semana 3

Luis Ambrocio

22/8/2021

r packages

¿Qué es un paquete R?

- Un mecanismo para ampliar la funcionalidad básica de R
- Una colección de funciones R u otros objetos (de datos)
- Organizado de manera sistemática para proporcionar una cantidad mínima de consistencia
- Escrito por usuarios/desarrolladores en todas partes.

donde estan los paquetes

- Disponible principalmente en CRAN y Bioconductor
- También disponible en GitHub, Bitbucket, Gitorious, etc. (y en otros lugares)
- Los paquetes de CRAN/Bioconductor se pueden instalar con `install.packages ()`
- Los paquetes de GitHub se pueden instalar usando `install_github ()` desde el paquete *devtools*

No es necesario que coloque un paquete en un repositorio central, pero al hacerlo facilita que otros instalen su paquete.

¿Cuál es el punto?

- "¿Por qué no poner un código disponible?"
- Documentación/viñetas
- Recursos centralizados como CRAN
- Estándares mínimos de confiabilidad y robustez
- Mantenibilidad/extensión
- Definición de interfaz/API clara
- Los usuarios saben que al menos se cargará correctamente.

Proceso de desarrollo de paquetes

- Escriba un código en un archivo de script R (.R)
- Quiere que el código esté disponible para otros
- Incorporar el archivo de script R en la estructura del paquete R
- Escribir documentación para funciones de usuario
- Incluya algún otro material (ejemplos, demostraciones, conjuntos de datos, tutoriales)
- ¡Empacalo!
- Envíe el paquete a CRAN o Bioconductor
- Envíe el repositorio de código fuente a GitHub u otro sitio web para compartir código fuente
- La gente encuentra todo tipo de problemas con tu código.
  - Escenario n. ° 1: le informan sobre esos problemas y esperan que los solucione
  - Escenario n. ° 2: solucionan el problema por usted y le muestran los cambios
- Incorporas los cambios y lanzas una nueva versión

Conceptos básicos del paquete R

- Un paquete R se inicia creando un directorio con el nombre del paquete R
- Un archivo DESCRIPTION que tiene información sobre el paquete.
- ¡Código R! (en el subdirectorio R /)
- Documentación (en el subdirectorio man)
- ESPACIO DE NOMBRES
- Requisitos completos para escribir extensiones R

El archivo DESCRIPTION

- *Package*: nombre del paquete (por ejemplo, biblioteca (nombre))
- *Title*: nombre completo del paquete
- *Description*: descripción más larga del paquete en una frase (normalmente)
- *Version*: número de versión (normalmente formato M.m-p)
- *Author*, *Authors@R*: nombre de los autores originales
- *Maintainer*: nombre y correo electrónico de la persona que soluciona los problemas
- *License*: Licencia para el código fuente

Estos campos son opcionales pero de uso común

- *Depends*: paquetes R de los que depende su paquete
- *Suggests*: paquetes R opcionales que los usuarios pueden querer tener instalados
- *Date*: fecha de lanzamiento en formato AAAA-MM-DD
- *URL*: página de inicio del paquete
- Se pueden agregar *Otros* campos

ejemplo creando paquete gpclib

Package: gpclib *Title*: Biblioteca general de recortes de polígonos para R *Description*: rutinas generales de recorte de polígonos para R basadas en la biblioteca C de Alan Murta. *Versión*: 1.5-5 *Autor*: Roger D. Peng rpeng@jhsph.edu con contribuciones de Duncan Murdoch y Barry Rowlingson; Biblioteca de GPC de Alan Murta *Maintainer*: Roger D. Peng rpeng@jhsph.edu *License*: LICENCIA de archivo *Depends*: R (> = 2.14.0),methods *Imports*: gráficos *Date*: 2013-04-01 *URL*: http://www.cs.man.ac.uk/~toby/gpc/, http://github.com/rdpeng/gpclib

Código R

- Copie el código R en el subdirectorio R/
- Puede haber cualquier número de archivos en este directorio.
- Por lo general, separe los archivos en grupos lógicos
- El código para todas las funciones debe incluirse aquí y no en ningún otro lugar del paquete.

