# 01: R Packages

**What is an R Package?**

* A mechanism for extending the basic functionality of R
* A collection of R functions, or other (data) objects
* Organized in a systematic fashion to provide a minimal amount of consistency
* Written by users/developers everywhere

**Where are These R Packages?**

* Primarily available from CRAN and Bioconductor
* Also available from GitHub, Bitbucket, Gitorious, etc. (and elsewhere)
* Packages from CRAN/Bioconductor can be installed with install.packages()
* Packages from GitHub can be installed using install\_github() from the **devtools** package

You do not have to put a package on a central repository, but doing so makes it easier for others to install your package.

**What's the Point?**

* "Why not just make some code available?"
* Documentation / vignettes
* Centralized resources like CRAN
* Minimal standards for reliability and robustness
* Maintainability / extension
* Interface definition / clear API
* Users know that it will at least load properly

**Package Development Process**

* Write some code in an R script file (.R)
* Want to make code available to others
* Incorporate R script file into R package structure
* Write documentation for user functions
* Include some other material (examples, demos, datasets, tutorials)
* Package it up!

**Package Development Process**

* Submit package to CRAN or Bioconductor
* Push source code repository to GitHub or other source code sharing web site
* People find all kinds of problems with your code
  + Scenario #1: They tell you about those problems and expect you to fix it
  + Scenario #2: They fix the problem for you and show you the changes
* You incorporate the changes and release a new version

**R Package Essentials**

* An R package is started by creating a directory with the name of the R package
* A DESCRIPTION file which has info about the package
* R code! (in the R/ sub-directory)
* Documentation (in the man/ sub-directory)
* NAMESPACE (optional, but do it)
* Full requirements in Writing R Extensions

**The DESCRIPTION File**

* **Package**: Name of package (e.g. library(name))
* **Title**: Full name of package
* **Description**: Longer description of package in one sentence (usually)
* **Version**: Version number (usually M.m-p format)
* **Author**, **Authors@R**: Name of the original author(s)
* **Maintainer**: Name + email of person who fixes problems
* **License**: License for the source code

**The DESCRIPTION File**

These fields are optional but commonly used

* **Depends**: R packages that your package depends on
* **Suggests**: Optional R packages that users may want to have installed
* **Date**: Release date in YYYY-MM-DD format
* **URL**: Package home page
* **Other** fields can be added

**DESCRIPTION File: gpclib**

**Package**: gpclib  
**Title**: General Polygon Clipping Library for R **Description**: General polygon clipping routines for R based on Alan Murta's C library  
**Version**: 1.5-5  
**Author**: Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu) with contributions from Duncan Murdoch and Barry Rowlingson; GPC library by Alan Murta  
**Maintainer**: Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu)  
**License**: file LICENSE  
**Depends**: R (>= 2.14.0), methods  
**Imports**: graphics  
**Date**: 2013-04-01  
**URL**: <http://www.cs.man.ac.uk/~toby/gpc/>, <http://github.com/rdpeng/gpclib>

**R Code**

* Copy R code into the R/ sub-directory
* There can be any number of files in this directory
* Usually separate out files into logical groups
* Code for all functions should be included here and not anywhere else in the package

**The NAMESPACE File**

* Used to indicate which functions are **exported**
* Exported functions can be called by the user and are considered the public API
* Non-exported functions cannot be called directly by the user (but the code can be viewed)
* Hides implementation details from users and makes a cleaner package interface

**The NAMESPACE File**

* You can also indicate what functions you **import** from other packages
* This allows for your package to use other packages without making other packages visible to the user
* Importing a function loads the package but does not attach it to the search list

**The NAMESPACE File**

Key directives

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

Also important

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

**NAMESPACE File: mvtsplot package**

export("mvtsplot")

importFrom(graphics, "Axis")

import(splines)

**NAMESPACE File: 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")

**Documentation**

* Documentation files (.Rd) placed in man/ sub-directory
* Written in a specific markup language
* Required for every exported function
  + Another reason to limit exported functions
* You can document other things like concepts, package overview

**Help File Example: 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}.

}

**Help File Example: line Function**

\usage{

line(x, y)

}

\arguments{

\item{x, y}{the arguments can be any way of specifying x-y pairs. See

\code{\link{xy.coords}}.}

}

**Help File Example: line Function**

\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}.

}

**Help File Example: line Function**

\references{

Tukey, J. W. (1977).

\emph{Exploratory Data Analysis},

Reading Massachusetts: Addison-Wesley.

