columnas, capas)**

Un ejemplo

Una matriz de cuatro dimensiones es el siguiente paso hacia arriba y se puede pensar en bloques de matrices tridimensionales.

Supongamos que tenemos una matriz de cuatro dimensiones compuesta de tres copias de AR. Esta nueva matriz se puede almacenar en R como sigue (una vez más, la matriz se copleta en columnas):

```
BR <- array(data=rep(1:24,times=3),dim=c(3,4,2,3))
BR
```

Continuación. . .

Con BR ahora tiene tres copias de AR. Cada una de estas copias se divide en sus dos capas para que R pueda imprimir el objeto en la pantalla.

Como antes, las filas están indexadas por el primer dígito, las columnas por el segundo dígito y las capas por el tercer dígito. El nuevo cuarto dígito indexa los bloques.

Subconjuntos, extracciones y reemplazos

Aunque los objetos de alta dimensión pueden ser difíciles de conceptualizar, R los indexa consistentemente. Esto hace que extraer elementos de estas estructuras sea sencillo.

Supongamos que se desea la segunda fila de la segunda capa de la matriz AR. Puedes introducir estas ubicaciones dimensionales exactas de AR en corchetes.

```
AR[2,,2]
```

```
[1] 14 17 20 23
```

Ejemplos

Si se desea conseguir elementos específicos de este vector, digamos el tercero y el primero, en ese orden, podemos hacer lo siguiente:

```
AR[2,c(3,1),2]
```

```
[1] 20 14
```

Una extracción que da lugar a múltiples vectores se presenta como columnas en la matriz resultante . Por ejemplo, para extraer las primeras filas de ambas capas de AR, se hace lo siguiente

```
AR[1,,]
```

Ejemplos 1

El siguiente muestra un único elemento de la segunda fila y la primera columna de la matriz en la primera capa de la matriz tridimensional situado en el tercer bloque.

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

El código siguiente devuelve todos los valores en la primera fila del primer bloque.

Desde que los índices de columna y capa están en blanco en este subconjunto *[1, , , 1]*, el comando ha devuelto valores para las cuatro columnas y ambas capas en ese bloque de BR.

```
BR[1,,,1]
```

Ejemplos 2

La siguiente línea devuelve todos los valores de la segunda capa de la matriz BR, compuesta de tres matrices

```
BR[,,2,]
```

En términos generales, si tiene una extracción que da como resultado matrices d dimensionales, el resultado será una matriz de la siguiente dimensión $d + 1$.

En el último ejemplo, extraemos múltiples matrices (bidimensionales) y el resultado fue devuelto como una matriz tridimensional.

Sobre eliminación y sobreescritura de elementos

La supresión y sobrescritura de elementos en arrays de alta dimensión sigue las mismas reglas que para vectores y matrices.

Especifica las posiciones de la dimensión de la misma manera, utilizando índices negativos (para borrar) o utilizando el operador de asignación para sobrescribir.

Lecturas

- The Book of R A First Course in Programming and Statistics. Tilman M. Davies 2016.

Clases

Todas las variables en R tienen una clase, que te dice qué tipos de variables son. Por ejemplo, la mayoría de los números tienen clase **numeric** y los valores lógicos tienen la clase **logic**. El tipo de datos más pequeño en R es un **vector**.

Podemos averiguar la clase de una variable utilizando **class(variable)**:

```
class(c(TRUE, FALSE))
```

```
[1] "logical"
```

Las variables también tienen un tipo de almacenamiento interno (accedido a través de **typeof**), un modo (**mode**) y un modo de almacenamiento (**storage.mode**).

Tipos de almacenamiento interno

Los tipos, modos y modos de almacenamiento existen en su mayoría para propósitos heredados, por lo que en la práctica sólo necesitarás usar **class** de un objeto.

```
set.seed(2)
x <- 1:10
typeof(x)
```

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

```
y <- x/5 + rnorm(10)
typeof(y)
```

```
[1] "double"
```

```
g <- lm(y ~ x)
typeof(g)
```

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

Diferentes tipos de números

R contiene tres clases diferentes de variable numérica: **numeric** para valores en coma flotante, **integer** para los enteros y **complex** para los números complejos. Podemos decir cuál es cuál examinando la clase de la variable:

```
class(sqrt(1:10))
```

```
[1] "numeric"
```

```
class(4L)
```

