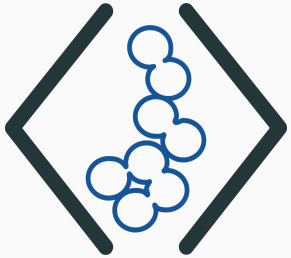

Paquetes en R (II)

Carlos Vergara-Hernández

Jueves, 8 de junio de 2017

Área de Desigualdades en Salud, FISABIO

Índice de contenidos

El problema de las dependencias

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

Pero ¿qué está pasando entre bastidores cuando instalamos un paquete?

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

Pero ¿qué está pasando entre bastidores cuando instalamos un paquete?

> *Everything that exist is an object. Everything that happens is a function call.*
>
> — *John Chambers*

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

Pero ¿qué está pasando entre bastidores cuando instalamos un paquete?

> *Everything that exist is an object. Everything that happens is a function call.*
>
> — *John Chambers*

Esto implica que

- se ejecuta todo el código contenido en los scripts del directorio R,

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

Pero ¿qué está pasando entre bastidores cuando instalamos un paquete?

> *Everything that exist is an object. Everything that happens is a function call.*
>
> — *John Chambers*

Esto implica que

- se ejecuta todo el código contenido en los scripts del directorio R,
  - esto produce tantos objetos (normalmente funciones) como asignaciones haya en los scripts.

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

Pero ¿qué está pasando entre bastidores cuando instalamos un paquete?

> *Everything that exist is an object. Everything that happens is a function call.*
>
> — *John Chambers*

Esto implica que

- se ejecuta todo el código contenido en los scripts del directorio R,
    - esto produce tantos objetos (normalmente funciones) como asignaciones haya en los scripts.
- solo se guardan aquellos objetos incluidos en el archivo *NAMESPACE* (con la función `export()`),

La instalación de paquetes

Es muy recomendable utilizar funciones de otros paquetes (sobre todo de aquellos que tienen un mantenimiento excelente: no es necesario reinventar la rueda).

Pero ¿qué está pasando entre bastidores cuando instalamos un paquete?

> *Everything that exist is an object. Everything that happens is a function call.*
>
> — *John Chambers*

Esto implica que

- se ejecuta todo el código contenido en los scripts del directorio R,
    - esto produce tantos objetos (normalmente funciones) como asignaciones haya en los scripts.
- solo se guardan aquellos objetos incluidos en el archivo *NAMESPACE* (con la función `export()`),
- los objetos guardados están disponibles en el directorio de destino del paquete.

Acceso a paquetes instalados

Es posible acceder al contenido de un paquete instalado con las instrucciones `library()` o `require()`, pero también empleando `::` y `:::` (aunque este último no es muy recomendable).

Acceso a paquetes instalados

Es posible acceder al contenido de un paquete instalado con las instrucciones `library()` o `require()`, pero también empleando `::` y `:::` (aunque este último no es muy recomendable).

```
library(devtools) # O require(devtools)
session_info()[[1]]
```

```
#  setting  value
#  version  R version 3.4.0 (2017-04-21)
#  system   x86_64, linux-gnu
#  ui       X11
#  language (EN)
#  collate  es_ES.UTF-8
#  tz       Europe/Madrid
#  date     2017-06-07
```

Acceso a paquetes instalados

Es posible acceder al contenido de un paquete instalado con las instrucciones `library()` o `require()`, pero también empleando `::` y `:::` (aunque este último no es muy recomendable).

```
library(devtools) # O require(devtools)
session_info()[[1]]
```

```
#  setting  value
#  version  R version 3.4.0 (2017-04-21)
#  system   x86_64, linux-gnu
#  ui       X11
#  language (EN)
#  collate  es_ES.UTF-8
#  tz       Europe/Madrid
#  date     2017-06-07
```

```
detach("package:devtools", unload = TRUE)
devtools::session_info()[[1]]
```

```
#  setting  value
#  version  R version 3.4.0 (2017-04-21)
#  system   x86_64, linux-gnu
#  ui       X11
#  language (EN)
#  collate  es_ES.UTF-8
#  tz       Europe/Madrid
#  date     2017-06-07
```

```
session_info()[[1]]
```

```
# Error in session_info(): no se pudo encontrar la función "session_info"
```

Acceso a paquetes instalados: destripando R

R guarda información acerca de las librerías (directorios) donde se almacenan los paquetes instalados.

Acceso a paquetes instalados: destripando R

R guarda información acerca de las librerías (directorios) donde se almacenan los paquetes instalados.

- Por ejemplo, podemos consultar el listado de paquetes instalados y obtener los primeros resultados por orden alfabético:

Acceso a paquetes instalados: destripando R

R guarda información acerca de las librerías (directorios) donde se almacenan los paquetes instalados.

- Por ejemplo, podemos consultar el listado de paquetes instalados y obtener los primeros resultados por orden alfabético:

```
rutas <- .libPaths()
directorios <- list.dirs(rutas, recursive = FALSE)
head(basename(directorios), 12)
```

```
#  [1] "abind"         "acepack"       "animation"     "ape"
#  [5] "aqp"           "arm"           "assertthat"    "backports"
#  [9] "base64"        "base64enc"     "bayesplot"     "BayesValidate"
```

Acceso a paquetes instalados: destripando R

R guarda información acerca de las librerías (directorios) donde se almacenan los paquetes instalados.

- Por ejemplo, podemos consultar el listado de paquetes instalados y obtener los primeros resultados por orden alfabético:

```
rutas <- .libPaths()
directorios <- list.dirs(rutas, recursive = FALSE)
head(basename(directorios), 12)
```

```
#  [1] "abind"         "acepack"       "animation"     "ape"
#  [5] "aqp"           "arm"           "assertthat"    "backports"
#  [9] "base64"        "base64enc"     "bayesplot"     "BayesValidate"
```

Además, R tiene un sistema de trabajo organizado en entornos, el cual comienza desde que se inicia el programa.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).
- Al mismo tiempo se genera un entorno de trabajo denominado `.GlobalEnv` o entorno global, y es ahí donde se almacena todo lo que produces durante tu sesión (salvo que crees otro entorno, algo poco habitual). Este es el entorno que tiene prioridad en toda la jerarquía.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).
- Al mismo tiempo se genera un entorno de trabajo denominado `.GlobalEnv` o entorno global, y es ahí donde se almacena todo lo que produces durante tu sesión (salvo que crees otro entorno, algo poco habitual). Este es el entorno que tiene prioridad en toda la jerarquía.
- A medida que vas cargando paquetes con `library()` o `require()`, estos se van agregando a la jerarquía, justo por detrás del entorno global.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).
- Al mismo tiempo se genera un entorno de trabajo denominado `.GlobalEnv` o entorno global, y es ahí donde se almacena todo lo que produces durante tu sesión (salvo que crees otro entorno, algo poco habitual). Este es el entorno que tiene prioridad en toda la jerarquía.
- A medida que vas cargando paquetes con `library()` o `require()`, estos se van agregando a la jerarquía, justo por detrás del entorno global.

Y todo esto ¿qué implica?

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).
- Al mismo tiempo se genera un entorno de trabajo denominado `.GlobalEnv` o entorno global, y es ahí donde se almacena todo lo que produces durante tu sesión (salvo que crees otro entorno, algo poco habitual). Este es el entorno que tiene prioridad en toda la jerarquía.
- A medida que vas cargando paquetes con `library()` o `require()`, estos se van agregando a la jerarquía, justo por detrás del entorno global.

Y todo esto ¿qué implica?

1. En el entorno global, si creas un objeto que tiene el mismo nombre que alguno de los objetos definidos en el resto de entornos, el primero será el que se utilice.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).
- Al mismo tiempo se genera un entorno de trabajo denominado `.GlobalEnv` o entorno global, y es ahí donde se almacena todo lo que produces durante tu sesión (salvo que crees otro entorno, algo poco habitual). Este es el entorno que tiene prioridad en toda la jerarquía.
- A medida que vas cargando paquetes con `library()` o `require()`, estos se van agregando a la jerarquía, justo por detrás del entorno global.

Y todo esto ¿qué implica?

1. En el entorno global, si creas un objeto que tiene el mismo nombre que alguno de los objetos definidos en el resto de entornos, el primero será el que se utilice.
2. La única forma de seleccionar una función concreta es utilizando `::`, instrucción que permite acceder al contenido de un paquete pero sin adjuntar todo su contenido a la jerarquía de entornos.

Entornos de trabajo al detalle

R se organiza en entornos de trabajo jerárquicos.