El archivo NAMESPACE

- Se utiliza para indicar qué funciones se *exportan*.
- Las funciones exportadas pueden ser invocadas por el usuario y se consideran la API pública.
- El usuario no puede llamar directamente a las funciones no exportadas (pero se puede ver el código)
- Oculta los detalles de implementación de los usuarios y crea una interfaz de paquete más limpia
- También puede indicar qué funciones *importa* de otros paquetes
- Esto permite que su paquete use otros paquetes sin hacer que otros paquetes sean visibles para el usuario.
- Importar una función carga el paquete pero no lo adjunta a la lista de búsqueda.

Directivas clave

- export("<function>")
- import("<package>")
- importFrom("<package>", "<function>")

tambien es importante

- exportClasses("<class>")
- exportMethods("<generic>")

ejemplo con `mvtsplot` package

```
export("mvtsplot")
import(splines)
import(RColorBrewer)
importFrom("grDevices", "colorRampPalette", "gray")
importFrom("graphics", "abline", "axis", "box", "image",
           "layout", "lines", "par", "plot", "points",
           "segments", "strwidth", "text", "Axis")
importFrom("stats", "complete.cases", "lm", "na.exclude",
           "predict", "quantile")
```

ejemplo con `gpclib` package

```
export("read.polyfile", "write.polyfile")

importFrom(graphics, plot)

exportClasses("gpc.poly", "gpc.poly.nohole")

exportMethods("show", "get.bbox", "plot", "intersect", "union",
              "setdiff", "[", "append.poly", "scale.poly",
              "area.poly", "get.pts", "coerce", "tristrip",
              "triangulate")
```

Documentación

- Archivos de documentación (.Rd) colocados en man/subdirectorio
- Escrito en un lenguaje de marcado específico.
- Requerido para cada función exportada
  - Otra razón para limitar las funciones exportadas
- Puede documentar otras cosas como conceptos, descripción general del paquete

ejemplo con `line` Function

```
\name{line}
\alias{line}
\alias{residuals.tukeyline}
\title{Robust Line Fitting}
\description{
  Fit a line robustly as recommended in \emph{Exploratory Data Analysis}.
}

\usage{
line(x, y)
}
\arguments{
  \item{x, y}{the arguments can be any way of specifying x-y pairs.  See
    \code{\link{xy.coords}}.}
}

\details{
```

```
  Cases with missing values are omitted.

  Long vectors are not supported.
}
\value{
  An object of class \code{"tukeyline"}.

  Methods are available for the generic functions \code{coef},
  \code{residuals}, \code{fitted}, and \code{print}.
}

\references{
  Tukey, J. W. (1977).
  \emph{Exploratory Data Analysis},
  Reading Massachusetts: Addison-Wesley.
}
```

Construcción y verificación

- R CMD build es un programa de línea de comandos que crea un archivo de paquete (`.tar.gz`)
- R CMD check ejecuta una batería de pruebas en el paquete
- Puede ejecutar la compilación de R CMD o la comprobación de R CMD desde la línea de comandos utilizando una aplicación de terminal o shell de comandos
- También puede ejecutarlos desde R usando la función system()

```
system("R CMD build newpackage")
system("R CMD check newpackage")
```

comprobación

- R CMD check ejecuta una prueba de batería
- Existe documentación
- El código se puede cargar, sin problemas o errores importantes de codificación
- Ejecutar ejemplos en la documentación
- Verifique el código de coincidencia de documentos
- Todas las pruebas deben pasar para poner el paquete en CRAN

Empezando

- La función `package.skeleton ()` en el paquete utils crea un paquete R "esqueleto"
- Estructura de directorio (R/,man/), archivo DESCRIPTION, archivo NAMESPACE, archivos de documentación
- Si hay funciones visibles en su espacio de trabajo, escribe archivos de código R en el directorio R/
- Los talones de documentación se crean en man/
- ¡Tienes que completar el resto!