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

```
class(1 + 2i)
```

```
[1] "complex"
```

```
class(0.5:4.5)
```

```
[1] "numeric"
```

Otras clases

Además de las tres clases numéricas y la clase lógica que hemos visto, hay tres clases más de vectores: **caracter** para almacenar el texto, **factors** para almacenar datos categóricos y **raw** para almacenar datos binarios.

```
class(c("ella", "es", "bella", "ella", "es", "una", "estrella"))
```

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

R tiene una solución más sofisticada para datos categóricos utilizando **factores** que son números enteros con etiquetas:

```
genero <- factor(c("masculino", "femenino", "femenino", "masculino", "femenino"))
```

Otras clases(1)

El contenido del factor se parece mucho a su equivalente de caracteres: obtiene etiquetas legibles para cada valor que se limitan a valores específicos conocidos como los **niveles del factor**:

```
nlevels(genero)
```

```
[1] 2
```

Por defecto, los niveles de factor se asignan alfabéticamente. Los valores deL factor se almacenan como números enteros en lugar de caracteres. Podemos ver esto más claramente usando **as.integer**:

```
as.integer(genero)
```

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

Otras clases(2)

left: 60% La clase **raw** almacena vectores de bytes raw. Cada byte está representado por un valor hexadecimal de dos dígitos. Estos se utilizan principalmente para almacenar el contenido de archivos binarios importados. Los números enteros 0 a 255 se pueden convertir en raw utilizando **as.raw**. Para las cadenas, **as.raw** no funciona, en este caso se debe usar **charToRaw** en su lugar:

```
as.raw(1:5)
```

```
[1] 01 02 03 04 05
```

```
letra <- charToRaw("R!")
letra
```

```
[1] 52 21
```

```
class(letra)
```

```
[1] "raw"
```

Clase matrix

Además de las clases de vectores mencionadas, existen muchos otros tipos de variables, por ejemplo las matrices contienen datos multidimensionales, y las matrices con la clase **matrix** son un caso especial de matrices bidimensionales.

```
x <- matrix( c(6,7), nrow=2 )
class(x)
```

```
[1] "matrix"
```

Comprobación y cambio de clases

Llamar a la función **class** es útil para examinar interactivamente las variables en el prompt de comandos, pero si queremos conocer el tipo de un objeto , es mejor usar la función **is** o una de sus variantes específicas de clase.

En una situación típica, nuestra prueba se verá como:

```
if(!is(x, "alguna_clase"))
{
   ...
}
```

La mayoría de las clases comunes tienen sus propias funciones `is.*` y llamarlas normalmente es un poco más eficiente que usar la función general **is**.

Comprobación y cambio de clases(1)

Por ejemplo:

```
is.character("rojo verde, amarillo fresa")
```

```
[1] TRUE
```

```
is.numeric(3L)
```

```
[1] TRUE
```

```
is.logical(FALSE)
```

```
[1] TRUE
```

```
is.list(list(a = 1, b = 2))
```

```
[1] TRUE
```

```
is.integer(3)
```

```
[1] FALSE
```

Comprobación y cambio de clases(2)

La función **is.numeric** devuelve **TRUE** para enteros así como valores de punto flotante. Si queremos probar sólo números de coma flotante, entonces debemos usar **is.double**. Sin embargo, esto no suele ser necesario, ya que R está diseñado para que los valores de punto flotante y enteros se puedan usar de forma más o menos intercambiable. Si se agrega el sufijo L el número se convierte en un entero.

```
is.integer(2L)
```

```
[1] TRUE
```

Podemos ver una lista completa de todas las funciones **is** del paquete base con el siguiente código:

```
ls(pattern = "^is", baseenv())
```

Casting

left: 69% A veces deseamos cambiar el tipo de un objeto. Esto se llama **casting** y la mayoría de funciones `is*` tienen una función correspondiente `as*` para lograrlo.

Las funciones especializadas como `as*` deben ser usadas sobre **as** cuando estén disponibles, ya que son generalmente más eficientes y a menudo contienen una lógica adicional específica para cada clase.

```
x <- "123.456"
as(x, "numeric")
```

```
[1] 123.456
```

```
as.numeric(x)
```

```
[1] 123.456
```

Examinando Variables

Siempre que hayamos hecho un cálculo o el nombre de una variable en la consola de R, el resultado se imprime. Esto sucede porque R llama implícitamente al método **print** del objeto.