}

**Building and Checking**

* R CMD build is a command-line program that creates a package archive file (.tar.gz)
* R CMD check runs a battery of tests on the package
* You can run R CMD build or R CMD check from the command-line using a terminal or command-shell application
* You can also run them from R using the system() function

system("R CMD build newpackage")

system("R CMD check newpackage")

**Checking**

* R CMD check runs a battery tests
* Documentation exists
* Code can be loaded, no major coding problems or errors
* Run examples in documentation
* Check docs match code
* All tests must pass to put package on CRAN

**Getting Started**

* The package.skeleton() function in the utils package creates a "skeleton" R package
* Directory structure (R/, man/), DESCRIPTION file, NAMESPACE file, documentation files
* If there are functions visible in your workspace, it writes R code files to the R/ directory
* Documentation stubs are created in man/
* You need to fill in the rest!

**Summary**

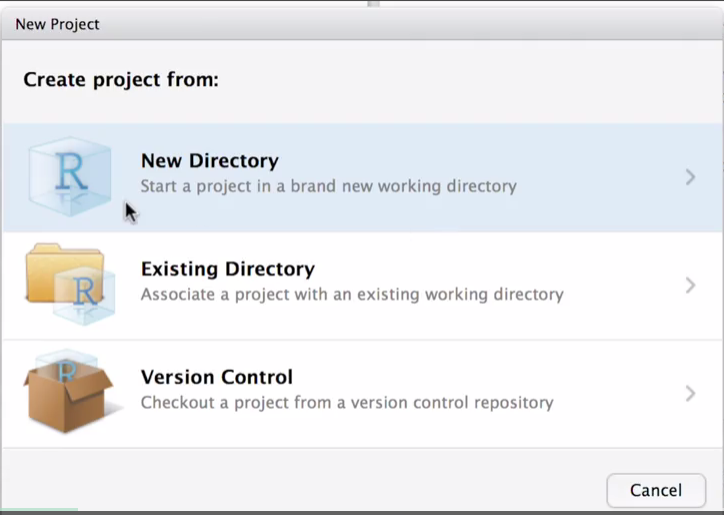
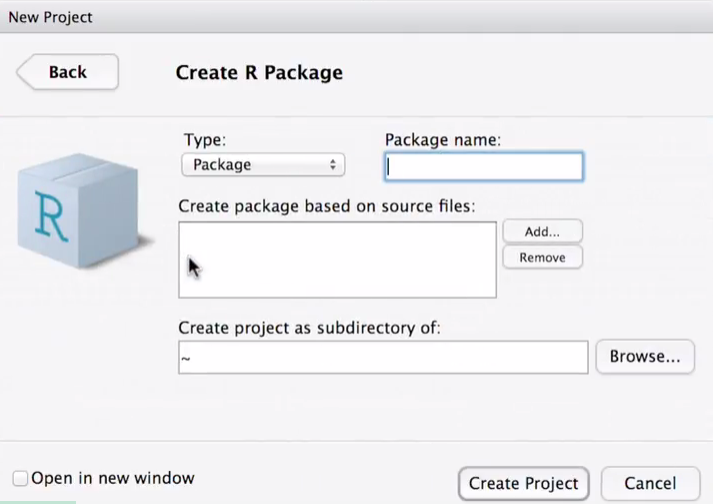
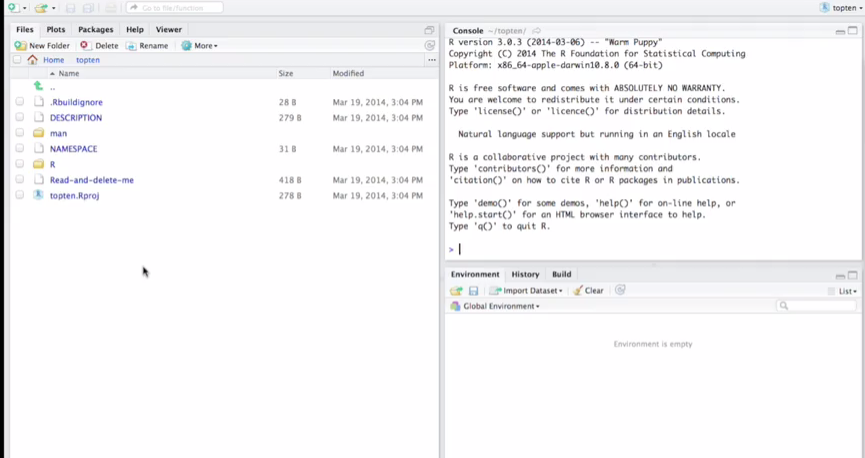
* R packages provide a systematic way to make R code available to others
* Standards ensure that packages have a minimal amount of documentation and robustness
* Obtained from CRAN, Bioconductor, Github, etc.