- Es posible consultar los entornos disponibles en cualquier momento mediante la instrucción `search()`.
- Cuando inicias una sesión de R, el software adjunta los entornos de unos paquetes desde su inicio (`base`, `methods`, `stats`, `utils`, entre otros).
- Al mismo tiempo se genera un entorno de trabajo denominado `.GlobalEnv` o entorno global, y es ahí donde se almacena todo lo que produces durante tu sesión (salvo que crees otro entorno, algo poco habitual). Este es el entorno que tiene prioridad en toda la jerarquía.
- A medida que vas cargando paquetes con `library()` o `require()`, estos se van agregando a la jerarquía, justo por detrás del entorno global.

Y todo esto ¿qué implica?

1. En el entorno global, si creas un objeto que tiene el mismo nombre que alguno de los objetos definidos en el resto de entornos, el primero será el que se utilice.
2. La única forma de seleccionar una función concreta es utilizando `::`, instrucción que permite acceder al contenido de un paquete pero sin adjuntar todo su contenido a la jerarquía de entornos.

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

```
set.seed(1)
un_vector <- rnorm(50)
str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

```
set.seed(1)
un_vector <- rnorm(50)
str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
str <- function(x) mean(x)
str(un_vector)
```

```
# [1] 0.1004483
```

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

```
set.seed(1)
un_vector <- rnorm(50)
str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
str <- function(x) mean(x)
str(un_vector)
```

```
# [1] 0.1004483
```

```
utils::str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

```
set.seed(1)
un_vector <- rnorm(50)
str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
str <- function(x) mean(x)
str(un_vector)
```

```
# [1] 0.1004483
```

```
utils::str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
rm(str, envir = .GlobalEnv)
identical(str(un_vector), utils::str(un_vector))
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
# [1] TRUE
```

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

```
set.seed(1)
un_vector <- rnorm(50)
str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
str <- function(x) mean(x)
str(un_vector)
```

```
# [1] 0.1004483
```

```
utils::str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
rm(str, envir = .GlobalEnv)
identical(str(un_vector), utils::str(un_vector))
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
# [1] TRUE
```

Entornos: un ejemplo del desastre

Vamos a liarla parda: ¿qué pasa si definimos un objeto empleando un nombre contenido en el entorno de un paquete?

```
set.seed(1)
un_vector <- rnorm(50)
str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
str <- function(x) mean(x)
str(un_vector)
```

```
# [1] 0.1004483
```

```
utils::str(un_vector)
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
rm(str, envir = .GlobalEnv)
identical(str(un_vector), utils::str(un_vector))
```

```
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
#  num [1:50] -0.626 0.184 -0.836 1.595 0.33 ...
# [1] TRUE
```

Moraleja

Evita nombrar objetos que ya estén definidos en otra parte (una búsqueda rápida ayuda), y borra el entorno global al inicio de cada sesión.

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.
- `source()`: ejecutamos un script directamente (con funciones definidas o con una parte del análisis — p. ej., limpieza o descriptiva—).

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.
- `source()`: ejecutamos un script directamente (con funciones definidas o con una parte del análisis — p. ej., limpieza o descriptiva—).

El problema

`library()` y `require()` adjuntan el entorno de un paquete, el cual contiene los objetos que se crearon cuando se instaló el paquete, mientras que `source()` puede sobrescribir el nombre de alguno de los objetos en los entornos adjuntados o a adjuntar en el futuro.

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.
- `source()`: ejecutamos un script directamente (con funciones definidas o con una parte del análisis — p. ej., limpieza o descriptiva—).

El problema

`library()` y `require()` adjuntan el entorno de un paquete, el cual contiene los objetos que se crearon cuando se instaló el paquete, mientras que `source()` puede sobrescribir el nombre de alguno de los objetos en los entornos adjuntados o a adjuntar en el futuro.

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.
- `source()`: ejecutamos un script directamente (con funciones definidas o con una parte del análisis — p. ej., limpieza o descriptiva—).

El problema

`library()` y `require()` adjuntan el entorno de un paquete, el cual contiene los objetos que se crearon cuando se instaló el paquete, mientras que `source()` puede sobrescribir el nombre de alguno de los objetos en los entornos adjuntados o a adjuntar en el futuro.

La solución

Al desarrollar paquetes jamás se utiliza `source()`, `library()` o `require()`, sino que siempre se emplea `::`.

Al crear un paquete con dependencias debemos asegurar que estas se instalan junto a nuestro paquete (gracias al archivo *DESCRIPTION*).

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.
- `source()`: ejecutamos un script directamente (con funciones definidas o con una parte del análisis — p. ej., limpieza o descriptiva—).

El problema

`library()` y `require()` adjuntan el entorno de un paquete, el cual contiene los objetos que se crearon cuando se instaló el paquete, mientras que `source()` puede sobrescribir el nombre de alguno de los objetos en los entornos adjuntados o a adjuntar en el futuro.

La solución

Al desarrollar paquetes jamás se utiliza `source()`, `library()` o `require()`, sino que siempre se emplea `::`.

Al crear un paquete con dependencias debemos asegurar que estas se instalan junto a nuestro paquete (gracias al archivo *DESCRIPTION*).

Trabajo diario y el desarrollo de paquetes

Pensemos un segundo acerca de cómo trabajamos en nuestro día a día en un script.

- `library()` o `require()`: indicamos los paquetes que vamos a utilizar en la sesión de trabajo.
- `source()`: ejecutamos un script directamente (con funciones definidas o con una parte del análisis — p. ej., limpieza o descriptiva—).

El problema

`library()` y `require()` adjuntan el entorno de un paquete, el cual contiene los objetos que se crearon cuando se instaló el paquete, mientras que `source()` puede sobrescribir el nombre de alguno de los objetos en los entornos adjuntados o a adjuntar en el futuro.

La solución

Al desarrollar paquetes jamás se utiliza `source()`, `library()` o `require()`, sino que siempre se emplea `::`.

Al crear un paquete con dependencias debemos asegurar que estas se instalan junto a nuestro paquete (gracias al archivo *DESCRIPTION*).

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.
  - `Depends` instala y adjunta el paquete a la jerarquía de entornos: las funciones son siempre accesibles por su nombre.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.
  - `Depends` instala y adjunta el paquete a la jerarquía de entornos: las funciones son siempre accesibles por su nombre.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.
  - `Depends` instala y adjunta el paquete a la jerarquía de entornos: las funciones son siempre accesibles por su nombre.
- ¿Cuál es recomendable usar?

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.
  - `Depends` instala y adjunta el paquete a la jerarquía de entornos: las funciones son siempre accesibles por su nombre.
- ¿Cuál es recomendable usar?

Recomendación de uso

Se recomienda usar `Imports`, y reservar `Depends` únicamente para especificar una versión de R concreta (opción por defecto en `devtools::create()`).

Por ejemplo: `Depends: R (>= 3.4.0)`.

Imports vs. Depends

En el archivo *DESCRIPTION* existen de dos campos donde marcar dependencias: `Imports` y `Depends`.

- ¿Cuál es la diferencia exacta entre ambas?
  - `Imports` instala el paquete en la librería: las funciones son accesibles mediante la estructura `paquete::función()`.
  - `Depends` instala y adjunta el paquete a la jerarquía de entornos: las funciones son siempre accesibles por su nombre.
- ¿Cuál es recomendable usar?

Recomendación de uso

Se recomienda usar `Imports`, y reservar `Depends` únicamente para especificar una versión de R concreta (opción por defecto en `devtools::create()`).

Por ejemplo: `Depends: R (>= 3.4.0)`.

Implementación en `devtools`

El archivo *DESCRIPTION* puede editarse a mano, aunque una forma más sencilla es usar la función `devtools::use_package()`, marcando para el argumento `type` la opción "*Imports*".

Una función de ejemplo: `resumen()`

Imaginemos que tenemos pululando por un script una función para extraer un resumen (media, desviación típica, cuantil .5, asimetría y curtosis) de un banco de datos numérico.

Una función de ejemplo: `resumen()`

Imaginemos que tenemos pululando por un script una función para extraer un resumen (media, desviación típica, cuantil .5, asimetría y curtosis) de un banco de datos numérico.

Esta función incorpora dos dependencias en el trabajo con el banco de datos (paquetes `dplyr` y `tidyr`) y otra para el cálculo de la asimetría y la curtosis (paquete `e1071`). La función que tendríamos en nuestro script sería algo como:

Una función de ejemplo: `resumen()`

Imaginemos que tenemos pululando por un script una función para extraer un resumen (media, desviación típica, cuantil .5, asimetría y curtosis) de un banco de datos numérico.