Dentro de bucles o funciones, la impresión automática no ocurre, por lo que tenemos que llamar explícitamente a print:

```
vector1 <- c(1, 8, 27, 64)
for(i in vector1) i # no se imprime
for(i in vector1) print(i)
```

```
[1] 1
[1] 8
[1] 27
[1] 64
```

Examinando Variables(1)

Es útil algún tipo de resumen del objeto. La función **summary** hace exactamente eso, dando la información apropiada para diferentes tipos de datos. Las variables numéricas se resumen con la media, mediana y algunos cuantiles.

```
num <- runif(30)
summary(num)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.01041 0.19470 0.49537 0.51163 0.81035 0.98173
```

Examinando Variables(2)

Los vectores categóricos y lógicos se resumen por los conteos de cada valor. En este ejemplo, **letters** es una constante incorporada que contiene los valores en minúsculas de a a la z (**LETTERS** contiene los equivalentes en mayúsculas, A a Z).

```
fac <- factor(sample(letters[1:6], 30, replace = TRUE))
summary(fac)
```

```
a b c d e f
4 8 4 3 6 5
```

Examinando Variables(3)

En este caso, se presenta el conteo de valores en un muestreo aleatorio con `replace` de la función **sample**:

```
bool <- sample(c(TRUE, FALSE, NA), 30, replace = TRUE)
summary(bool)
```

```
   Mode   FALSE    TRUE    NA's
logical      12      10       8
```

Examinando Variables(4)

Los objetos multidimensionales, como matrices y data frames, se resumen en columnas. En un data frame de 30 filas y objetos más grandes es necesario reducir la información, usando la función **head** que se puede

utilizar para mostrar sólo las primeras filas (seis por defecto):

```
dfr <- data.frame(num, fac, bool)
summary(dfr)
```

```
      num          fac      bool
 Min.   :0.01041   a:4   Mode :logical
 1st Qu.:0.19470   b:8   FALSE:12
 Median :0.49537   c:4   TRUE :10
 Mean   :0.51163   d:3   NA's :8
 3rd Qu.:0.81035   e:6
 Max.   :0.98173   f:5
```

Estructura de un objeto

La función **str** muestra la estructura del objeto. No es tan interesante para los vectores, pero str es muy útil para los data frames y listas anidadas:

```
str(dfr)
```

```
'data.frame':   30 obs. of  3 variables:
 $ num : num  0.662 0.388 0.837 0.151 0.347 ...
 $ fac : Factor w/ 6 levels "a","b","c","d",..: 1 1 5 6 2 5 5 6 4 5 ...
 $ bool: logi  FALSE NA TRUE FALSE TRUE NA ...
```

Más sobre clases

La función **unclass** se puede utilizar para evitar que en la muestra de un objeto, se pierda información útil, permitiendo ver cómo se construye una variable.

```
unclass(fac)
```

```
 [1] 1 1 5 6 2 5 5 6 4 5 5 6 4 2 6 3 3 3 2 1 2 2 1 2 2 5 2 6 3 4
attr(,"levels")
[1] "a" "b" "c" "d" "e" "f"
```

Es útil saber que la función **attributes** proporciona una lista de todos los atributos que pertenecen a un objeto:

```
attributes(fac)
```

El espacio de trabajo en R

Es bueno saber los nombres de las variables que hemos creado y lo que contienen. Para enumerar los nombres de las variables existentes, se utiliza la función **ls**. Este nombre lleva el nombre del comando Unix equivalente y sigue la misma convención: por defecto, los nombres de las variables que comienzan con un . están escondidos. Para verlos, pase el argumento **all.names = TRUE**:

```
r <- 1
b <- "Jessica"
c <- TRUE
ls()
```

```
 [1] "a"               "A"               "a_vector"
 [4] "a.concatenacion" "algunos.numeros" "algunos.pares"
 [7] "AR"              "b"               "B"
```

```
[10] "bool"            "c"               "dfr"
[13] "fac"             "g"               "genero"
[16] "i"               "letra"           "m"
[19] "mymat"           "num"             "numb5a20"
[22] "r"               "vector1"         "x"
[25] "y"
```

Flujo de control

En R, al igual que en otros lenguajes, hay muchos casos en los que es posible que se desee ejecutar código de forma condicional o ejecutar código similar repetidamente.