Resumen

- Los paquetes R proporcionan una forma sistemática de hacer que el código R esté disponible para otros
- Los estándares garantizan que los paquetes tengan una cantidad mínima de documentación y solidez.
- Obtenido de CRAN, Bioconductor, Github, etc.
- Cree un nuevo directorio con los subdirectorios R/y man/(o simplemente use package.skeleton ())
- Escribir un archivo DESCRIPTION
- Copie el código R en el subdirectorio R /
- Escribir archivos de documentación en man/subdirectorio

- Escriba un archivo NAMESPACE con exportaciones/importaciones
- Construir y verificar

Clases y métodos

- Un sistema para realizar programación orientada a objetos
- R originalmente era bastante interesante porque es interactivo *y* tiene un sistema para la orientación de objetos.
  - Otros lenguajes que admiten OOP (C ++, Java, Lisp, Python, Perl) en general no son lenguajes interactivos
- En R, gran parte del código para las clases/métodos de apoyo está escrito por el propio John Chambers (el creador del lenguaje S original) y documentado en el libro *Programming with Data: A Guide to the S Language*
- Una extensión natural de la idea de Chambers de permitir que alguien cruce al usuario -> espectro del programador
- La programación orientada a objetos es un poco diferente en R que en la mayoría de los lenguajes - incluso si está familiarizado con la idea, es posible que desee prestar atención a los detalles

Dos estilos de clases y métodos

- Clases/métodos S3
- Incluido con la versión 3 del lenguaje S.
- Informal, un poco torpe
- A veces llamadas clases/métodos *old-style*
- Clases/métodos S4
- más formal y riguroso
- Incluido con S-PLUS 6 y R 1.4.0 (diciembre de 2001)
- También llamado clases/métodos *new-style*

Dos mundos viviendo uno al lado del otro

- Por ahora (y en el futuro previsible), las clases/métodos S3 y las clases/métodos S4 son sistemas separados (pero se pueden mezclar hasta cierto punto).
- Cada sistema se puede utilizar de forma bastante independiente del otro.
- Se anima a los desarrolladores de nuevos proyectos (¡usted!) A utilizar las clases/métodos de estilo S4.
  - Utilizado ampliamente en el proyecto Bioconductor.
- Pero muchos desarrolladores todavía usan clases/métodos S3 porque son "rápidos y sucios" (y más fáciles).
- En esta conferencia nos centraremos principalmente en las clases/métodos de S4
- El código para implementar clases/métodos S4 en R está en el paquete *methods*, que generalmente se carga por defecto (pero puede cargarlosi por alguna razón no está cargado)

Programación orientada a objetos en R

- Una clase es una descripción de una cosa. Se puede definir una clase usando `setClass()` en el paquete*methods*.
- Un objeto es una instancia de una clase. Los objetos se pueden crear usando `new()`.
- Un *método* es una función que solo opera en una determinada clase de objetos.
- Una función genérica es una función R que distribuye métodos. Una función genérica generalmente encapsula un concepto "genérico" (p. Ej., `plot`,`mean`, `predict`, ...)

  - La función genérica en realidad no realiza ningún cálculo.

- Un *metodo* es la implementación de una función genérica para un objeto de una clase particular.
- Los archivos de ayuda para el paquete 'métodos' son extensos - léalos ya que son la documentación principal
- Es posible que desee comenzar con `?classes` y`?methods`
- Echa un vistazo a `?SetClass`,`?SetMethod` y `?SetGeneric`
- Algo se vuelve técnico, pero haz tu mejor esfuerzo por ahora; tendrá sentido en el futuro a medida que lo sigas usando.
- La mayor parte de la documentación en el paquete *methods* está orientada a desarrolladores/programadores, ya que estas son las personas principales que utilizan clases/métodos.

Classes

todos los objetos en R tienen una clase

```
class(1)
```

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

```
class(TRUE)
```

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

```
class(rnorm(100))
```

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

```
class(NA)
```

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

```
class("foo")
```

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

Las clases de datos van más allá de las clases atómicas

```
x <- rnorm(100)
y <- x + rnorm(100)
fit <- lm(y ~ x)  ## linear regression model
class(fit)
```

```
## [1] "lm"
```

Genéricos/Métodos en R

- Las funciones genéricas de estilo S4 y S3 se ven diferentes pero conceptualmente son las mismas (juegan el mismo papel).
- Cuando programe, puede escribir nuevos métodos para un genérico existente O crear sus propios genéricos y métodos asociados.
- Por supuesto, si no existe un tipo de datos en R que coincida con sus necesidades, siempre puede definir una nueva clase junto con genéricos/métodos que la acompañen.