Esta función incorpora dos dependencias en el trabajo con el banco de datos (paquetes **dplyr** y **tidyr**) y otra para el cálculo de la asimetría y la curtosis (paquete **e1071**). La función que tendríamos en nuestro script sería algo como:

```
library(dplyr); library(e1071); library(tidyr)
resumen <- function(datos) {
  stopifnot(is.data.frame(datos))
  stopifnot(all(sapply(datos, class) == "numeric"))
  mi_funs <- funs(
    media     = mean(., na.rm = T),
    sd        = sd(., na.rm = TRUE),
    q50       = quantile(., probs = 0.50, na.rm = TRUE),
    curtosis  = kurtosis(., na.rm = TRUE),
    asimetria = skewness(., na.rm = TRUE)
  )
  calculo     <- summarise_all(datos, mi_funs)
  es_caracter <- mutate_all(calculo, as.character)
  agrupado    <- gather(es_caracter, stat, val)
  separado    <- separate(agrupado, stat, into = c("variable", "stat"), sep = "_")
  resumen     <- spread(separado, stat, val)
  resumen     <- mutate_each(resumen, funs(as.numeric), -1)
  return(resumen)
}
```

Una función de ejemplo: `resumen()`

Imaginemos que tenemos pululando por un script una función para extraer un resumen (media, desviación típica, cuantil .5, asimetría y curtosis) de un banco de datos numérico.

Esta función incorpora dos dependencias en el trabajo con el banco de datos (paquetes **dplyr** y **tidyr**) y otra para el cálculo de la asimetría y la curtosis (paquete **e1071**). La función que tendríamos en nuestro script sería algo como:

```
library(dplyr); library(e1071); library(tidyr)
resumen <- function(datos) {
  stopifnot(is.data.frame(datos))
  stopifnot(all(sapply(datos, class) == "numeric"))
  mi_funs <- funs(
    media     = mean(., na.rm = T),
    sd        = sd(., na.rm = TRUE),
    q50       = quantile(., probs = 0.50, na.rm = TRUE),
    curtosis  = kurtosis(., na.rm = TRUE),
    asimetria = skewness(., na.rm = TRUE)
  )
  calculo     <- summarise_all(datos, mi_funs)
  es_caracter <- mutate_all(calculo, as.character)
  agrupado    <- gather(es_caracter, stat, val)
  separado    <- separate(agrupado, stat, into = c("variable", "stat"), sep = "_")
  resumen     <- spread(separado, stat, val)
  resumen     <- mutate_each(resumen, funs(as.numeric), -1)
  return(resumen)
}
```

¿Cómo quedaría esta función dentro de un paquete?

Gestión de dependencias en `resumen()`

Nuestro punto de partida siempre será asegurar que las dependencias se instalen junto a nuestro paquete gracias al archivo *DESCRIPTION* —podemos usar `devtools::use_package("pkg", type = "Imports")`—:

Gestión de dependencias en `resumen()`

Nuestro punto de partida siempre será asegurar que las dependencias se instalen junto a nuestro paquete gracias al archivo *DESCRIPTION* —podemos usar `devtools::use_package("pkg", type = "Imports")`—:

```
Imports:
    dplyr (>= 0.5.0),
    e1071 (>= 1.6-8),
    tidyr (>= 0.6.2)
```

Gestión de dependencias en `resumen()`

Nuestro punto de partida siempre será asegurar que las dependencias se instalen junto a nuestro paquete gracias al archivo *DESCRIPTION* —podemos usar `devtools::use_package("pkg", type = "Imports")`—:

```
Imports:
    dplyr (>= 0.5.0),
    e1071 (>= 1.6-8),
    tidyr (>= 0.6.2)
```

A partir de ahí, ya podemos incorporar las buenas prácticas en el desarrollo de la función.

Gestión de dependencias en `resumen()`

Nuestro punto de partida siempre será asegurar que las dependencias se instalen junto a nuestro paquete gracias al archivo *DESCRIPTION* —podemos usar `devtools::use_package("pkg", type = "Imports")`—:

```
Imports:
    dplyr (>= 0.5.0),
    e1071 (>= 1.6-8),
    tidyr (>= 0.6.2)
```

A partir de ahí, ya podemos incorporar las buenas prácticas en el desarrollo de la función.

```
resumen <- function(datos) {
  stopifnot(is.data.frame(datos))
  stopifnot(all(sapply(datos, class) == "numeric"))
  mi_funs <- dplyr::funs(
    media     = mean(., na.rm = T),
    sd        = sd(., na.rm = TRUE),
    q50       = quantile(., probs = 0.50, na.rm = TRUE),
    curtosis  = e1071::kurtosis(., na.rm = TRUE),
    asimetria = e1071::skewness(., na.rm = TRUE)
  )
  calculo     <- dplyr::summarise_all(datos, mi_funs)
  es_caracter <- dplyr::mutate_all(calculo, as.character)
  agrupado    <- tidyr::gather_(es_caracter, "stat", "val", colnames(es_caracter))
  separado    <- tidyr::separate_(agrupado, "stat", into = c("variable", "stat"), sep = "_")
  resumen     <- tidyr::spread_(separado, "stat", "val")
  resumen     <- dplyr::mutate_each_(resumen, dplyr::funs(as.numeric), colnames(resumen)[-1])
  return(resumen)
}
```

Documentando paquetes

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.
- La sintaxis de la documentación guarda una gran semejanza con la que encontramos en LaTeX, aunque no es idéntica.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.
- La sintaxis de la documentación guarda una gran semejanza con la que encontramos en LaTeX, aunque no es idéntica.
    - Los archivos usan la extensión '.Rd'.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.
- La sintaxis de la documentación guarda una gran semejanza con la que encontramos en LaTeX, aunque no es idéntica.
  - Los archivos usan la extensión '.Rd'.
  - El formato final puede ser HTML, texto plano o un archivo PDF.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.
- La sintaxis de la documentación guarda una gran semejanza con la que encontramos en LaTeX, aunque no es idéntica.
  - Los archivos usan la extensión '.Rd'.
  - El formato final puede ser HTML, texto plano o un archivo PDF.

¿A quién beneficia?

Generar una buena ayuda permite sacar el máximo partido al paquete, tanto al usuario como a futuros desarrolladores que se unan al proyecto.

El beneficio es para todo usuario del paquete, que puede ser cualquiera o tú mismo dentro de dos meses.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.
- La sintaxis de la documentación guarda una gran semejanza con la que encontramos en LaTeX, aunque no es idéntica.
  - Los archivos usan la extensión '.Rd'.
  - El formato final puede ser HTML, texto plano o un archivo PDF.

¿A quién beneficia?

Generar una buena ayuda permite sacar el máximo partido al paquete, tanto al usuario como a futuros desarrolladores que se unan al proyecto.

El beneficio es para todo usuario del paquete, que puede ser cualquiera o tú mismo dentro de dos meses.

Documentación de objetos

Un objeto (habitualmente una función, aunque también podría ser una base de datos) sin documentar —sin un archivo de ayuda— no solo es difícil de usar, sino que a medio plazo también es difícil de mantener.

- R tiene un sistema propio de documentación de funciones (accesible mediante `?` o `help()`).
- Los archivos de documentación se almacenan en el directorio `./man/` del directorio raíz.
- La sintaxis de la documentación guarda una gran semejanza con la que encontramos en LaTeX, aunque no es idéntica.
  - Los archivos usan la extensión '.Rd'.
  - El formato final puede ser HTML, texto plano o un archivo PDF.

¿A quién beneficia?

Generar una buena ayuda permite sacar el máximo partido al paquete, tanto al usuario como a futuros desarrolladores que se unan al proyecto.

El beneficio es para todo usuario del paquete, que puede ser cualquiera o tú mismo dentro de dos meses.

Ejemplo de documentación: `resumen.Rd`

Siguiendo con el ejemplo que vimos hace un momento, veamos cómo podría ser el archivo `resumen.Rd` que la documenta.

Ejemplo de documentación: `resumen.Rd`

Siguiendo con el ejemplo que vimos hace un momento, veamos cómo podría ser el archivo `resumen.Rd` que la documenta.

```
\name{resumen}
\title{Resumen de un banco de datos numérico}
\usage{
  resumen(datos)
}
\arguments{
  \item{datos}{Un banco de datos numérico de clase \code{data.frame}.}
}
\value{
  La función devuelve un \code{data.frame} con la media, desviación típica, cuantil 0.5 y
    asimetría y curtosis de las variables del banco de datos numérico.
}
\description{
  Realizar un resumen (de andar por casa) de un banco de datos numérico.
}
\examples{
  set.seed(1)
  resumen(data.frame(a = rnorm(100), b = rnorm(100)))
}
```

La pesadilla de la documentación

Pongámonos en situación:

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.
3. Adaptamos nuestra función siguiendo las prácticas de escritura en paquetes.