La ejecución condicionada vectorizada a través de la función *ifelse* es también parte de R. Debido a la naturaleza vectorizada de R y algunas alternativas más estéticas, estos bucles son menos utilizados en R de lo que se pueda esperar.

If-else

La forma más simple de control de flujo es la ejecución condicional usando *if*. *if* toma un valor lógico (más precisamente, un vector lógico de longitud uno) y ejecuta la siguiente instrucción sólo si ese valor es TRUE:

```
if(TRUE) message("Es cierto!")
## Eso fue cierto
if(FALSE) message("No es cierto!")
```

If-else

Por supuesto, la mayor parte del tiempo, no pasaremos valores TRUE o FALSE. En su lugar, colocaremos una variable o una expresión: si supieras que la sentencia iba a ejecutarse de antemano, no necesitaría la cláusula if. En este ejemplo siguiente, runif(1) genera un número aleatorio uniformemente distribuido entre 0 y 1. Si ese valor es superior a 0,5, entonces se muestra un mensaje.

```
if(runif(1) > 0.5) message("Este mensaje aparece con un  50% probabilidad.")
```

If-else

El siguiente paso en la complejidad de if es incluir una declaración else. El código que sigue a una instrucción else se ejecuta si la condición if era FALSE:

```
if(FALSE)
{
    message("Esto no se ejecuta....")
} else
{
    message("esto deberia ejecutarse.")
}
```

El if vectorizado

La instrucción estándar if toma un único valor lógico. Si pasas un vector lógico con una longitud de más de uno (!no lo hagas!), R te advertirá que has dado varias opciones y sólo se usará la primera:

```r
if(c(TRUE, FALSE)) message("dos opciones")
```

Dado que gran parte de R es vectorizada, también es vectorizado el control de flujo, en forma de la función ifelse. ifelse toma tres argumentos. El primero es un vector lógico de condiciones. El segundo contiene valores que se devuelven cuando el primer vector es TRUE. El tercero contiene valores que se devuelven cuando el primer vector es FALSE.

El if vectorizado

En el siguiente ejemplo, rbinom genera números aleatorios de una distribución binomial para simular el lanzamiento de una moneda :

```r
ifelse(rbinom(10, 1, 0.5), "Cara", "Sello")
```

```
 [1] "Cara"  "Sello" "Cara"  "Cara"  "Cara"  "Cara"  "Sello" "Cara"
 [9] "Cara"  "Sello"
```

Bucles

Hay tres tipos de bucles en R: repeat, while, y for y aunque la vectorización muestra que nos los necesita tanto en R como en otros lenguajes, todavía pueden ayudar en la ejecución repetida del código.

Bucle while

El bucles while, verifica primero el código y luego (tal vez) se ejecuta. Dado que la comprobación ocurre al principio, es posible que el contenido del bucle nunca se ejecute.

```r
# Un ejemplo numerico

i <- 0
while (i < 12){
  i <- i + 2
  print(i)
}
```

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
[1] 12
```

Bucle for

El tercer tipo de bucle se utiliza cuando se sabe exactamente cuántas veces desea que el código se repita. El bucle for acepta una variable de iteración y un vector. La sintaxis para el bucle **for** es

```
for (nombre in valores ) expresion
```

El bucle for, iteraciona a través de los componentes nombre de valores uno a la vez. nombre toma el valor de cada elemento sucesivo de valores, hasta que se complete sus componentes.

Bucle for

```
lenguajes <- c("Python", "JS", "C", "C++", "R", "Bash")
for(l in lenguajes){
  print(l)
}
```

```
[1] "Python"
[1] "JS"
[1] "C"
[1] "C++"
[1] "R"
[1] "Bash"
```

Bucle for

Los bucles for son flexibles en el sentido de que no se limitan a números enteros. Podemos pasar vectores de caracteres, vectores lógicos o listas:

```
l <- list(
    pi,
    LETTERS[1:5],
    charToRaw("R es superfacil..."),
    list(
        TRUE
    )
)
for(i in l)
{
    print(i)
}
```