**Summary**

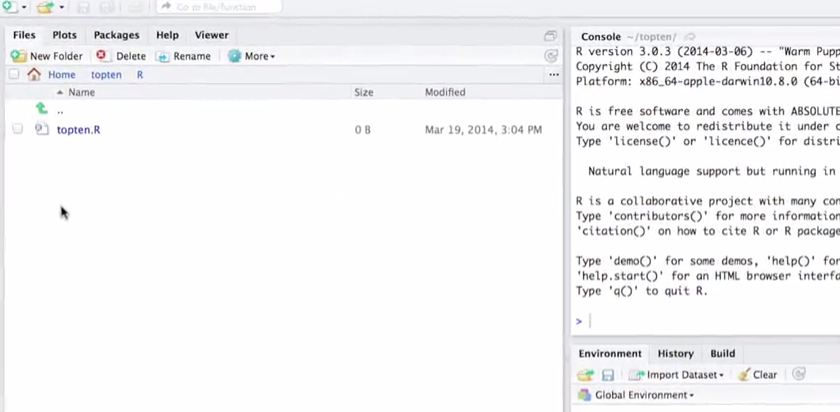
* Create a new directory with R/ and man/ sub-directories (or just use package.skeleton())
* Write a DESCRIPTION file
* Copy R code into the R/ sub-directory
* Write documentation files in man/ sub-directory
* Write a NAMESPACE file with exports/imports
* Build and check

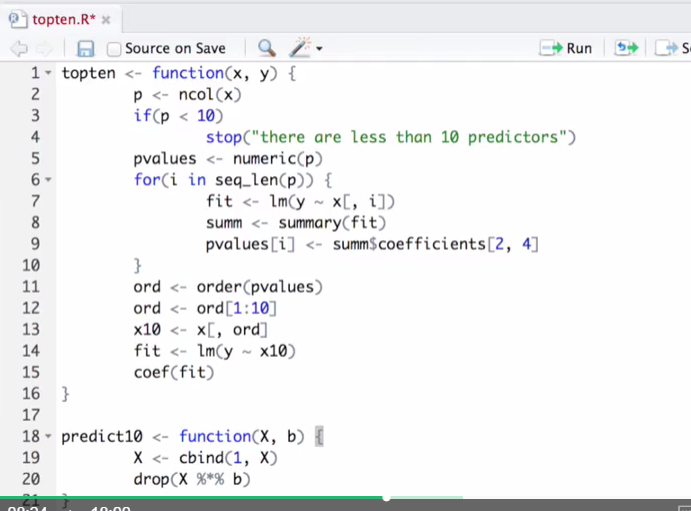
**Building R Packages Demo**

[**http://goo.gl/Ro8R3J**](http://goo.gl/Ro8R3J)