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.
3. Adaptamos nuestra función siguiendo las prácticas de escritura en paquetes.
4. Escribimos la documentación de nuestra función editando desde cero un archivo con la extensión `.Rd`.

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.
3. Adaptamos nuestra función siguiendo las prácticas de escritura en paquetes.
4. Escribimos la documentación de nuestra función editando desde cero un archivo con la extensión `.Rd`.
5. Creamos el paquete y empezamos a trabajar con él en nuestro día a día.

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.
3. Adaptamos nuestra función siguiendo las prácticas de escritura en paquetes.
4. Escribimos la documentación de nuestra función editando desde cero un archivo con la extensión `.Rd`.
5. Creamos el paquete y empezamos a trabajar con él en nuestro día a día.

En un mundo perfecto el trabajo terminaría aquí, pero el mundo no es perfecto y pronto nos veremos en la necesidad de editar la función y su documentación (por un *bug* o una mejora).

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.
3. Adaptamos nuestra función siguiendo las prácticas de escritura en paquetes.
4. Escribimos la documentación de nuestra función editando desde cero un archivo con la extensión `.Rd`.
5. Creamos el paquete y empezamos a trabajar con él en nuestro día a día.

En un mundo perfecto el trabajo terminaría aquí, pero el mundo no es perfecto y pronto nos veremos en la necesidad de editar la función y su documentación (por un *bug* o una mejora).

La pesadilla de la documentación

Pongámonos en situación:

1. Hemos escrito una función y queremos darle forma de paquete de R.
2. Gestionamos las posibles dependencias del paquete.
3. Adaptamos nuestra función siguiendo las prácticas de escritura en paquetes.
4. Escribimos la documentación de nuestra función editando desde cero un archivo con la extensión `.Rd`.
5. Creamos el paquete y empezamos a trabajar con él en nuestro día a día.

En un mundo perfecto el trabajo terminaría aquí, pero el mundo no es perfecto y pronto nos veremos en la necesidad de editar la función y su documentación (por un *bug* o una mejora).

Solución

Incluir la documentación dentro de cada script, y dejar que los paquetes `roxygen2` y `devtools` se ocupen de adaptarla al formato `.Rd`.

De este modo, cada vez que editemos la función, será más fácil que a su vez recordemos editar su documentación.

Documentación de objetos: `roxygen2` y `devtools`

Ejemplo de comentarios `roxygen2`:

Documentación de objetos: roxygen2 y devtools

Ejemplo de comentarios roxygen2:

```
#' @title Resumen de un banco de datos numérico
#' @usage resumen(datos)
#' @param datos Un banco de datos numérico de clase \code{data.frame}.
#' @return La función devuelve un \code{data.frame} con la media, desviación típica, cuantil 0.5 y
#'  asimetría y curtosis de las variables del banco de datos numérico.
#' @description Realizar un resumen (de andar por casa) de un banco de datos numérico.
#' @examples set.seed(1)
#' resumen(data.frame(a = rnorm(100), b = rnorm(100)))
resumen <- function(datos) {
  stopifnot(is.data.frame(datos))
  stopifnot(all(sapply(datos, class) == "numeric"))
  mi_funs <- dplyr::funs(
    media     = mean(., na.rm = T),
    sd        = sd(., na.rm = TRUE),
    q50       = quantile(., probs = 0.50, na.rm = TRUE),
    curtosis  = e1071::kurtosis(., na.rm = TRUE),
    asimetria = e1071::skewness(., na.rm = TRUE)
  )
  calculo     <- dplyr::summarise_all(datos, mi_funs)
  es_caracter <- dplyr::mutate_all(calculo, as.character)
  agrupado    <- tidyr::gather_(es_caracter, "stat", "val", colnames(es_caracter))
  separado    <- tidyr::separate_(agrupado, "stat", into = c("variable", "stat"), sep = "_")
  resumen     <- tidyr::spread_(separado, "stat", "val")
  resumen     <- dplyr::mutate_each_(resumen, dplyr::funs(as.numeric), colnames(resumen)[-1])
  return(resumen)
}
```

Documentación de objetos: roxygen2 y devtools

Ejemplo de comentarios roxygen2:

```
#' @title Resumen de un banco de datos numérico
#' @usage resumen(datos)
#' @param datos Un banco de datos numérico de clase \code{data.frame}.
#' @return La función devuelve un \code{data.frame} con la media, desviación típica, cuantil 0.5 y
#'  asimetría y curtosis de las variables del banco de datos numérico.
#' @description Realizar un resumen (de andar por casa) de un banco de datos numérico.
#' @examples set.seed(1)
#' resumen(data.frame(a = rnorm(100), b = rnorm(100)))
resumen <- function(datos) {
  stopifnot(is.data.frame(datos))
  stopifnot(all(sapply(datos, class) == "numeric"))
  mi_funs <- dplyr::funs(
    media     = mean(., na.rm = T),
    sd        = sd(., na.rm = TRUE),
    q50       = quantile(., probs = 0.50, na.rm = TRUE),
    curtosis  = e1071::kurtosis(., na.rm = TRUE),
    asimetria = e1071::skewness(., na.rm = TRUE)
  )
  calculo     <- dplyr::summarise_all(datos, mi_funs)
  es_caracter <- dplyr::mutate_all(calculo, as.character)
  agrupado    <- tidyr::gather_(es_caracter, "stat", "val", colnames(es_caracter))
  separado    <- tidyr::separate_(agrupado, "stat", into = c("variable", "stat"), sep = "_")
  resumen     <- tidyr::spread_(separado, "stat", "val")
  resumen     <- dplyr::mutate_each_(resumen, dplyr::funs(as.numeric), colnames(resumen)[-1])
  return(resumen)
}
```

Una vez introducidos los comentarios tipo roxygen2, podemos usar la función devtools::document() para crear los archivos .Rd (uno para cada función).

El resultado final

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/kk.R
\name{resumen}
\title{Resumen de un banco de datos numérico}
\usage{
  resumen(datos)
}
\arguments{
  \item{datos}{Un banco de datos numérico de clase \code{data.frame}.}
}
\value{
  La función devuelve un \code{data.frame} con la media, desviación típica, cuantil 0.5 y
    asimetría y curtosis de las variables del banco de datos numérico.
}
\description{
  Realizar un resumen (de andar por casa) de un banco de datos numérico.
}
\examples{
  set.seed(1)
  resumen(data.frame(a = rnorm(100), b = rnorm(100)))
}
```

Exportar e importar funciones

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
    - nombre y descripción del paquete,

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
    - nombre y descripción del paquete,
    - identificación de autores,

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
    - nombre y descripción del paquete,
    - identificación de autores,
    - elección de una licencia.

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
    - nombre y descripción del paquete,
    - identificación de autores,
    - elección de una licencia.
3. conversión de una función al formato paquete:

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
    - nombre y descripción del paquete,
    - identificación de autores,
    - elección de una licencia.
3. conversión de una función al formato paquete:
    - gestión de dependencias en *DESCRIPTION* —edición manual o mediante `devtools::use_package()`—,

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
    - nombre y descripción del paquete,
    - identificación de autores,
    - elección de una licencia.
3. conversión de una función al formato paquete:
    - gestión de dependencias en *DESCRIPTION* —edición manual o mediante `devtools::use_package()`—,
    - especificación de las funciones externas mediante la llamada `::`.

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
   - nombre y descripción del paquete,
   - identificación de autores,
   - elección de una licencia.
3. conversión de una función al formato paquete:
   - gestión de dependencias en *DESCRIPTION* —edición manual o mediante `devtools::use_package()`—,
   - especificación de las funciones externas mediante la llamada `::`.
4. documentación de objetos mediante `roxygen2` y `devtools`

Recapitulación de contenido

Llegados a este punto, los pasos que hemos ido viendo han sido:

1. crear la estructura de paquete con la función `devtools::create()`.
2. edición del archivo *DESCRIPTION*:
   - nombre y descripción del paquete,
   - identificación de autores,
   - elección de una licencia.
3. conversión de una función al formato paquete:
   - gestión de dependencias en *DESCRIPTION* —edición manual o mediante `devtools::use_package()`—,
   - especificación de las funciones externas mediante la llamada `::`.
4. documentación de objetos mediante `roxygen2` y `devtools`

WTF!

Si en este momento construyes e instalas el paquete, al adjuntarlo y tratar de usarlo verías que... ¡NO APARECE NINGUNA FUNCIÓN!

¿Qué está pasando?

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

> *En el archivo NAMESPACE se importan funciones de otros paquetes, código foráneo a R, se definen clases y métodos, y se exportan las funciones de nuestro paquete.*

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

> *En el archivo NAMESPACE se importan funciones de otros paquetes, código foráneo a R, se definen clases y métodos, y se exportan las funciones de nuestro paquete.*

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

> *En el archivo NAMESPACE se importan funciones de otros paquetes, código foráneo a R, se definen clases y métodos, y se exportan las funciones de nuestro paquete.*

- ¡El motivo por el que nuestro paquete no contiene ninguna función operativa es porque todavía no la hemos exportado en el archivo *NAMESPACE*!

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

> *En el archivo NAMESPACE se importan funciones de otros paquetes, código foráneo a R, se definen clases y métodos, y se exportan las funciones de nuestro paquete.*

- ¡El motivo por el que nuestro paquete no contiene ninguna función operativa es porque todavía no la hemos exportado en el archivo *NAMESPACE*!
- Ahora que ya hemos visto cómo trabajar con `roxygen2`, no vamos a perder el tiempo en editar el archivo a mano.

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

> *En el archivo NAMESPACE se importan funciones de otros paquetes, código foráneo a R, se definen clases y métodos, y se exportan las funciones de nuestro paquete.*

- ¡El motivo por el que nuestro paquete no contiene ninguna función operativa es porque todavía no la hemos exportado en el archivo *NAMESPACE*!
- Ahora que ya hemos visto cómo trabajar con `roxygen2`, no vamos a perder el tiempo en editar el archivo a mano.
- En lugar de eso, podemos exportar nuestra función incorporando el siguiente comentario `roxygen2` justo sobre ella: `#' @export`.

El archivo *NAMESPACE*: one more time

En la clase anterior...

Hace un rato hablamos acerca del archivo *NAMESPACE*, pero quedó pendiente profundizar en él. Según dijimos...

> *En el archivo NAMESPACE se importan funciones de otros paquetes, código foráneo a R, se definen clases y métodos, y se exportan las funciones de nuestro paquete.*

- ¡El motivo por el que nuestro paquete no contiene ninguna función operativa es porque todavía no la hemos exportado en el archivo *NAMESPACE*!
- Ahora que ya hemos visto cómo trabajar con `roxygen2`, no vamos a perder el tiempo en editar el archivo a mano.
- En lugar de eso, podemos exportar nuestra función incorporando el siguiente comentario `roxygen2` justo sobre ella: `#' @export`.
- Ahora, al utilizar `devtools::document()` se actualizará el archivo *NAMESPACE* con la instrucción correcta.

Exportar funciones

Los pasos a seguir son muy sencillos:

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
resumen <- function(datos) {
  ...
}
```