Una función genérica de S3 (en el paquete 'base')

Las funciones `mean` e`print` son genéricas

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x00000000151c7e70>
## <environment: namespace:base>
```

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x0000000012cb3828>
## <environment: namespace:base>
```

La función genérica `mean` tiene varios métodos asociados.

```
methods("mean")
```

```
## [1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
## [6] mean.quosure*
## see '?methods' for accessing help and source code
```

Una función genérica de S4

La función `show` es del paquete `methods` y es el equivalente en s4 de print

```
library(methods)
show
```

```
## standardGeneric for "show" defined from package "methods"
##
## function (object)
## standardGeneric("show")
## <bytecode: 0x00000000147d4290>
## <environment: 0x0000000013a12580>
## Methods may be defined for arguments: object
## Use  showMethods(show)  for currently available ones.
## (This generic function excludes non-simple inheritance; see ?setIs)
```

La función `show` generalmente no se llama directamente (al igual que`print`) porque los objetos se imprimen automáticamente.

```
showMethods("show")
```

```
## Function: show (package methods)
## object="ANY"
## object="classGeneratorFunction"
## object="classRepresentation"
## object="envRefClass"
## object="externalRefMethod"
## object="function"
##     (inherited from: object="ANY")
## object="genericFunction"
## object="genericFunctionWithTrace"
## object="MethodDefinition"
## object="MethodDefinitionWithTrace"
## object="MethodSelectionReport"
## object="MethodWithNext"
## object="MethodWithNextWithTrace"
```

```
## object="namedList"
## object="ObjectsWithPackage"
## object="oldClass"
## object="refClassRepresentation"
## object="refMethodDef"
## object="refObjectGenerator"
## object="signature"
## object="sourceEnvironment"
## object="standardGeneric"
##     (inherited from: object="genericFunction")
## object="traceable"
```

Mecanismo genérico/de método

El primer argumento de una función genérica es un objeto de una clase particular (puede haber otros argumentos)

1. La función genérica verifica la clase del objeto.
2. Se realiza una búsqueda para ver si existe un método apropiado para esa clase.
3. Si existe un método para esa clase, entonces se llama a ese método en el objeto y listo.
4. Si no existe un método para esa clase, se realiza una búsqueda para ver si hay un método predeterminado para el genérico. Si existe un valor predeterminado, se llama al método predeterminado.
5. Si no existe un método predeterminado, se genera un error.

Examen de código para métodos

- No puede simplemente imprimir el código de un método como otras funciones porque el código del método generalmente está oculto.
- Si desea ver el código de un método S3, puede utilizar la función `getS3method`.
- La llamada es `getS3method (<generic>, <class>)`
- Para los métodos S4, puede usar la función `getMethod`
- La llamada es `getMethod (<generic>, <signature>)` (más detalles más adelante)

S3 Class/Method:

ejemplo 1

¿Qué está pasando aquí?

```
set.seed(2)
x <- rnorm(100)
mean(x)
```

```
## [1] -0.03069816
```

1. La clase de x es “numérica”
2. ¡Pero no existe un método mean para los objetos “numéricos”!
3. Así que llamamos a la función predeterminada para “mean”.

```
head(getS3method("mean", "default"), 10)
```

```
##
## 1  function (x, trim = 0, na.rm = FALSE, ...)
## 2  {
## 3      if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
## 4          warning("argument is not numeric or logical: returning NA")
## 5          return(NA_real_)
## 6      }
```

```
## 7      if (na.rm)
## 8          x <- x[!is.na(x)]
## 9      if (!is.numeric(trim) || length(trim) != 1L)
## 10         stop("'trim' must be numeric of length one")
```

```
tail(getS3method("mean", "default"), 10)
```

```
##
## 15         if (anyNA(x))
## 16             return(NA_real_)
## 17         if (trim >= 0.5)
## 18             return(stats::median(x, na.rm = FALSE))
## 19         lo <- floor(n * trim) + 1
## 20         hi <- n + 1 - lo
## 21         x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
## 22     }
## 23     .Internal(mean(x))
## 24 }
```

ejemplo 2

¿Qué pasa aquí?

```
set.seed(3)
df <- data.frame(x = rnorm(100), y = 1:100)
sapply(df, mean)
```

```
##           x           y
##  0.01103557 50.50000000
```