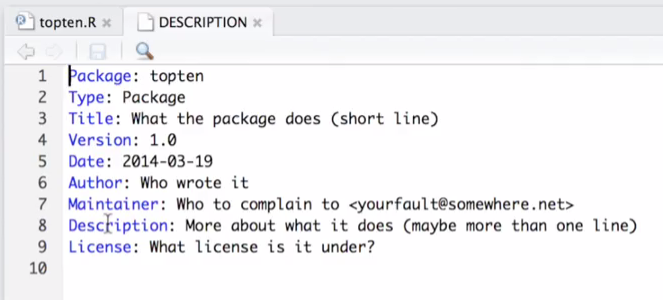
**  **

R Directory

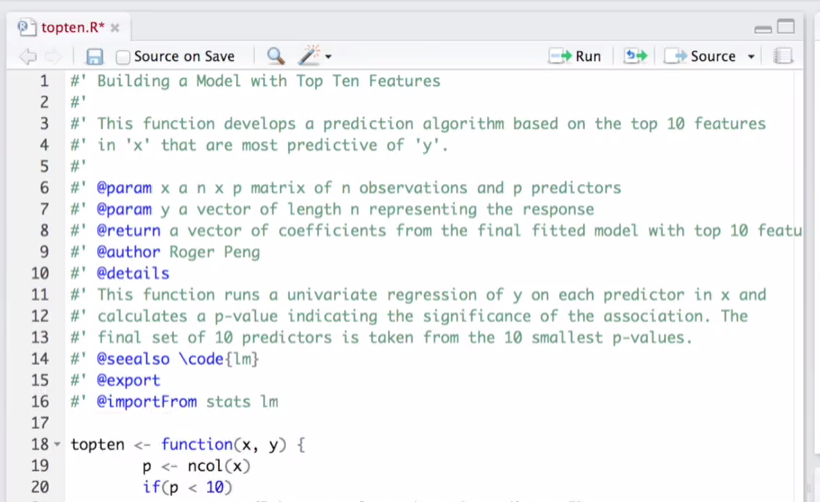
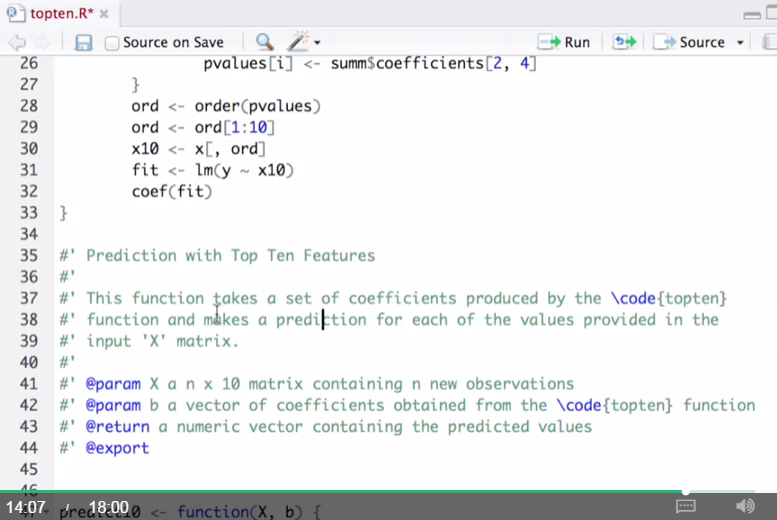




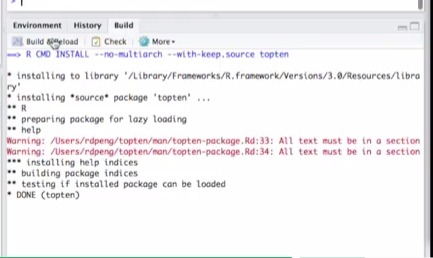
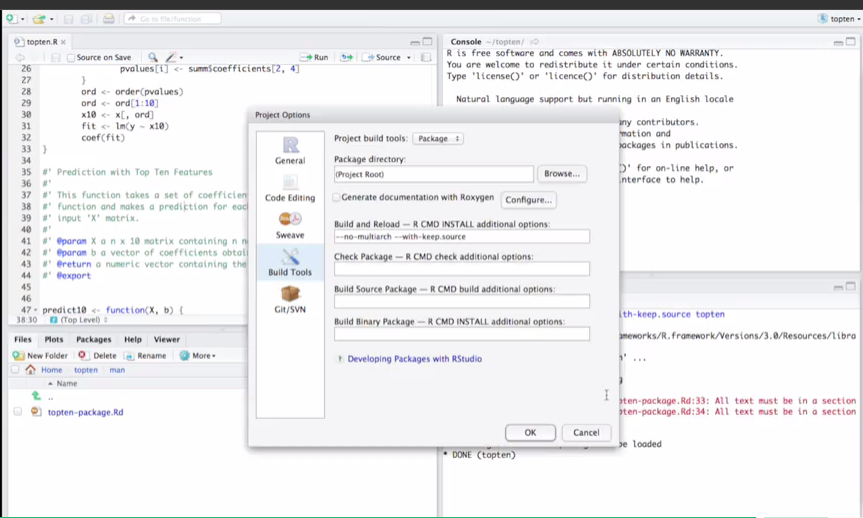
Description file