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
resumen <- function(datos) {
  ...
}
```

2. Guardamos el script y ejecutamos.

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
resumen <- function(datos) {
  ...
}
```

2. Guardamos el script y ejecutamos.

```
devtools::document()
```

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
resumen <- function(datos) {
  ...
}
```

2. Guardamos el script y ejecutamos.

```
devtools::document()
```

3. El archivo *NAMESPACE* pasa de estar en blanco a incorporar contenido.

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
resumen <- function(datos) {
  ...
}
```

2. Guardamos el script y ejecutamos.

```
devtools::document()
```

3. El archivo *NAMESPACE* pasa de estar en blanco a incorporar contenido.

```
# Generated by roxygen2: do not edit by hand

export(resumen)
```

Exportar funciones

Los pasos a seguir son muy sencillos:

1. Incorporamos el comentario `#' @export`.

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
resumen <- function(datos) {
  ...
}
```

2. Guardamos el script y ejecutamos.

```
devtools::document()
```

3. El archivo *NAMESPACE* pasa de estar en blanco a incorporar contenido.

```
# Generated by roxygen2: do not edit by hand

export(resumen)
```

¡Al fin!

Ahora, tras construir, instalar y adjuntar el paquete, comprobamos que podemos utilizar nuestra función además de poder acceder a su ayuda.

Importar funciones

Cambiando de escenario, imaginemos que no nos acaba de convencer tener que utilizar `::` cada vez que queramos emplear una función contenida en otro paquete.

Importar funciones

Cambiando de escenario, imaginemos que no nos acaba de convencer tener que utilizar `::` cada vez que queramos emplear una función contenida en otro paquete.

- Una forma de solucionar este inconveniente es importar la función en el archivo *NAMESPACE* con la instrucción adecuada de `roxygen2`.

Importar funciones

Cambiando de escenario, imaginemos que no nos acaba de convencer tener que utilizar `::` cada vez que queramos emplear una función contenida en otro paquete.

- Una forma de solucionar este inconveniente es importar la función en el archivo *NAMESPACE* con la instrucción adecuada de `roxygen2`.
- Volviendo a nuestra función de resumen, el resultado sería:

Importar funciones

Cambiando de escenario, imaginemos que no nos acaba de convencer tener que utilizar `::` cada vez que queramos emplear una función contenida en otro paquete.

- Una forma de solucionar este inconveniente es importar la función en el archivo *NAMESPACE* con la instrucción adecuada de `roxygen2`.
- Volviendo a nuestra función de resumen, el resultado sería:

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
#' @importFrom dplyr funs summarise_all mutate_each_ mutate_all
#' @importFrom tidyr gather_ separate_ spread_
#' @importFrom e1071 kurtosis skewness
resumen <- function(datos) {
  ...
  mi_funs <- funs(
    media     = mean(., na.rm = T),
    ...
    curtosis  = kurtosis(., na.rm = TRUE),
    asimetria = skewness(., na.rm = TRUE)
  )
  return(resumen)
}
```

Importar funciones

Cambiando de escenario, imaginemos que no nos acaba de convencer tener que utilizar `::` cada vez que queramos emplear una función contenida en otro paquete.

- Una forma de solucionar este inconveniente es importar la función en el archivo *NAMESPACE* con la instrucción adecuada de `roxygen2`.
- Volviendo a nuestra función de resumen, el resultado sería:

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
#' @importFrom dplyr funs summarise_all mutate_each_ mutate_all
#' @importFrom tidyr gather_ separate_ spread_
#' @importFrom e1071 kurtosis skewness
resumen <- function(datos) {
  ...
  mi_funs <- funs(
    media     = mean(., na.rm = T),
    ...
    curtosis  = kurtosis(., na.rm = TRUE),
    asimetria = skewness(., na.rm = TRUE)
  )
  return(resumen)
}
```

Mi recomendación

En los scripts de un paquete utiliza siempre el contenido de otros paquetes mediante `::`.

Importar funciones

Cambiando de escenario, imaginemos que no nos acaba de convencer tener que utilizar `::` cada vez que queramos emplear una función contenida en otro paquete.

- Una forma de solucionar este inconveniente es importar la función en el archivo *NAMESPACE* con la instrucción adecuada de `roxygen2`.
- Volviendo a nuestra función de resumen, el resultado sería:

```
#' @title Resumen de un banco de datos numérico
#' ...
#' @export
#' @importFrom dplyr funs summarise_all mutate_each_ mutate_all
#' @importFrom tidyr gather_ separate_ spread_
#' @importFrom e1071 kurtosis skewness
resumen <- function(datos) {
  ...
  mi_funs <- funs(
    media     = mean(., na.rm = T),
    ...
    curtosis  = kurtosis(., na.rm = TRUE),
    asimetria = skewness(., na.rm = TRUE)
  )
  return(resumen)
}
```

Mi recomendación

En los scripts de un paquete utiliza siempre el contenido de otros paquetes mediante `::`.

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

- bucles eternos y simulaciones costosas,

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

- bucles eternos y simulaciones costosas,
- gestión de grandes bancos de datos.

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

- bucles eternos y simulaciones costosas,
- gestión de grandes bancos de datos.

El código C++ está dentro de archivos con extensión `.cpp` y se almacena en el directorio `./src/`.

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

- bucles eternos y simulaciones costosas,
- gestión de grandes bancos de datos.

El código C++ está dentro de archivos con extensión `.cpp` y se almacena en el directorio `./src/`.

Existen varios paquetes de R orientados a facilitar la sinergia entre ambos lenguajes, aunque el más importante es `Rcpp`. Para poder usarlo hay que añadir en cualquier parte de los comentarios `roxygen2`, y solo una vez, las siguientes dos líneas

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

- bucles eternos y simulaciones costosas,
- gestión de grandes bancos de datos.

El código C++ está dentro de archivos con extensión `.cpp` y se almacena en el directorio `./src/`.

Existen varios paquetes de R orientados a facilitar la sinergia entre ambos lenguajes, aunque el más importante es `Rcpp`. Para poder usarlo hay que añadir en cualquier parte de los comentarios `roxygen2`, y solo una vez, las siguientes dos líneas

