

Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures	vii
List of Tables	ix
1 Base R: Adding New “Words”	1
1.1 Aims of this chapter	1
1.2 Defining functions and operators	1
1.2.1 Scope of names	5
1.2.2 Ordinary functions	6
1.2.3 Operators	7
1.3 Objects, classes, and methods	8
1.4 Packages	10
1.4.1 Sharing of R-language extensions	10
1.4.2 Download, installation and use	11
1.4.3 Finding suitable packages	14
1.4.4 How packages work	15
1.5 Further reading	16
Bibliography	17
General Index	19
Alphabetic Index of R Names	21
Index of R Names by Category	23
Frequently Asked Questions	25



List of Figures

1.1	Diagram of function with no side effects	3
1.2	Diagram of function with side effects	3



List of Tables



1

Base R: Adding New “Words”

Computer Science is a science of abstraction—creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.

Alfred V. Aho and Jeffrey D. Ullman
Foundations of Computer Science, 1992

1.1 Aims of this chapter

In earlier chapters we have only used base R features. In this chapter you will learn how to expand the range of features available. We will start by discussing how to define and use new functions, operators and classes. Later we will focus on using existing packages and touch briefly on how they work. We will not consider the important, but more advanced question of packaging functions and classes into new R packages.

1.2 Defining functions and operators

Abstraction can be defined as separating the fundamental properties from the accidental ones. Say obtaining the mean from a given vector of numbers is an actual operation. There can be many such operations on different numeric vectors, each one a specific case. When we describe an algorithm for computing the mean from any numeric vector we have created the abstraction of *mean*. In the same way, each time we separate operations from specific data we create a new abstraction. In this sense, functions are abstractions of operations or actions; they are like “verbs” describing actions separately from actors.

The main role of functions is that of providing an abstraction allowing us to avoid repeating blocks of code (groups of statements) applying the same opera-

tions on different data. The reasons to avoid repetition of similar blocks of code statements are that 1) if the algorithm or implementation needs to be revised—e.g., to fix a bug or error—it is best to make edits in a single place; 2) sooner or later pieces of repeated code can become different leading to inconsistencies and hard-to-track bugs; 3) abstraction and division of a problem into smaller chunks, greatly helps with keeping the code understandable to humans; 4) textual repetition makes the script file longer, and this makes debugging, commenting, etc., more tedious, and error prone.

How do we, in practice, avoid repeating bits of code? We write a function containing the statements that we would need to repeat, and later we *call* (“use”) the function in their place. We have been calling R functions or operators in almost every example in this book; what we will next tackle is how to define new functions of our own.

We saw in section ?? on page ?? a diagram of a compound statement. A function is a code statement, simple or compound, that is partly isolated from the enclosing environment. The *function* abstraction relies on formal parameters working as placeholders for arguments within the function body. When the function is called (or “used”) values are passed as arguments to the parameters, and used when executing the code within the function.

New functions and operators are defined using function `function()`, and saved like any other object in R by assignment to a variable name. In the example below, `x` and `y` are both formal parameters, or names used within the function for objects that will be supplied as *arguments* when the function is called.

Function `my.prod()` has two formal parameters, `x` and `y`.

```
fun1 <- function(x, y){x * y}
```

When we call `fun1()` with 4 and 3 as arguments, the computation that takes place is `4 * 3` and the value returned is 12. In this example it is printed, but we could have assigned it to a variable or used it in further computations within the calling statement.

```
fun1(x = 4, y = 3)
## [1] 12
```

1.1 What is the computation that takes places in this function call?

```
fun1(x = 10, y = 50)
```



Even though the statements within the function body do have access to the environment in which the function is called, it is safest to pass all input through the function parameters, and return all values to the caller. This ensures that the users of the function can treat it a black box with no side effects.

In R, statements within the function usually do not affect directly any variable defined outside the function, the result from the computation is returned as a value. The diagram in Figure 1.1 describes a function that has no *side effects*, as it does not affect its environment, it only returns a value to the caller. A value on which the caller has full control. The statement that calls the function “decides” what to do with the value received from the function.