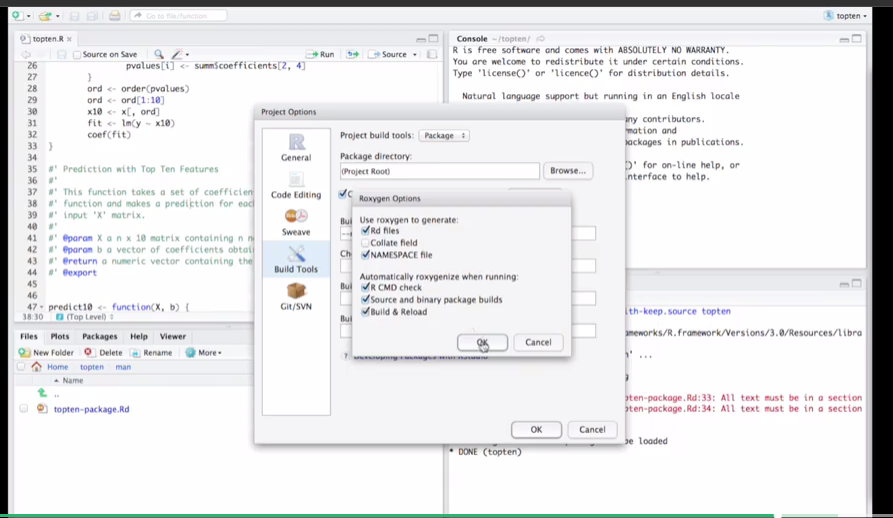
Docuementation

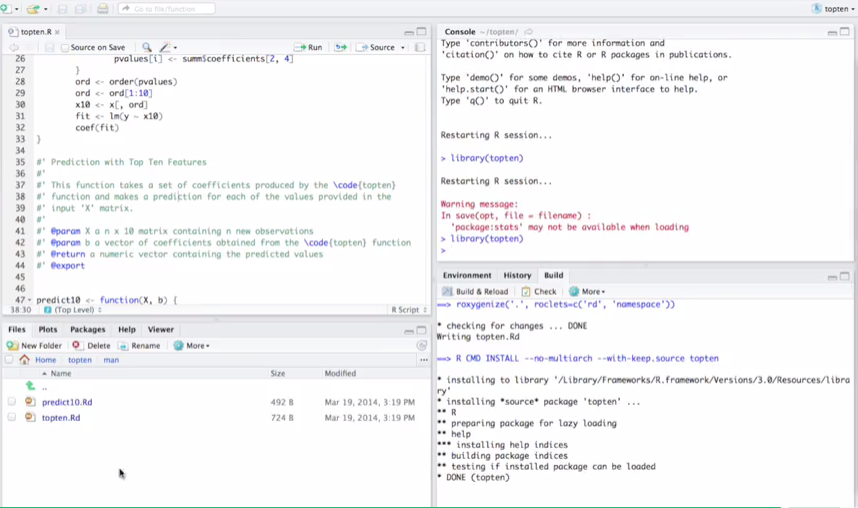
Build

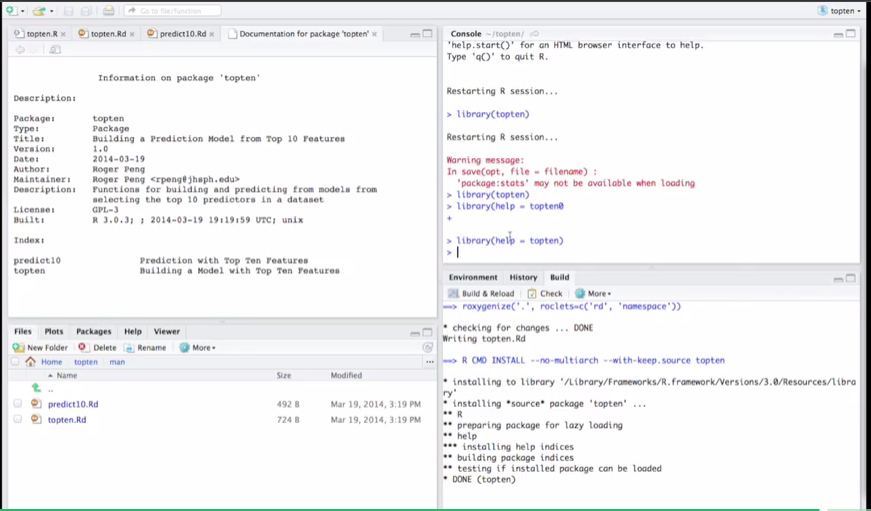
Generate Doumentation with Roxn…

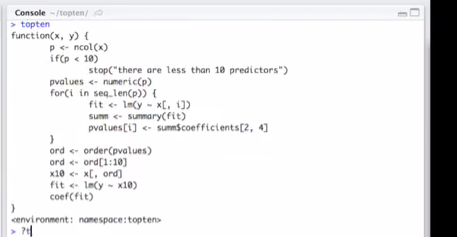


Build and reload, and 2 new documentation created. Topten.Rd file and predict10.Rd file