```
[1] 3.141593
[1] "A" "B" "C" "D" "E"
 [1] 52 20 65 73 20 73 75 70 65 72 66 61 63 69 6c 2e 2e 2e
[[1]]
[1] TRUE
```

Bucle for

Comparando la velocidad de un bucle for con una version vectorizada

```
n = 1000000
x=0
system.time(for (i in 1:n) x = sin(i/n))
```

```
   user  system elapsed
  0.109   0.000   0.109
```

```
i = 1:n # vectorizado
system.time(sin(i/n))
```

```
   user  system elapsed
  0.011   0.004   0.015
```

Funciones

Escribir tus propias funciones en R permite al usuario combinar un conjunto de comandos de R en una función fácil de llamar y con capacidad de ser generalizable. Las funciones son fundamentales para R. Para poder convertirse en un usuario más avanzado o desarrollador de R, una buena comprensión de qué son las funciones y cómo escribirlas es crucial.

Lo más importante para entender acerca de las funciones en R es que las funciones son objetos por derecho propio. Se puede trabajar con funciones, exactamente de la misma manera que se trabaja con cualquier otro tipo de objeto.

Componentes de una función

Con algunas excepciones, la mayoría de las funciones en R tienen tres componentes:

- formals, lista de argumentos que controla cómo se puede llamar a la función.
- body, el código dentro de la función.
- environment, que determina la ubicación de las variables de la función.

```
f <- function(x, y = 5) {
x + y
}
```

Funciones primitivas

Como se mencionó, hay una excepción a la regla de que las funciones tienen tres componentes. Las funciones primitivas, como sum(), llaman al código C directamente con .Primitive() y no contienen código R. Por lo tanto las funciones formals(), body() y environment() son NULL.

```
sum
```

```
function (..., na.rm = FALSE)  .Primitive("sum")
```

```
formals(sum)
```

```
NULL
```

Funciones primitivas

```
body(sum)
```

```
NULL
```

```
environment(sum)
```

```
NULL
```

Las funciones primitivas sólo se encuentran en el paquete base y puesto que funcionan a un nivel bajo, pueden ser más eficientes y pueden tener reglas diferentes para las correspondencia de argumentos (por ejemplo, switch y call). Esto, sin embargo, tiene un costo de comportarse diferente de todas las demás funciones en R.

Creando y llamando funciones

Para crear nuestras propias funciones, solo las asignamos como cualquier otra variable. A modo de ejemplo, vamos a crear una función para calcular la longitud de la hipotenusa de un triángulo rectángulo (para

simplificar, usaremos el algoritmo obvio, para el código del mundo real, esto no funciona bien con números grandes y muy pequeños, por lo que no debe calcular la hipotenusa de esta manera):

```
hipotenusa <- function(x, y)
{
    sqrt(x ^ 2 + y ^ 2)
}
```

Creando y llamando funciones

Ahora podemos llamar a esta función como lo haríamos con cualquier otra:

```
hipotenusa(3, 4)
```

```
[1] 5
```

```
hipotenusa(y = 24, x = 7)
```

```
[1] 25
```

Creando y llamando funciones

Cuando llamamos a una función, si no nombramos los argumentos, entonces R los hará coincidir con la posición. En el caso de `hipotenusa(3, 4)`, 3 viene en primer lugar por lo que se asigna a x, 4 viene segundo, por lo que se asigna a y.

Si queremos cambiar el orden en que pasamos los argumentos, entonces podemos pasar argumentos con nombre. En el caso de `hipotenusa(y = 24, x = 7)`, aunque pasamos las variables en el orden "erróneo", R todavía determina correctamente qué variable debe asignarse a x, y a y.

No tiene mucho sentido para una función de cálculo como hipotenusa, pero si quisiéramos, podríamos proporcionar valores por defecto para x e y. En esta nueva versión, si no pasamos nada a la función, x toma el valor 5 e y toma el valor de 12:

Creando y llamando funciones

```
hipotenusa <- function(x = 5, y = 12)
{
    sqrt(x ^ 2 + y ^ 2)
}
hipotenusa() #equivalente a hipotenusa(5, 12)
```

```
[1] 13
```

La función formals ecupera los argumentos de una función como una lista (par). La función args hace lo mismo de una manera más legible por el usuario, pero menos programable. formalArgs devuelve un vector de caracteres de los nombres de los argumentos:

Creando y llamando funciones