When a function has a side effect, the caller is no longer in full control (Figure

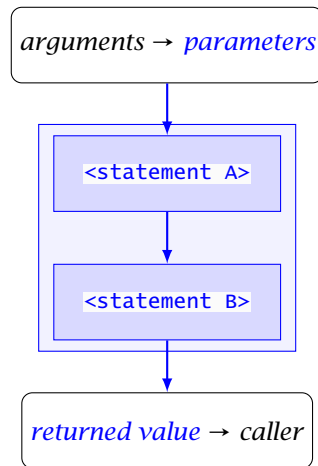
**FIGURE 1.1**

Diagram of function with no side effects, seen as a compound code statement receiving its input as arguments passed to its formal parameters and returning an object or value to the statement from where it was called or run. The body of the function is represented by the filled box.

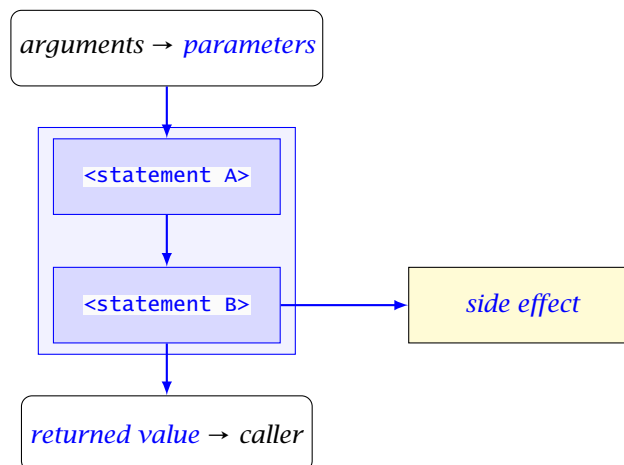

**FIGURE 1.2**

Diagram of function as a compound code statement receiving its input as arguments passed to its formal parameters and returning an object or value to the statement from where it was called or run. The body of the function is represented by the box filled in blue, while the side effect of the code in the function directly outside is represented by the box filled in yellow.

1.2). Side effects can be actions that do not alter any object in the calling code, like when a call to `print()` displays text or numbers. Side effects can also be an assignment that modifies an object in the caller’s environment, such as assigning a new value to a variable in the caller’s environment, i.e., “outside the function”.


A function can return only one object, so when multiple results are produced they need to be collected into a single object. In many cases, lists are used to collect all the values to be returned into one R object. For example, model fit functions like `lm()`, discussed in section ?? on page ??, return lists with multiple heterogeneous members, plus ancillary information stored in several attributes. In the case on `lm()` the returned object’s class is `lm`, and its mode is `list`.

 **1.2** When function `return()` is called within a function, flow of execution within the function stops and the argument passed to `return()` is the value returned by the function call. In contrast, if function `return()` is not explicitly called, the value returned by the function call is that returned by the last statement *executed* within the body of the function. Run these examples, and your own variations.

```
FN1 <- function(x) print("prn")
FN1("arg")
FN2 <- function(x){print("prn")
                    return(x)}
FN2("arg")
FN3 <- function(x){return(x)
                    print("prn")}
FN3("arg")
FN4 <- function(x){return()
                    print("prn")}
FN4("arg")
FN5 <- function(x){return(print(x))
                    print("prn")}
FN5("arg")
```

In base R, arguments to functions are passed by copy. This is something important to remember. If code in a function’s body modifies the value of a parameter (the placeholder for an argument), its value outside the function is not affected, e.g., if the argument passed was a variable.

```
fn2 <- function(x){x <- 99}
a <- 1
fn2(a)
a
## [1] 1
```

 In some other computer languages, arguments can be passed by reference, meaning that assignments to a formal parameter within the body of the function are back-referenced to the argument and modify it. It is possible to imitate such behavior in R using some language trickery and consequently, occasionally functions in R use this approach.

Functions have their own *scope*. Any new variables created by normal assignment within the body of a function are visible only within the body of the function and are destroyed when the function returns from the call. In normal use, functions in R do not affect their environment through side effects.

1.2.1 Scope of names

Scoping in R is implemented using *environments* and *name spaces*. We can think of environments as having a boundary with asymmetric visibility. The code within a function runs in its own environment, in isolation from the calling environment in relation to assignments, but the values stored in objects in the calling environment can be retrieved. This protects from unintentional side effects by making difficult to overwrite object definitions in the calling environment. It is possible to override this protection with operator `<-` or with function `assign()`. When used, assignment as side effects, can make the code much more difficult to read and debug, so its best to avoid them.