```
#' @useDynLib nombre_del_paquete
#' @importFrom Rcpp sourceCpp
```

Código compilado: C++

Hay veces en que R no es lo suficientemente rápido y es ahí donde trabajar con un lenguaje de bajo nivel como C++ ofrece claras ventajas:

- bucles eternos y simulaciones costosas,
- gestión de grandes bancos de datos.

El código C++ está dentro de archivos con extensión `.cpp` y se almacena en el directorio `./src/`.

Existen varios paquetes de R orientados a facilitar la sinergia entre ambos lenguajes, aunque el más importante es Rcpp. Para poder usarlo hay que añadir en cualquier parte de los comentarios roxygen2, y solo una vez, las siguientes dos líneas

```
#' @useDynLib nombre_del_paquete
#' @importFrom Rcpp sourceCpp
```

Mi recomendación

Aunque la ganancia en velocidad es impresionante, esta no suele resultar especialmente relevante dada la inversión que conlleva.

Existen varios paquetes que incorporan rutinas en C++, resultando muy probable que alguno de ellos te ayude a resolver el problema que te preocupe.

> *No malgastes tu tiempo en el rendimiento de una función si esta todavía no produce ningún resultado.*

Uso de datos

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`.

.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`. Una vez instalado y adjuntado el paquete, los datos son accesibles con la instrucción `data(mis_datos)`.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`. Una vez instalado y adjuntado el paquete, los datos son accesibles con la instrucción `data(mis_datos)`. Instrucción clave: `devtools::use_data()`.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`. Una vez instalado y adjuntado el paquete, los datos son accesibles con la instrucción `data(mis_datos)`. Instrucción clave: `devtools::use_data()`.
- datos brutos (por ejemplo, formatos `.txt`, `.dat` o `.csv`): se almacenan en el directorio `./inst/extdata/`.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`. Una vez instalado y adjuntado el paquete, los datos son accesibles con la instrucción `data(mis_datos)`. Instrucción clave: `devtools::use_data()`.
- datos brutos (por ejemplo, formatos `.txt`, `.dat` o `.csv`): se almacenan en el directorio `./inst/extdata/`. Una vez instalado el paquete, la ruta a los datos es accesible con la instrucción `system.file("extdata", "mis_datos.csv", package = "paquete")`, tras la que ya podremos leerlos de la forma habitual.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`. Una vez instalado y adjuntado el paquete, los datos son accesibles con la instrucción `data(mis_datos)`. Instrucción clave: `devtools::use_data()`.
- datos brutos (por ejemplo, formatos `.txt`, `.dat` o `.csv`): se almacenan en el directorio `./inst/extdata/`. Una vez instalado el paquete, la ruta a los datos es accesible con la instrucción `system.file("extdata", "mis_datos.csv", package = "paquete")`, tras la que ya podremos leerlos de la forma habitual. Instrucción clave: `devtools::use_data_raw()`.

Datos en los paquetes

En muchas ocasiones querremos incorporar datos al paquete

- datos de muestra para los ejemplos,
- un paquete solo compuesto por datos.

¿Dónde guardar los datos?

Los datos se pueden almacenar en dos lugares dependiendo del formato:

- datos en formato de R (`.RData` o `.rda` y `.rds`): se almacenan en el directorio `./data/`. Una vez instalado y adjuntado el paquete, los datos son accesibles con la instrucción `data(mis_datos)`. Instrucción clave: `devtools::use_data()`.
- datos brutos (por ejemplo, formatos `.txt`, `.dat` o `.csv`): se almacenan en el directorio `./inst/extdata/`. Una vez instalado el paquete, la ruta a los datos es accesible con la instrucción `system.file("extdata", "mis_datos.csv", package = "paquete")`, tras la que ya podremos leerlos de la forma habitual. Instrucción clave: `devtools::use_data_raw()`.

Recuerda

Independientemente del formato escogido, es imprescindible documentar los datos, indicando su procedencia y descripción de variables.

Usando los datos

Imaginemos que tenemos unos datos relacionados con la distribución espacial de mortalidad.

Usando los datos

Imaginemos que tenemos unos datos relacionados con la distribución espacial de mortalidad.

- Utilizamos la instrucción `devtools::use_data(mis_datos)`, creándose el directorio `./data/` y se guarda el objeto de nuestra elección en un archivo `.rda`.

Usando los datos

Imaginemos que tenemos unos datos relacionados con la distribución espacial de mortalidad.

- Utilizamos la instrucción `devtools::use_data(mis_datos)`, creándose el directorio `./data/` y se guarda el objeto de nuestra elección en un archivo `.rda`.
- También creamos un script `./R/data.R` en el que describiremos los datos.

Usando los datos

Imaginemos que tenemos unos datos relacionados con la distribución espacial de mortalidad.

- Utilizamos la instrucción `devtools::use_data(mis_datos)`, creándose el directorio `./data/` y se guarda el objeto de nuestra elección en un archivo `.rda`.
- También creamos un script `./R/data.R` en el que describiremos los datos.
  - Aclara la tipología y características de los datos.
  - Si la clase que define los datos requiere un paquete que hasta ahora no figuraba como dependencia, es una buena práctica incorporarlo en la sección `Suggest` del archivo *DESCRIPTION* —`devtools:use_package("paquete", type = "Suggest")`—.

Usando los datos

Imaginemos que tenemos unos datos relacionados con la distribución espacial de mortalidad.

- Utilizamos la instrucción `devtools::use_data(mis_datos)`, creándose el directorio `./data/` y se guarda el objeto de nuestra elección en un archivo `.rda`.
- También creamos un script `./R/data.R` en el que describiremos los datos.
  - Aclara la tipología y características de los datos.
  - Si la clase que define los datos requiere un paquete que hasta ahora no figuraba como dependencia, es una buena práctica incorporarlo en la sección `Suggest` del archivo *DESCRIPTION* —`devtools:use_package("paquete", type = "Suggest")`—.

```
#' @title Mortalidad por IAM en hombres en Aragón (1991-2000)
#' @description Base de datos que contiene la mortalidad observada y esperada
#'   para hombres por IAM en 729 municipios de Aragón entre 1991 y 2001 y la
#'   cartografía de la región.
#' @format Un Spatial Polygons Data Frame (paquete \code{sp}) con 729 observaciones y
#'   seis variables:
#' \describe{
#'   \item{area}{área del municipio}
#'   \item{perimeter}{perímetro del municipio}
#'   \item{muni_id}{id INE del municipio}
#'   \item{nombre}{nombre del municipio}
#'   \item{observada}{mortalidad observada}
#'   \item{esperada}{mortalidad esperada}
#' }
"aragon_iam"
```

Programar pruebas

Pruebas de código: ¿qué son y por qué usarlas?

Las pruebas de código son una formalización explícita que aseguran que el código es funcional.

Pruebas de código: ¿qué son y por qué usarlas?

Las pruebas de código son una formalización explícita que aseguran que el código es funcional.

Gracias a ellas, podemos

Pruebas de código: ¿qué son y por qué usarlas?

Las pruebas de código son una formalización explícita que aseguran que el código es funcional.

Gracias a ellas, podemos

- detectar errores (*bugs*) fácilmente,

Pruebas de código: ¿qué son y por qué usarlas?

Las pruebas de código son una formalización explícita que aseguran que el código es funcional.

Gracias a ellas, podemos

- detectar errores (*bugs*) fácilmente,
- estructurar mejor el código (romper funciones largas en funciones más pequeñas),

Pruebas de código: ¿qué son y por qué usarlas?

Las pruebas de código son una formalización explícita que aseguran que el código es funcional.

Gracias a ellas, podemos

- detectar errores (*bugs*) fácilmente,
- estructurar mejor el código (romper funciones largas en funciones más pequeñas),
- generar un código más robusto de cara a futuros desarrollos.

Pruebas de código: ¿qué son y por qué usarlas?

Las pruebas de código son una formalización explícita que aseguran que el código es funcional.

Gracias a ellas, podemos

- detectar errores (*bugs*) fácilmente,
- estructurar mejor el código (romper funciones largas en funciones más pequeñas),
- generar un código más robusto de cara a futuros desarrollos.

Pruebas informales y pruebas formales

Todos solemos utilizar cláusulas condicionales del tipo `stop()`, `stopifnot()` o `warning()` en nuestras funciones.

Las pruebas de código explícitas permiten encapsular todo este contenido en un único lugar, y aunque su proposición exige una inversión de tiempo, sin duda merece la pena a medio o largo plazo.

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,
- añade `testthat` al campo `Suggest` en el archivo *DESCRIPTION*,

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,
- añade `testthat` al campo `Suggest` en el archivo *DESCRIPTION*,
- crea el archivo `.test/testthat.R` que lanza todas las pruebas al comprobar el estado del paquete (`R CMD check`).

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,
- añade `testthat` al campo `Suggest` en el archivo *DESCRIPTION*,
- crea el archivo `.test/testthat.R` que lanza todas las pruebas al comprobar el estado del paquete (`R CMD check`).

A partir de aquí, el flujo de trabajo es muy sencillo:

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,
- añade `testthat` al campo `Suggest` en el archivo *DESCRIPTION*,
- crea el archivo `.test/testthat.R` que lanza todas las pruebas al comprobar el estado del paquete (`R CMD check`).

A partir de aquí, el flujo de trabajo es muy sencillo:

1. Adapta las pruebas para el código que te genere cierta inseguridad,

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,
- añade `testthat` al campo `Suggest` en el archivo *DESCRIPTION*,
- crea el archivo `.test/testthat.R` que lanza todas las pruebas al comprobar el estado del paquete (`R CMD check`).

A partir de aquí, el flujo de trabajo es muy sencillo:

1. Adapta las pruebas para el código que te genere cierta inseguridad,
2. Averigua si el paquete supera las pruebas ejecutando `devtools::test()`,

Pruebas en R: el paquete `testthat`

El paquete `testthat` de R nos ayuda a elaborar conjuntos de pruebas para asegurar que el comportamiento de las funciones es el esperado. Para usarlo en el paquete que desarrollamos basta con emplear