1. La clase de `df` es" data.frame "; cada columna puede ser un objeto de una clase diferente
2. Hacemos "sapply" sobre las columnas y llamamos a la función "mean"
3. En cada columna, `mean` verifica la clase del objeto y envía el método apropiado.
4. Tenemos una columna "numérica" y una columna "entero"; `mean` llama al método predeterminado para ambos

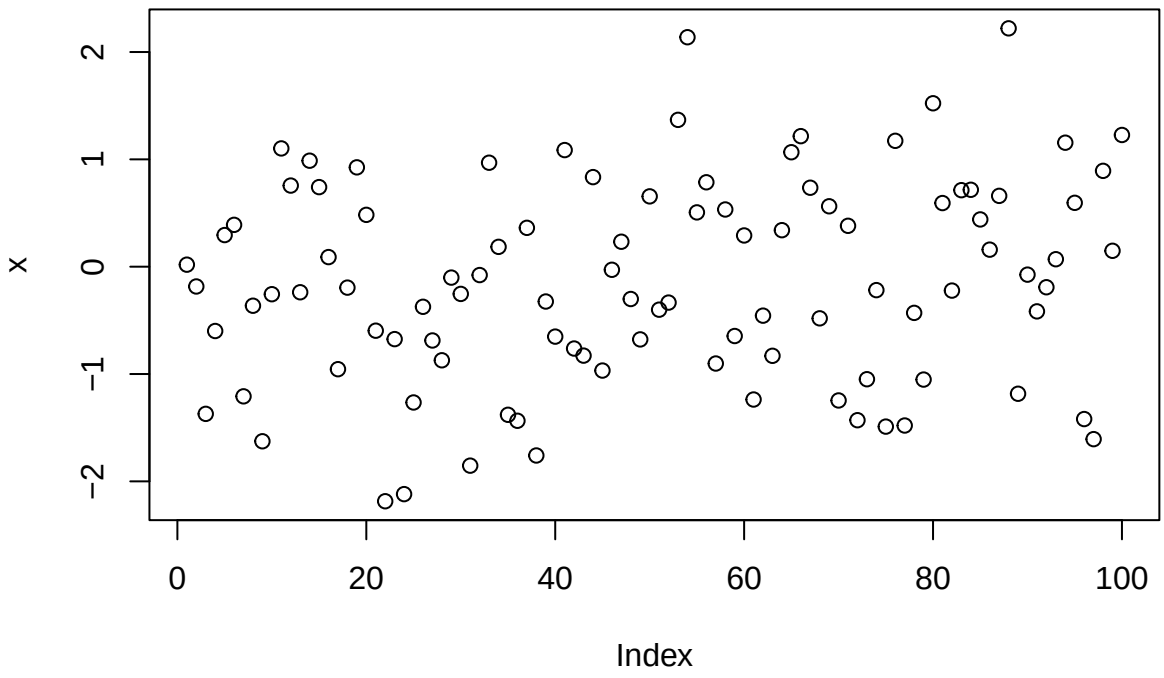
Métodos de llamada directamente

- Algunos métodos de S3 son visibles para el usuario (es decir, "mean.default"),
- *Nunca* llamar a métodos directamente
- Utilice la función genérica y deje que el método se envíe automáticamente.
- Con los métodos S4 no puede llamarlos directamente en absoluto