The visibility of names is determined by the *scoping rules* of a language. The clearest, but not the only situation when scoping rules matter, is when objects with the same name coexist. In such a situation one will be accessible by its unqualified name and the other hidden but possibly accessible by qualifying the name with the namespace where it is defined.


As the R language has few reserved words for which no redefinition is allowed, we should take care not to accidentally reuse names that are part of language. For example `pi` is a constant defined in R with the value of the mathematical constant π . If we use the same name for one of our variables, the original definition becomes hidden.

```
pi
## [1] 3.141593
pi <- "apple pie"
pi
## [1] "apple pie"
rm(pi)
pi
## [1] 3.141593
exists("pi")
## [1] TRUE
```

In the example above, the two variables are not defined in the same scope. In the example below we assign a new value to a variable we have earlier created within the same scope, and consequently the second assignment overwrites, rather than hides, the existing definition.

```
my.pie <- "raspberry pie"
my.pie
## [1] "raspberry pie"
my.pie <- "apple pie"
my.pie
## [1] "apple pie"
rm(my.pie)
exists("my.pie")
## [1] FALSE
```

Name spaces play an important role in avoiding name clashes when contributed packages are attached (see section 1.4.4 on page 15).

 Environments can be explicitly created with function `environment()`. However, `environment()` is rarely used in scripts while it can be useful within packages.

1.2.2 Ordinary functions

After the toy examples above, we will define a small but useful function: a function for calculating the standard error of the mean from a numeric vector. The standard error is given by $S_{\hat{x}} = \sqrt{S^2/n}$. We can translate this into the definition of an R function called `SEM`.

```
SEM <- function(x){sqrt(var(x) / length(x))}
```

We can test our function.

```
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
SEM(x = a)
## [1] 1.796988
SEM(a)
## [1] 1.796988
SEM(a.na)
## [1] NA
```

For example in `SEM(a)` we are calling function `SEM()` with `a` as an argument.

The function we defined above will always give the correct answer because `NA` values in the input will always result in an `NA` being returned. The problem is that unlike R’s functions like `var()`, there is no option to omit `NA` values in the function we defined.

This could be implemented by adding a second parameter `na.omit` to the definition of our function and passing its argument to the call to `var()` within the body of `SEM()`. However, to avoid returning wrong values we need to make sure `NA` values are also removed before counting the number of observations with `length()`.


A readable way of implementing this in code is to define the function as follows.

```
sem <- function(x, na.omit = FALSE) {
  if (na.omit) {
    x <- na.omit(x)
  }
  sqrt(var(x)/length(x))
}
```

```
sem(x = a)
## [1] 1.796988
sem(x = a.na)
## [1] NA
sem(x = a.na, na.omit = TRUE)
## [1] 1.796988
```

R does not provide a function for standard error, so the function above is generally useful. Its user interface is consistent with that of functionally similar existing functions. We have added a new word to the R vocabulary available to us.

In the definition of `sem()` we set a default argument for parameter `na.omit` which is used unless the user explicitly passes an argument to this parameter.

 **1.3** Define your own function to calculate the mean in a similar way as `SEM()` was defined above. Hint: function `sum()` could be of help.

Within an expression, a function name followed by parentheses is interpreted as a call to the function, while the bare name of a function, returns its definition (similarly to any other R object). If the name is entered as a statement at the R console, its value is printed.

We first print (implicitly) the definition of our function from earlier in this section.

```
sem
## function(x, na.omit = FALSE) {
##   if (na.omit) {
##     x <- na.omit(x)
##   }
##   sqrt(var(x)/length(x))
## }
## <bytecode: 0x0000024457946918>
```

Next we print the definition of R's standard deviation function `sd()`.

```
sd
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##   na.rm = na.rm))
## <bytecode: 0x0000024458fb1430>
## <environment: namespace:stats>
```

As can be seen at the end of the printouts, these functions written in the R language have been byte-compiled so that they executes faster. We can also see that the definition of `sd()` resides in `namespace:stats` because it has been attached from package 'stats'.