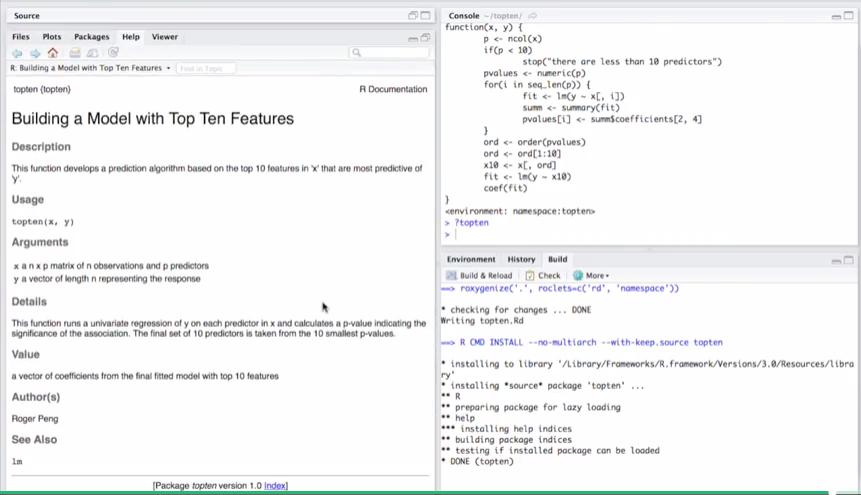


Look at the topten library (help = topten) at the console

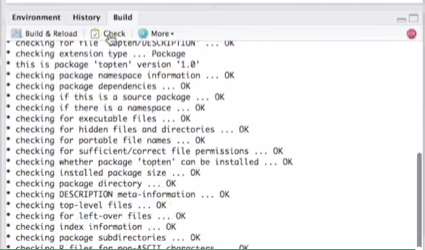


At console, to print the code, Topten 

To print the help file at console, ?topten



To check



# 02: R Classes and Methods

**Classes and Methods**

* A system for doing object oriented programming
* R was originally quite interesting because it is both interactive *and* has a system for object orientation.
  + Other languages which support OOP (C++, Java, Lisp, Python, Perl) generally speaking are not interactive languages
* In R much of the code for supporting classes/methods is written by John Chambers himself (the creator of the original S language) and documented in the book *Programming with Data: A Guide to the S Language*
* A natural extension of Chambers’ idea of allowing someone to cross the user −→ programmer spectrum
* Object oriented programming is a bit different in R than it is in most languages — even if you are familiar with the idea, you may want to pay attention to the details

**Two styles of classes and methods**

S3 classes/methods

* Included with version 3 of the S language.
* Informal, a little kludgey
* Sometimes called *old-style* classes/methods

S4 classes/methods

* more formal and rigorous
* Included with S-PLUS 6 and R 1.4.0 (December 2001)
* Also called *new-style* classes/methods

**Two worlds living side by side**

* For now (and the forseeable future), S3 classes/methods and S4 classes/methods are separate systems (but they can be mixed to some degree).
* Each system can be used fairly independently of the other.
* Developers of new projects (you!) are encouraged to use the S4 style classes/methods.
  + Used extensively in the Bioconductor project
* But many developers still use S3 classes/methods because they are “quick and dirty” (and easier).
* In this lecture we will focus primarily on S4 classes/methods
* The code for implementing S4 classes/methods in R is in the *methods* package, which is usually loaded by default (but you can load it with library(methods) if for some reason it is not loaded)

**Object Oriented Programming in R**

* A class is a description of an thing. A class can be defined using setClass() in the *methods* package.
* An *object* is an instance of a class. Objects can be created using new().
* A *method* is a function that only operates on a certain class of objects.
* A generic function is an R function which dispatches methods. A generic function typically encapsulates a “generic” concept (e.g. plot, mean, predict, ...)
  + The generic function does not actually do any computation.
* A *method* is the implementation of a generic function for an object of a particular class.

**Things to look up**

* The help files for the ‘methods’ package are extensive — do read them as they are the primary documentation
* You may want to start with ?Classes and ?Methods
* Check out ?setClass, ?setMethod, and ?setGeneric
* Some of it gets technical, but try your best for now—it will make sense in the future as you keep using it.
* Most of the documentation in the *methods* package is oriented towards developers/programmers as these are the primary people using classes/methods

**Classes**

All objects in R have a class which can be determined by the class function

class(1)

## [1] "numeric"

class(TRUE)

## [1] "logical"

**Classes**

class(rnorm(100))

## [1] "numeric"

class(NA)

## [1] "logical"

class("foo")

## [1] "character"

**Classes (cont’d)**

Data classes go beyond the atomic classes

x <- rnorm(100)

y <- x + rnorm(100)

fit <- lm(y ~ x) ## linear regression model

class(fit)

## [1] "lm"

**Generics/Methods in R**

* S4 and S3 style generic functions look different but conceptually, they are the same (they play the same role).
* When you program you can write new methods for an existing generic OR create your own generics and associated methods.
* Of course, if a data type does not exist in R that matches your needs, you can always define a new class along with generics/methods that go with it