```
devtools::use_testthat()
```

Esta sencilla instrucción...

- crea el directorio `./test/testthat/`,
- añade `testthat` al campo `Suggest` en el archivo *DESCRIPTION*,
- crea el archivo `.test/testthat.R` que lanza todas las pruebas al comprobar el estado del paquete (`R CMD check`).

A partir de aquí, el flujo de trabajo es muy sencillo:

1. Adapta las pruebas para el código que te genere cierta inseguridad,
2. Averigua si el paquete supera las pruebas ejecutando `devtools::test()`,
3. Si algo falla, adapta el código o las pruebas y repite lo anterior hasta superarlas todas.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.
    - Todas las expectativas comienzan con `expect_`.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.
    - Todas las expectativas comienzan con `expect_`.
2. Pruebas: agrupa un conjunto de expectativas relacionadas en una función.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.
    - Todas las expectativas comienzan con `expect_`.
2. Pruebas: agrupa un conjunto de expectativas relacionadas en una función.
    - Todas las pruebas se construyen con `test_that("mensaje", {expectativas})`.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.
    - Todas las expectativas comienzan con `expect_`.
2. Pruebas: agrupa un conjunto de expectativas relacionadas en una función.
    - Todas las pruebas se construyen con `test_that("mensaje", {expectativas})`.
3. Archivos: agrupa un conjunto de pruebas dentro de un mismo contexto.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.
    - Todas las expectativas comienzan con `expect_`.
2. Pruebas: agrupa un conjunto de expectativas relacionadas en una función.
    - Todas las pruebas se construyen con `test_that("mensaje", {expectativas})`.
3. Archivos: agrupa un conjunto de pruebas dentro de un mismo contexto.
    - Los archivos comienzan cargando el paquete que desarrollamos y señalando un contexto con `context("Resumen del conjunto de pruebas")`.

Trabajando con `testthat`

Las pruebas se organizan de forma jerárquica en tres niveles:

1. Expectativas: es la unidad mínima de prueba y describe el resultado esperado de una función.
    - Todas las expectativas comienzan con `expect_`.
2. Pruebas: agrupa un conjunto de expectativas relacionadas en una función.
    - Todas las pruebas se construyen con `test_that("mensaje", {expectativas})`.
3. Archivos: agrupa un conjunto de pruebas dentro de un mismo contexto.
    - Los archivos comienzan cargando el paquete que desarrollamos y señalando un contexto con `context("Resumen del conjunto de pruebas")`.

Recomendación

No escribas mensajes crípticos para las pruebas: el desarrollador que los lea no sabrá de qué va la película y no podrá identificar dónde falla nuestra función.

Pruebas en la práctica: `resumen()`

Vamos a programar una sencilla prueba para nuestra función `resumen()`.

Pruebas en la práctica: `resumen()`

Vamos a programar una sencilla prueba para nuestra función `resumen()`.

Antes de continuar, reflexionemos un instante acerca de lo que se espera que devuelva la función como resultado:

Pruebas en la práctica: `resumen()`

Vamos a programar una sencilla prueba para nuestra función `resumen()`.

Antes de continuar, reflexionemos un instante acerca de lo que se espera que devuelva la función como resultado:

- Un objeto de clase `data.frame` con seis columnas, cinco de las cuales son estadísticos con valores numéricos.

Pruebas en la práctica: `resumen()`

Vamos a programar una sencilla prueba para nuestra función `resumen()`.

Antes de continuar, reflexionemos un instante acerca de lo que se espera que devuelva la función como resultado:

- Un objeto de clase `data.frame` con seis columnas, cinco de las cuales son estadísticos con valores numéricos.

```
library(paquete)
context("Resultado de clase y dimensiones correctas")
datos_prueba <- data.frame(a = rnorm(100), b = rnorm(100))
test_that("resumen devuelve un data.frame", {
  expect_is(resumen(datos_prueba), class = "data.frame")
  expect_error(resumen("a"))
  expect_error(resumen(TRUE))
  expect_error(resumen(as.list(datos_prueba)))
})
test_that("resumen devuelve valores de clase numeric", {
  expect_true(all(apply(resumen(datos_prueba)[, -1], 1, is.numeric)))
})
test_that("resumen devuelve cinco estadísticos", {
  expect_length(resumen(datos_prueba)[,-1], 5)
})
```

Pruebas en la práctica: `resumen()`

Vamos a programar una sencilla prueba para nuestra función `resumen()`.

Antes de continuar, reflexionemos un instante acerca de lo que se espera que devuelva la función como resultado:

- Un objeto de clase `data.frame` con seis columnas, cinco de las cuales son estadísticos con valores numéricos.

```
library(paquete)
context("Resultado de clase y dimensiones correctas")
datos_prueba <- data.frame(a = rnorm(100), b = rnorm(100))
test_that("resumen devuelve un data.frame", {
  expect_is(resumen(datos_prueba), class = "data.frame")
  expect_error(resumen("a"))
  expect_error(resumen(TRUE))
  expect_error(resumen(as.list(datos_prueba)))
})
test_that("resumen devuelve valores de clase numeric", {
  expect_true(all(apply(resumen(datos_prueba)[, -1], 1, is.numeric)))
})
test_that("resumen devuelve cinco estadísticos", {
  expect_length(resumen(datos_prueba)[,-1], 5)
})
```

Para comprobar si se superan las pruebas, basta con ejecutar

```
devtools::test()
```

Construir viñetas

Demos

La ayuda de una función puede estar bien si sabemos el nombre de la función que nos interesa y esta está escrita con cierta gracia.

Demos

La ayuda de una función puede estar bien si sabemos el nombre de la función que nos interesa y esta está escrita con cierta gracia.

- No obstante, el contenido de la ayuda no guarda una relación óptima con el resto del contenido del paquete (falta de contexto).
- Los ejemplos son limitados.

Demos

La ayuda de una función puede estar bien si sabemos el nombre de la función que nos interesa y esta está escrita con cierta gracia.

- No obstante, el contenido de la ayuda no guarda una relación óptima con el resto del contenido del paquete (falta de contexto).
- Los ejemplos son limitados.

Para suplir esta carencia, en los paquetes se pueden programar demostraciones, las cuales se orientan a mostrar y describir como resolver un problema.

Demos

La ayuda de una función puede estar bien si sabemos el nombre de la función que nos interesa y esta está escrita con cierta gracia.

- No obstante, el contenido de la ayuda no guarda una relación óptima con el resto del contenido del paquete (falta de contexto).
- Los ejemplos son limitados.

Para suplir esta carencia, en los paquetes se pueden programar demostraciones, las cuales se orientan a mostrar y describir como resolver un problema.

- una demostración es un script de R almacenado dentro del directorio `./demo/`,
- toda demostración debe listarse en el archivo `./demo/00Index` usando una estructura nombre-espacio-descripción,
- su ejecución es dinámica,
- `R CMD check` no prueba las demostraciones: pueden dejar de funcionar con facilidad,
- el texto tiene un formato de salida de consola (sin formato, vamos).

Viñetas

Las carencias de las demostraciones nos dirigen hacia la creación de viñetas, las cuales actúan a modo de artículos, concentrando en un único documento texto, código y las salidas que este produce (incluyendo gráficos y tablas).

Viñetas

Las carencias de las demostraciones nos dirigen hacia la creación de viñetas, las cuales actúan a modo de artículos, concentrando en un único documento texto, código y las salidas que este produce (incluyendo gráficos y tablas).

- Las viñetas se almacenan en el directorio `./vignettes/`.
- Se trata de documentos `rmarkdown`, con los que ya estamos familiarizados, pero incorporan los siguientes metadatos:

Viñetas

Las carencias de las demostraciones nos dirigen hacia la creación de viñetas, las cuales actúan a modo de artículos, concentrando en un único documento texto, código y las salidas que este produce (incluyendo gráficos y tablas).

- Las viñetas se almacenan en el directorio `./vignettes/`.
- Se trata de documentos `rmarkdown`, con los que ya estamos familiarizados, pero incorporan los siguientes metadatos:

```
---
title: "Título"
author: "Autor"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

Viñetas

Las carencias de las demostraciones nos dirigen hacia la creación de viñetas, las cuales actúan a modo de artículos, concentrando en un único documento texto, código y las salidas que este produce (incluyendo gráficos y tablas).