Functions that are part of the R language, but that are not coded using the R language, are called primitives and their full definition cannot be accessed through their name (c.f., `sem()` defined above and `sd`, with `list()` below).

```
list
## function (...) .Primitive("list")
```

1.2.3 Operators

Operators are functions that use a different syntax for being called. If their name is enclosed in back ticks they can be called as ordinary functions. Binary operators like `+` have two formal parameters, and unary operators like unary `-` have only one formal parameter. The parameters of many binary R operators are named `e1` and `e2`.


```
1 / 2
## [1] 0.5
`/`(1, 2)
## [1] 0.5
`/`(e1 = 1, e2 = 2)
## [1] 0.5
```

An important consequence of the possibility of calling operators using ordinary

syntax is that operators can be used as arguments to *apply* functions in the same way as ordinary functions. When passing operator names as arguments to *apply* functions we only need to enclose them in back ticks (see section ?? on page ??).

The name by itself and enclosed in back ticks allows us to access the definition of an operator.

```
`/`  
## function (e1, e2) .Primitive("/")
```

 **Defining a new operator.** We will define a binary operator (taking two arguments) that subtracts from the numbers in a vector the mean of another vector. First we need a suitable name, but we have less freedom as names of user-defined operators must be enclosed in percent signs. We will use `%-mean%` and as with any *special name*, we need to enclose it in quotation marks for the assignment.

```
"%-mean%" <- function(e1, e2) {  
  e1 - mean(e2)  
}
```

We can then use our new operator in an example.

```
10:15 %-mean% 1:20  
## [1] -0.5  0.5  1.5  2.5  3.5  4.5
```

To print the definition, we enclose the name of our new operator in back ticks—i.e., we *back quote* the special name.

```
`%-mean%`  
## function(e1, e2) {  
##   e1 - mean(e2)  
## }
```

1.3 Objects, classes, and methods

New classes are normally defined within packages rather than in user scripts. To be really useful implementing a new class involves not only defining a class but also a set of specialized functions or *methods* that implement operations on objects belonging to the new class. Nevertheless, an understanding of how classes work is important even if only very occasionally a user will define a new method for an existing class within a script.

Classes are abstractions, but abstractions describing the shared properties of “types” or groups of similar objects. In this sense, classes are abstractions of “actors,” they are like “nouns” in natural language. What we obtain with classes is the possibility of defining multiple versions of functions (or *methods*) sharing the same name but tailored to operate on objects belonging to different classes. We have already been using methods with multiple *specializations* throughout the book, for example `plot()` and `summary()`.

We start with a quotation from *S Poetry* (Burns 1998, page 13).

The idea of object-oriented programming is simple, but carries a lot of

weight. Here's the whole thing: if you told a group of people “dress for work,” then you would expect each to put on clothes appropriate for that individual's job. Likewise it is possible for S[R] objects to get dressed appropriately depending on what class of object they are.

We say that specific methods are *dispatched* based on the class of the argument passed. This, together with the loose type checks of R, allows writing code that functions as expected on different types of objects, e.g., character and numeric vectors.

R has good support for the object-oriented programming paradigm, but as a system that has evolved over the years, currently R supports multiple approaches. The still most popular approach is called S3, and a more recent and powerful approach, with slower performance, is called S4. The general idea is that a name like “plot” can be used as a generic name, and that the specific version of `plot()` called depends on the arguments of the call. Using computing terms we could say that the *generic* of `plot()` dispatches the original call to different specific versions of `plot()` based on the class of the arguments passed. S3 generic functions dispatch, by default, based only on the argument passed to a single parameter, the first one. S4 generic functions can dispatch the call based on the arguments passed to more than one parameter and the structure of the objects of a given class is known to the interpreter. In S3 functions, the specializations of a generic are recognized/identified only by their name. And the class of an object by a character string stored as an attribute to the object (see section ?? on page ?? about attributes).

We first explore one of the methods already available in R. The definition of `mean` shows that it is the generic for a method.


```
mean
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000024458480da8>
## <environment: namespace:base>
```

We can find out which specializations of method are available in the current search path using `methods()`.