ejemplo 3

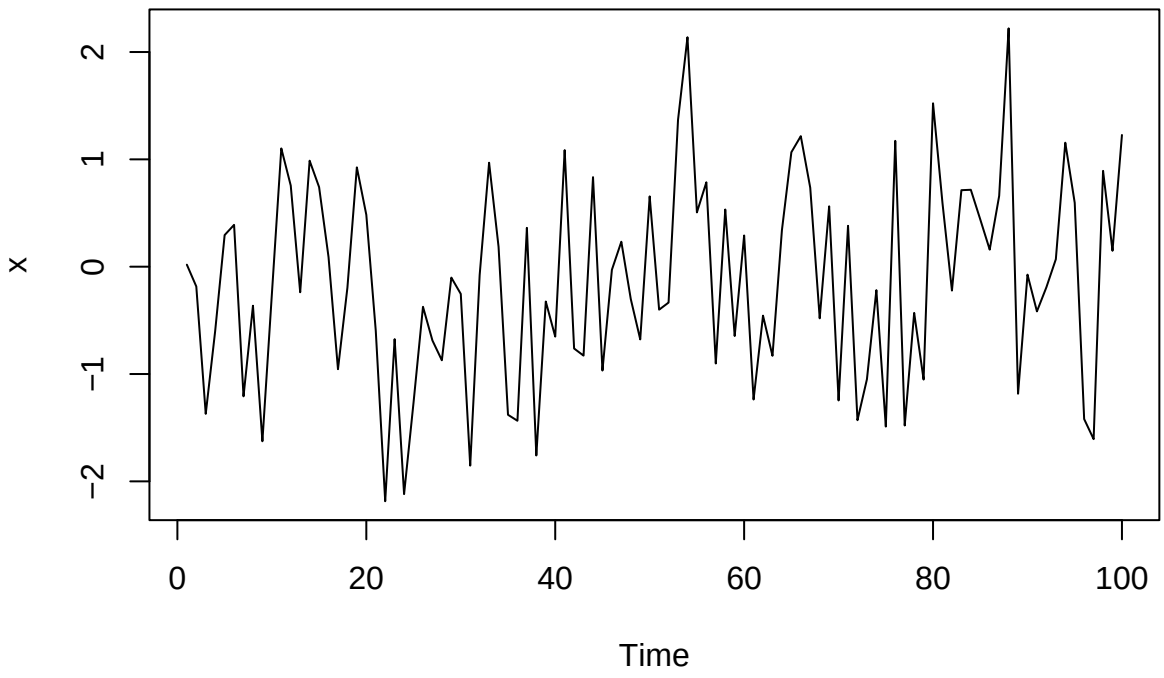
La función `plot` es genérica y su comportamiento depende del objeto que se traza.

```
set.seed(10)
x <- rnorm(100)
plot(x)
```



para series de tiempo `plot` conecta los puntos

```
set.seed(10)
x <- rnorm(100)
x <- as.ts(x) ## Convert to a time series object
plot(x)
```



¡Escriba sus propios métodos!

Si escribe nuevos métodos para nuevas clases, probablemente terminará escribiendo métodos para los siguientes genéricos: - print/show - summary - plot

Hay dos formas de extender el sistema R a través de clases/métodos - Escriba un método para una nueva clase pero para una función genérica existente (es decir, como `print`) - Escribe nuevas funciones genéricas y nuevos métodos para esos genéricos.

S4 Class/Method

¿Por qué querrías crear una nueva clase? - Para representar nuevos tipos de datos (por ejemplo, expresión genética, espacio-tiempo, matrices jerárquicas, dispersas) - Nuevos conceptos/ideas que aún no se han pensado (por ejemplo, un modelo de proceso de puntos ajustados, un modelo de efectos mixtos, una matriz dispersa) - Para abstraer/ocultar los detalles de implementación del usuario Digo que las cosas son "nuevas", lo que significa que R no las conoce (no es que sean nuevas para la comunidad estadística).

Se puede definir una nueva clase usando la función `setClass` - Como mínimo, debe especificar el nombre de la clase. - También puede especificar elementos de datos que se denominan *slots* - Luego puede definir métodos para la clase con la función `setMethod` - Se puede obtener información sobre la definición de una clase con la función `showClass`

La creación de nuevas clases/métodos generalmente no se hace en la consola; es probable que desee guardar el código en un archivo separado

```
library(methods)
setClass("polygon",
         representation(x = "numeric",
                        y = "numeric"))
```

- Los espacios para esta clase son `x` e`y`
- Se puede acceder a las ranuras para un objeto S4 con el operador "@".

Se puede crear un método de trazado con la función `setMethod`.

- Para `setMethod` necesitas especificar una función genérica (`plot`), y una *signature*.
- Una firma es un vector de caracteres que indica las clases de objetos que acepta el método.
- En este caso, el método `plot` tomará un tipo de objeto, un objeto `polygon`.

Creando un método `plot` con`setMethod`.

```
setMethod("plot", "polygon",
          function(x, y, ...) {
                  plot(x@x, x@y, type = "n", ...)
                  xp <- c(x@x, x@x[1])
                  yp <- c(x@y, x@y[1])
                  lines(xp, yp)
      })
```

- Observe que se accede a las ranuras del polígono (las coordenadas x e y) con el operador "@".