**An S3 generic function (in the ‘base’ package)**

The mean and print functions are generic

mean

## function (x, ...)

## UseMethod("mean")

## <bytecode: 0x7facdb660ad0>

## <environment: namespace:base>

print

## function (x, ...)

## UseMethod("print")

## <bytecode: 0x7facd9ccfd58>

## <environment: namespace:base>

**S3 methods**

The mean generic function has a number of methods associated with it.

methods("mean")

## [1] mean.Date mean.default mean.difftime mean.POSIXct mean.POSIXlt

**An S4 generic function**

The show function is from the **methods** package and is the S4 equivalent of print

show

## standardGeneric for "show" defined from package "methods"

##

## function (object)

## standardGeneric("show")

## <bytecode: 0x7facdb8034d8>

## <environment: 0x7facdb779868>

## Methods may be defined for arguments: object

## Use showMethods("show") for currently available ones.

## (This generic function excludes non-simple inheritance; see ?setIs)

The show function is usually not called directly (much like print) because objects are auto-printed.

**S4 methods**

showMethods("show")

## Function: show (package methods)

## object="ANY"

## object="classGeneratorFunction"

## object="classRepresentation"

## object="envRefClass"

## 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"

**Generic/method mechanism**

The first argument of a generic function is an object of a particular class (there may be other arguments)

1. The generic function checks the class of the object.
2. A search is done to see if there is an appropriate method for that class.
3. If there exists a method for that class, then that method is called on the object and we’re done.
4. If a method for that class does not exist, a search is done to see if there is a default method for the generic. If a default exists, then the default method is called.
5. If a default method doesn’t exist, then an error is thrown.

**Examining Code for Methods**

* You cannot just print the code for a method like other functions because the code for the method is usually hidden.
* If you want to see the code for an S3 method, you can use the function getS3method.
* The call is getS3method(<generic>, <class>)
* For S4 methods you can use the function getMethod
* The call is getMethod(<generic>, <signature>) (more details later)

**S3 Class/Method: Example 1**

What’s happening here?

set.seed(2)

x <- rnorm(100)

mean(x)

## [1] -0.0307

1. The class of x is “numeric”
2. But there is no mean method for “numeric” objects!
3. So we call the default function for mean.

**S3 Class/Method: Example 1**

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")

**S3 Class/Method: Example 1**

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

##

## 15 if (any(is.na(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 }

**S3 Class/Method: Example 2**

What happens here?

set.seed(3)

df <- data.frame(x = rnorm(100), y = 1:100)

sapply(df, mean)

## x y

## 0.01104 50.50000

1. The class of df is "data.frame"; each column can be an object of a different class
2. We sapply over the columns and call the mean function
3. In each column, mean checks the class of the object and dispatches the appropriate method.
4. We have a numeric column and an integer column; mean calls the default method for both

**Calling Methods Directly**

* Some S3 methods are visible to the user (i.e. mean.default),
* **Never** call methods directly
* Use the generic function and let the method be dispatched automatically.
* With S4 methods you cannot call them directly at all

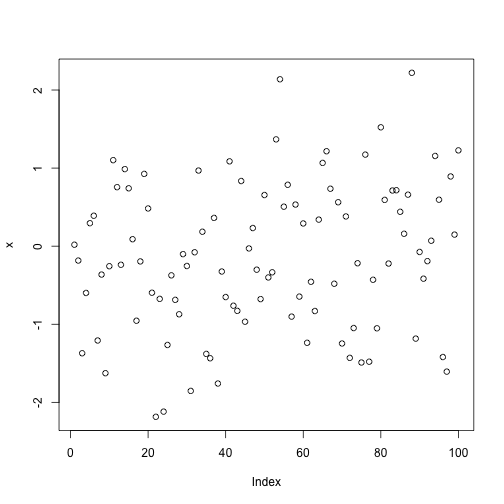
**S3 Class/Method: Example 3**

The plot function is generic and its behavior depends on the object being plotted.

set.seed(10)

x <- rnorm(100)

plot(x)

[](https://github.com/bcaffo/courses/blob/master/09_DevelopingDataProducts/classes-methods/assets/fig/unnamed-chunk-12.png)

**S3 Class/Method: Example 3**

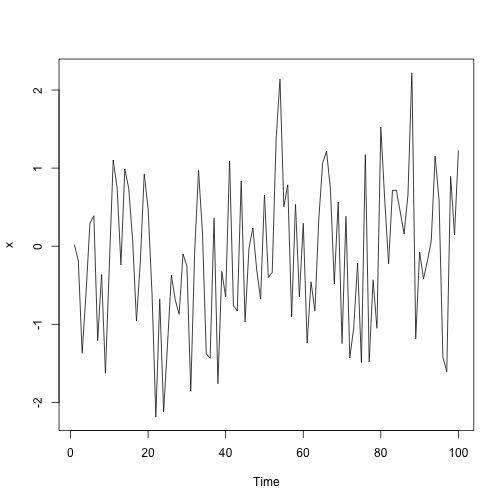
For time series objects, plot connects the dots

set.seed(10)

x <- rnorm(100)

x <- as.ts(x) ## Convert to a time series object

plot(x)