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

We can also use `methods()` to query all methods, including operators, defined for objects of a given class.

```
methods(class = "list")
## [1] all.equal      as.data.frame  coerce         Ops            relist
## [6] sew            type.convert   within
## see '?methods' for accessing help and source code
```

 S3 class information is stored as a character vector in an attribute named “`class`”. The most basic approach to creation of an object of a new S3 class, is to add the new class name to the class attribute of the object. As the implied class hierarchy is given by the order of the members of the character vector, the name

of the new class must be added at the head of the vector. Even though this step can be done as shown here, in practice this step would normally take place within a *constructor* function and the new class, if defined within a package, would need to be registered. We show here this bare-bones example to demonstrate how S3 classes are implemented in R.


```
a <- 123
class(a)
## [1] "numeric"
class(a) <- c("myclass", class(a))
class(a)
## [1] "myclass" "numeric"
```

Now we create a print method specific to "myclass" objects. Internally we are using function `sprintf()` and for the format template to work we need to pass a numeric value as an argument—i.e., obviously `sprintf()` does not “know” how to handle objects of the class we have just created!

```
print.myclass <- function(x) {
  sprintf("[myclass] %.0f", as.numeric(x))
}
```

Once a specialized method exists for a class, it will be used for objects of this class.

```
print(a)
## [1] "[myclass] 123"
print(as.numeric(a))
## [1] 123
```

 The S3 class system is “lightweight” in that it adds very little additional computation load, but it is rather “fragile” in that most of the responsibility for consistency and correctness of the design—e.g., not messing up dispatch by re-defining functions or loading a package exporting functions with the same name, etc., is not checked by the R interpreter.

1.4 Packages

1.4.1 Sharing of R-language extensions

The most elegant way of adding new features or capabilities to R is through packages. This is without doubt the best mechanism when these extensions to R need to be shared. However, in most situations it is also the best mechanism for managing code that will be reused even by a single person over time. R packages have strict rules about their contents, file structure, and documentation, which makes it possible among other things for the package documentation to be merged into R’s help system when a package is loaded. With a few exceptions, packages can be written so that they will work on any computer where R runs.

Packages can be shared as source or binary package files, sent for example

through e-mail. However, the largest public repository of R packages is called CRAN (<https://cran.r-project.org/>), an acronym for Comprehensive R Archive Network. Packages available through CRAN are guaranteed to work, in the sense of not failing any tests built into the packages and not crashing or aborting prematurely. They are tested daily, as they may depend on other packages whose code will change when updated. The number of packages available through CRAN at the time of printing (2023-09-28) was 1.99×10^4 .

A key repository for bioinformatics with R is Bioconductor (<https://www.bioconductor.org/>), containing packages that pass strict quality tests, adding an additional 3 400 packages. ROpenScience has established guidelines and a system for code peer review for R packages. These peer-reviewed packages are available through CRAN or other repositories and listed at the ROpenScience website (<https://ropensci.org/>). Occasionally one may have or want to install packages directly from Git repositories such as versions still under development and not yet submitted to CRAN.

One good way of learning how the extensions provided by a package work, is by experimenting with them. When using a function we are not yet familiar with, looking at its help to check all its features will expand your “toolbox.” How much documentation is included with packages varies, while documentation of exported objects is enforced, many packages include, in addition, comprehensive user guides or articles as *vignettes*. It is not unusual to decide which package to use from a set of alternatives based on the quality of available documentation. In the case of packages adding extensive new functionality, they may be documented in depth in a book. Well-known examples are *Mixed-Effects Models in S and S-Plus* (Pinheiro and Bates 2000), *Lattice: Multivariate Data Visualization with R* (Sarkar 2008) and *ggplot2: Elegant Graphics for Data Analysis* (Wickham and Sievert 2016).

1.4.2 Download, installation and use

In R speak, “library” is the location where packages are installed. Packages are sets of functions, and data, specific for some particular purpose, that can be loaded into an R session to make them available so that they can be used in the same way as built-in R functions and data. Function `library()` is used to load and attach packages that are already installed in the local R library. In contrast, function `install.packages()` is used to install packages.



The instructions below assume that user has access to repositories in the internet and enough user rights to install packages. This is rarely the cases in organizations using strict security protocols. In such cases, the organization may keep a mirror of CRAN in the intranet. The local/user’s private R library can be kept in a folder where the user has writing and reading rights.



How to install or update a package from CRAN?

CRAN is the default repository for R packages. If you use RStudio or another IDE as front end on any operating system or RGUI under MS-Windows, installation and updates can be done through a menu or GUI ‘button’. These menus use calls to `install.packages()` and `update.packages()` behind the scenes.