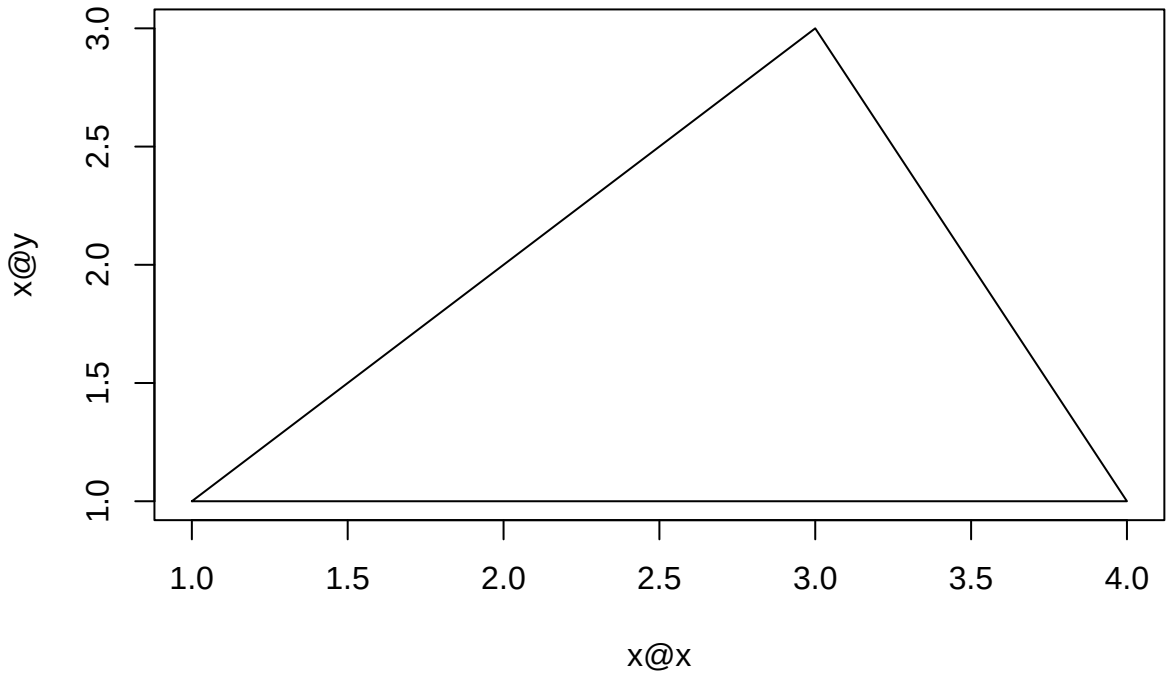
Después de llamar a `setMethod`, el nuevo método`plot` se agregará a la lista de métodos para `plot`.

```
library(methods)
showMethods("plot")
```

```
## Function: plot (package base)
## x="ANY"
## x="polygon"
```

Observe que se incluye la signature de la clase "polígono". El método para `ANY` es el método predeterminado y es lo que se llama cuando ahora coinciden otras firmas

```
p <- new("polygon", x = c(1, 2, 3, 4), y = c(1, 2, 3, 1))
plot(p)
```



Resumen

- El desarrollo de clases y métodos asociados es una forma poderosa de ampliar la funcionalidad de R
- Las **clases** definen nuevos tipos de datos.
- Los **métodos** amplían las **funciones genéricas** para especificar el comportamiento de funciones genéricas en nuevas clases.
- A medida que se crean nuevos tipos de datos y conceptos, las clases/métodos proporcionan una forma de desarrollar una interfaz intuitiva para esos datos/conceptos para los usuarios.

Dónde buscar, lugares para comenzar

- La mejor manera de aprender estas cosas es mirar ejemplos.
- Hay bastantes ejemplos en CRAN que usan clases/métodos S4. Por lo general, puede saber si usan clases/métodos S4 si el paquete **methods** aparece en el campo `Depends:`
- Bioconductor (http://www.bioconductor.org): un recurso rico, incluso si no sabe nada sobre bioinformática
- Algunos paquetes en CRAN (que yo sepa) - SparseM, gpclib, flexmix, its, lme4, orientlib, filehash
- El paquete `stats4` (viene con R) tiene un montón de clases/métodos para realizar análisis de máxima verosimilitud.