[](https://github.com/bcaffo/courses/blob/master/09_DevelopingDataProducts/classes-methods/assets/fig/unnamed-chunk-13.png)

**Write your own methods!**

If you write new methods for new classes, you’ll probably end up writing methods for the following generics:

* print/show
* summary
* plot

There are two ways that you can extend the R system via classes/methods

* Write a method for a new class but for an existing generic function (i.e. like print)
* Write new generic functions and new methods for those generics

**S4 Classes**

Why would you want to create a new class?

* To represent new types of data (e.g. gene expression, space-time, hierarchical, sparse matrices)
* New concepts/ideas that haven’t been thought of yet (e.g. a fitted point process model, mixed-effects model, a sparse matrix)
* To abstract/hide implementation details from the user I say things are “new” meaning that R does not know about them (not that they are new to the statistical community).

**S4 Class/Method: Creating a New Class**

A new class can be defined using the setClass function

* At a minimum you need to specify the name of the class
* You can also specify data elements that are called *slots*
* You can then define methods for the class with the setMethod function Information about a class definition can be obtained with the showClass function

**S4 Class/Method: Polygon Class**

Creating new classes/methods is usually not something done at the console; you likely want to save the code in a separate file

library(methods)

setClass("polygon",

representation(x = "numeric",

y = "numeric"))

* The slots for this class are xand y
* The slots for an S4 object can be accessed with the @ operator.

**S4 Class/Method: Polygon Class**

A plot method can be created with the setMethod function.

* For setMethod you need to specify a generic function (plot), and a *signature*.
* A signature is a character vector indicating the classes of objects that are accepted by the method.
* In this case, the plot method will take one type of object, a polygon object.

**S4 Class/Method: Polygon Class**

Creating a plot method with 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)

})

## Creating a generic function for 'plot' from package 'graphics' in the global environment

## [1] "plot"

* Notice that the slots of the polygon (the x- and y-coordinates) are accessed with the @ operator.

**S4 Class/Method: Polygon Class**

After calling setMethod the new plot method will be added to the list of methods for plot.

library(methods)

showMethods("plot")

## Function: plot (package graphics)

## x="ANY"

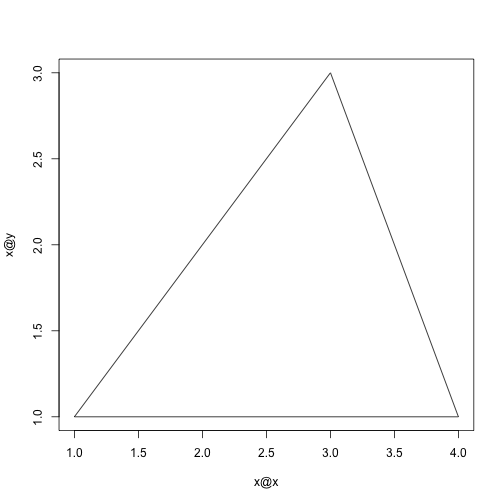
## x="polygon"

Notice that the signature for class polygon is listed. The method for ANY is the default method and it is what is called when now other signature matches

**S4 Class/Method: Polygon class**

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

plot(p)

[](https://github.com/bcaffo/courses/blob/master/09_DevelopingDataProducts/classes-methods/assets/fig/unnamed-chunk-17.png)

**Summary**

* Developing classes and associated methods is a powerful way to extend the functionality of R
* **Classes** define new data types
* **Methods** extend **generic functions** to specificy the behavior of generic functions on new classes
* As new data types and concepts are created, classes/methods provide a way for you to develop an intuitive interface to those data/concepts for users

**Where to Look, Places to Start**

* The best way to learn this stuff is to look at examples
* There are quite a few examples on CRAN which use S4 classes/methods. You can usually tell if they use S4 classes/methods if the **methods** package is listed in the Depends: field
* Bioconductor ([http://www.bioconductor.org](http://www.bioconductor.org/)) — a rich resource, even if you know nothing about bioinformatics
* Some packages on CRAN (as far as I know) — SparseM, gpclib, flexmix, its, lme4, orientlib, filehash
* The stats4 package (comes with R) has a bunch of classes/methods for doing maximum likelihood analysis.