Alternatively, at the R command line, or in a script, `install.packages()` can be called with the name of the package as argument. For example, to install package ‘learnrbook’ we use

```
install.packages("learnrbook")
```

or alternatively, using package ‘pak’.

```
pak::pkg_install("learnrbook")
```

Already installed packages are updated with function `update.packages()`.

How to install an R package from GitHub?

Package ‘remotes’ makes it possible to install packages directly from GitHub, Bitbucket and other code repositories based on Git. The code in the next chunk (not run here) can be used to install the latest, possibly under development version of package ‘learnrbook’.

```
remotes::install_github("aphalo/learnrbook")
```


Alternatively, the newer package ‘pak’ can be used.

```
pak::pkg_install("aphalo/learnrbook")
```

R packages can be installed either from sources, or from already built “binaries”. Installing from sources, depending on the package, may require additional software to be available. This is because some R packages contain source code in other languages such as C, C++ or FORTRAN that needs to be compiled into machine code during installation. Under MS-Windows, the needed shell, commands and compilers are not available as part of the operating system. Installing them is not difficult as they are available prepackaged in an installer under the name RTools (available from CRAN). MiKTeX is usually needed to build the PDF of the package’s manual.

Under MS-Windows it is easier to install packages from binary .zip files than from .tar.gz source files. For OS X (Apple Mac) the situation is similar, with binaries available both for Intel and ARM (M1, M2 series) processors. Most, but not all, Linux distributions include in the default setup the tools needed for installation of R packages. Under Linux it is rather common to install packages from sources, although package binaries have recently become more easily available.

If the tools are available, packages can be very easily installed from sources from within RStudio. However, binaries are for most packages also readily available. In CRAN, the binary for a new version of a package becomes available with a delay of one or two days compared to the source. For packages that need compilation, the installation from sources takes more time than installation from binaries.

 **1.4** Use `help` to look up the help page for `install.packages()`, and explore how to control whether the package is installed from a source or a binary file. Also explore, how to install a package from a file in a local disk instead of from a repository like CRAN.

Frequently the README file of a package includes instructions on how to install it from CRAN or another on-line repository. Exceptionally, packages may require additionally the installation of software outside R before their installation and/or use. When present, these rather exceptional requirements are always listed

in the DESCRIPTION under `systemRequirements`: and explained in more detail in the README file. In CRAN, each package has a home web page that can be easily found if one knows the name of the package, e.g., <https://CRAN.R-project.org/package=learnrbook>. Nowadays, it is common for the help for a package being also available as a web site, e.g., <https://docs.r4photobiology.info/learnrbook/>.

How to change the repository used to install packages?

Function `setRepositories()` can be used to enable other repositories in addition or instead of CRAN during an R session. In recent versions of R the default list of repositories is taken from R option "repos" if defined. Consult `help("setRepositories")` for the details.

Alternatively, one can use `pak::pkg_install()` as this function attempts to automatically set the correct repository based on the name of the package.

How to use an installed package?


To use the functions and other objects defined in a package, the package must first be loaded, and for the names of these objects to be visible in the user's workspace, the package needs to be attached. Function `library()` loads and attaches one package at a time. For example, to load and attach package 'learnrbook' we use.

```
library("learnrbook")
```