```
formals(hipotenusa)
```

```
$x
[1] 5
```

```
$y
[1] 12
```

```
args(hipotenusa)
```

```
function (x = 5, y = 12)
NULL
```

```
formalArgs(hipotenusa)
```

```
[1] "x" "y"
```

Creando y llamando funciones

```
formalArgs(hipotenusa)
```

```
[1] "x" "y"
```

El cuerpo de una función se recupera utilizando la función body. Esto no suele ser muy útil por sí solo, pero a veces queremos examinarlo como texto, para encontrar funciones que llaman a otra función, por ejemplo. Podemos usar deparse para lograr esto.

Veamos como se hace:

Creando y llamando funciones

```
(cuerpo_de_funcion_hipotenusa <- body(hipotenusa))
```

```
{
    sqrt(x^2 + y^2)
}
```

```
deparse(cuerpo_de_funcion_hipotenusa)
```

```
[1] "{"                   "    sqrt(x^2 + y^2)" "}"
```

Más sobre funciones

Los valores predeterminados dados a argumentos formales de funciones pueden ser algo más que valores constantes - podemos pasar cualquier código de R a ellos, e incluso usar otros argumentos formales. La siguiente función, normaliza, escala un vector. Los argumentos m y s son, por defecto, la media y la desviación estándar del primer argumento, de modo que el vector devuelto tendrá una media de 0 y una desviación estándar de 1:

```
normaliza <- function(x, m = mean(x), s = sd(x)){
    (x - m) / s
}
normalizado <- normaliza(c(1, 3, 6, 10, 15))
```

Más sobre funciones

```
mean(normalizado)
```

```
[1] -5.572799e-18
```

```
sd(normalizado)
```

```
[1] 1
```

Hay un pequeño problema con nuestra función de normalización, ya que si alguno de los elementos de x son valores NA, no funcionaria:

Más sobre funciones

```
normaliza(c(1, 3, 6, 10, NA))
```

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

Si hay elementos de un vector faltante, entonces por defecto, mean y sd devolverán NA. Por consiguiente, nuestra función normaliza devuelve valores de NA en todas partes. Podría ser preferible tener la opción de devolver solamente los valores de NA donde la entrada fue NA.

Tanto mean como sd tienen un argumento, na.rm, que nos permite eliminar los valores faltantes antes de que se produzcan los cálculos. Para evitar todos los valores de NA, podríamos incluir tal argumento en normalize:

Más sobre funciones

```
normaliza <- function(x, m = mean(x, na.rm = na.rm),
                      s = sd(x, na.rm = na.rm), na.rm = FALSE)
{
    (x - m) / s
}
normaliza(c(1, 3, 6, 10, NA))
```

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

```
normaliza(c(1, 3, 6, 10, NA), na.rm = TRUE)
```

```
[1] -1.0215078 -0.5107539  0.2553770  1.2768848         NA
```

Más sobre funciones

Esto funciona, pero la sintaxis es un poco simple. Para ahorrarnos tener que escribir explícitamente los nombres de los argumentos que no son realmente utilizados por la función (na.rm sólo se pasa a mean y sd), R tiene un argumento especial, ... , que contiene todos los argumentos que no coinciden con la posición o el nombre:

```
normaliza <- function(x, m = mean(x, ...), s = sd(x, ...), ...)
{
    (x - m) / s
}
normaliza(c(1, 3, 6, 10, NA))
```

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

Más sobre funciones

```
normaliza(c(1, 3, 6, 10, NA), na.rm = TRUE)
```

```
[1] -1.0215078 -0.5107539  0.2553770  1.2768848         NA
```

Ahora en la llamada a `normaliza(c (1, 3, 6, 10, NA), na.rm = TRUE)`, el argumento `na.rm` no coincide con ninguno de los argumentos formales de normaliza, ya que no es x o m o s. Eso significa que se almacena en el argumento `...` de `normaliza`. Cuando evaluamos m, la expresión `mean(x, ...)` es ahora `mean(x, na.rm = TRUE)`.

Mayor información

- Aplicaciones de vectores y matrices.
- Probabilidad en R.
- Combinatoria en R
- Lista y data frames.
- Bucles avanzados-primera parte.
- Estructuras avanzadas de R.