- Las viñetas se almacenan en el directorio `./vignettes/`.
- Se trata de documentos `rmarkdown`, con los que ya estamos familiarizados, pero incorporan los siguientes metadatos:

```
---
title: "Título"
author: "Autor"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

- a día de hoy, el formato más habitual para las viñetas HTML.
- Recuerda añadir posibles dependencias de las viñetas en el campo `Suggest` del archivo *DESCRIPTION* (como `knitr` y `rmarkdown`, entre otras).

Viñetas

Las carencias de las demostraciones nos dirigen hacia la creación de viñetas, las cuales actúan a modo de artículos, concentrando en un único documento texto, código y las salidas que este produce (incluyendo gráficos y tablas).

- Las viñetas se almacenan en el directorio `./vignettes/`.
- Se trata de documentos `rmarkdown`, con los que ya estamos familiarizados, pero incorporan los siguientes metadatos:

```
---
title: "Título"
author: "Autor"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

- a día de hoy, el formato más habitual para las viñetas HTML.
- Recuerda añadir posibles dependencias de las viñetas en el campo `Suggest` del archivo *DESCRIPTION* (como `knitr` y `rmarkdown`, entre otras).

Uso de `devtools`

Nuevamente, `devtools` nos ofrece funciones que nos facilitan la vida, se trata de `devtools::use_vignette()` y `devtools::build_vignettes()`.

Se acerca el final (winter is coming)

Últimos pasos

Por ahora hemos discutido aspectos del desarrollo de paquetes aunque hemos hablado muy poco acerca del proceso a seguir para comprobar que las rutinas que creamos son funcionales.

Últimos pasos

Por ahora hemos discutido aspectos del desarrollo de paquetes aunque hemos hablado muy poco acerca del proceso a seguir para comprobar que las rutinas que creamos son funcionales.

- Mientras escribimos una nueva función es conveniente utilizar `devtools::load_all()`, generando una instalación temporal del paquete y que este se adjunte a la jerarquía de entornos.

Últimos pasos

Por ahora hemos discutido aspectos del desarrollo de paquetes aunque hemos hablado muy poco acerca del proceso a seguir para comprobar que las rutinas que creamos son funcionales.

- Mientras escribimos una nueva función es conveniente utilizar `devtools::load_all()`, generando una instalación temporal del paquete y que este se adjunte a la jerarquía de entornos.

Una vez estemos contentos con el comportamiento de nuestras funciones, ha llegado el momento de comprobar si se puede convertir en un paquete e instalarse.

Últimos pasos

Por ahora hemos discutido aspectos del desarrollo de paquetes aunque hemos hablado muy poco acerca del proceso a seguir para comprobar que las rutinas que creamos son funcionales.

- Mientras escribimos una nueva función es conveniente utilizar `devtools::load_all()`, generando una instalación temporal del paquete y que este se adjunte a la jerarquía de entornos.

Una vez estemos contentos con el comportamiento de nuestras funciones, ha llegado el momento de comprobar si se puede convertir en un paquete e instalarse.

1. La comprobación se realiza con `devtools::check()`, que detallará si se producen errores, avisos o notas durante la misma.

Últimos pasos

Por ahora hemos discutido aspectos del desarrollo de paquetes aunque hemos hablado muy poco acerca del proceso a seguir para comprobar que las rutinas que creamos son funcionales.

- Mientras escribimos una nueva función es conveniente utilizar `devtools::load_all()`, generando una instalación temporal del paquete y que este se adjunte a la jerarquía de entornos.

Una vez estemos contentos con el comportamiento de nuestras funciones, ha llegado el momento de comprobar si se puede convertir en un paquete e instalarse.

1. La comprobación se realiza con `devtools::check()`, que detallará si se producen errores, avisos o notas durante la misma.
2. La construcción del paquete se realiza con `devtools::build()`, que construirá la versión *bundled* del paquete (un archivo `.tar.gz`, u opcionalmente un binario) que cualquiera podrá instalar.

Consejos y utilidades

Consejos

1. Idealmente, es preferible que un paquete centre el foco en un problema y lo resuelva satisfactoriamente, a que se difumine en multitud de problemas y abordajes.

Consejos

1. Idealmente, es preferible que un paquete centre el foco en un problema y lo resuelva satisfactoriamente, a que se difumine en multitud de problemas y abordajes.
2. Crea un archivo README en el directorio raíz donde describas un poco el paquete (hablaremos de esto en la sesión de Git-GitHub).
    - Será lo primero que vean potenciales desarrolladores que te ayuden en tu empeño.

Consejos

1. Idealmente, es preferible que un paquete centre el foco en un problema y lo resuelva satisfactoriamente, a que se difumine en multitud de problemas y abordajes.
2. Crea un archivo README en el directorio raíz donde describas un poco el paquete (hablaremos de esto en la sesión de Git-GitHub).
    - Será lo primero que vean potenciales desarrolladores que te ayuden en tu empeño.
3. Documenta, adjunta unos datos de prueba (o vincula a un paquete que los aporte), crea pruebas y no desesperes por el camino.
    - A nivel personal, el que un paquete disponga de viñetas es uno de los motivos que me animan a probarlo.

Consejos

1. Idealmente, es preferible que un paquete centre el foco en un problema y lo resuelva satisfactoriamente, a que se difumine en multitud de problemas y abordajes.
2. Crea un archivo README en el directorio raíz donde describas un poco el paquete (hablaremos de esto en la sesión de Git-GitHub).
    - Será lo primero que vean potenciales desarrolladores que te ayuden en tu empeño.
3. Documenta, adjunta unos datos de prueba (o vincula a un paquete que los aporte), crea pruebas y no desesperes por el camino.
    - A nivel personal, el que un paquete disponga de viñetas es uno de los motivos que me animan a probarlo.
4. No te preocupes en fases tempranas por el rendimiento o la velocidad de ejecución de tu código: ya llegará el momento de hacerlo.
    - Mientras las funciones devuelvan resultados, todo irá bien.

Consejos

1. Idealmente, es preferible que un paquete centre el foco en un problema y lo resuelva satisfactoriamente, a que se difumine en multitud de problemas y abordajes.

2. Crea un archivo README en el directorio raíz donde describas un poco el paquete (hablaremos de esto en la sesión de Git-GitHub).
   - Será lo primero que vean potenciales desarrolladores que te ayuden en tu empeño.

3. Documenta, adjunta unos datos de prueba (o vincula a un paquete que los aporte), crea pruebas y no desesperes por el camino.
   - A nivel personal, el que un paquete disponga de viñetas es uno de los motivos que me animan a probarlo.

4. No te preocupes en fases tempranas por el rendimiento o la velocidad de ejecución de tu código: ya llegará el momento de hacerlo.
   - Mientras las funciones devuelvan resultados, todo irá bien.

5. Empápate de pereza y evita editar a mano aquellos archivos que pueden editarse de forma automática (gracias `roxygen2` o `devtools`).
   - En general, DRY (salvo en el caso de documentación o pruebas).

Comandos útiles en el desarrollo de paquetes

```r
devtools::create()

devtools::use_mit_license() # devtools::use_gpl3_license()

devtools::use_package(type = "Imports") # devtools::use_package(type = "Suggest")

utils::file.edit()

devtools::load_all()

devtools::use_readme_rmd()

devtools::use_data() # devtools::use_data_raw()

devtools::document()

devtools::use_testthat()

devtools::use_vignette() # devtools::build_vignettes()

devtools::check()

devtools::build()
```

Muchas gracias por la atención

Referencias bibliográficas

Referencias bibliográficas (1/1)

Wickham, H. (2015a). *Advanced R*. Boca Raton, FL: CRC.
Wickham, H. (2015b). *R Packages*. Sebastopol, CA: O'Reilly.