As packages are contributed by independent authors, they should be cited in addition to citing R itself when they are used to obtain results or plots included in publications. R function `citation()` when called with the name of a package as its argument provides the reference that should be cited for the package, and without an explicit argument, the reference to cite for the version of R in use as shown below.

```
citation()
## To cite R in publications use:
##
## R Core Team (2023). _R: A Language and Environment for Statistical
## Computing_. R Foundation for Statistical Computing, Vienna, Austria.
## <https://www.R-project.org/>.
##
## A BibTeX entry for LaTeX users is
##
## @Manual{,
##   title = {R: A Language and Environment for Statistical Computing},
##   author = {{R Core Team}},
##   organization = {R Foundation for Statistical Computing},
##   address = {Vienna, Austria},
##   year = {2023},
##   url = {https://www.R-project.org/},
## }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

 **1.5** Look at the help page for function `citation()` for a discussion of why it is important that users cite R and packages when using them.

 Conflicts among packages can easily arise, for example, when they use the same names for objects or functions. These are reported when the packages are attached (see section 1.4.4 on page 15 for a workaround). In addition, many packages use functions defined in packages in the R distribution itself or other independently developed packages by importing them. Updates to depended-upon packages can “break” (make non-functional) the dependent packages or parts of them. The rigorous testing by CRAN detects such problems in most cases when package revisions are submitted, forcing package maintainers to fix problems before distribution through CRAN is possible. However, if you use other repositories, I recommend that you make sure that revised (especially if under development) versions do work with your own code, before their use in “production” (important) data analyses.

1.4.3 Finding suitable packages

Due to the large number of contributed R packages it can sometimes be difficult to find a suitable package for a task at hand. It is good to first check if the necessary capability is already built into base R. Base R plus the recommended packages (installed when R is installed) cover a lot of ground. To analyze data using almost any of the more common statistical methods does not require the use of special packages. Sometimes, contributed packages duplicate or extend the functionality in base R with advantage. When one considers the use of novel or specialized types of data analysis, the use of contributed packages can be unavoidable. Even in such cases, it is not unusual to have alternatives to choose from within the available contributed packages. Sometimes groups or suites of packages are designed to work well together.

The CRAN repository has very broad scope and includes a section called “views.” R views are web pages providing annotated lists of packages frequently used within a given field of research, engineering or specific applications. These views are edited and updated by different editors. They can be found at <https://cran.r-project.org/web/views/>.


The Bioconductor repository specializes in bioinformatics with R. It also has a section with “views” and within it, descriptions of different data analysis workflows. The workflows are especially good as they reveal which sets of packages work well together. These views can be found at <https://www.bioconductor.org/packages/release/BiocViews.html>.

rOpenSci (Ram et al. 2019) fosters a culture that values open and reproducible research using shared data and reusable software. One aspect of this is making possible peer-review of R packages. rOpenSci does not keep a separate package repository for the peer-reviewed packages, they keep an index at <https://ropensci.org/packages/>. The packages included have becoming more diverse, but initially the focus was in facilitating access to open data sources.

The CRAN repository keeps an archive of earlier versions of packages, on an individual package basis. This is also important for long-term reproducibility.


1.4.4 How packages work

R packages define all objects within a *namespace* with the same name as the package itself. Loading and attaching a package with `library()` makes visible only the exported objects. Attaching a package adds these objects to the search path so that they can be accessed without prepending the name of the namespace. Most packages do not export all the functions and objects defined in their code; some are kept internal, in most cases because they may change or be removed in future versions.


 Package namespaces can be detached and also unloaded with function `detach()` using a slightly different notation for the argument from that which we described for data frames in section ?? on page ?. This is very seldom needed, but one case I have come across are packages that define a generic functions of the same name and interfere with each other.

When we reuse a name defined in a package, its definition in the package does not get overwritten, but instead, only hidden. These hidden objects remain accessible using the name *qualified* by prepending the name of the package followed by two colons, e.g., `base:mean()`.


If two packages define objects with the same name, then which one is visible depends on the order in which the packages were attached. To avoid confusion in such cases, in scripts it is best to use the qualified names for calling objects defined with the same name in two packages. Using the qualified name for an object from an already attached package, is inconsequential for its interpretation by R, but can enhance the readability of the code.

 If one uses a qualified name for an object but does not attach the package with a call to `library`, the package is only loaded. In other words, the names of the exported objects are not added to the search pass, but the code defining them is retrieved and available using qualified names.

Some functions that are part of R are collected into packages grouped by category: ‘base’, ‘stats’, ‘datasets’, etc., and can be called when needed using qualified names. We can find out the search order by calling `search()`, with the search starting at the `".GlobalEnv"` for statements evaluated at the R command line.

 **1.6** Namespaces isolate the names defined within them from those in other namespaces. This helps prevent name clashes, and makes it possible to access objects even when they are “hidden” by a different object with the same name.

```
class(cars)
head(cars, 3)
getAnywhere("cars")$where # defined in package
cars <- 1:10
class(cars)
head(cars, 3) # prints 'cars' defined in the global environment
rm(cars) # clean up
head(cars, 3)
getAnywhere("cars")$where # the first visible definition is in the global environment
```

 In the playground above I used a data frame object, but the same mechanisms apply to all R objects including functions. The situation when one of the definitions is a function and the other is not, is slightly different in that a call using parenthesis notation will distinguish between a function and an object of the same name that is not a function. Relying on this distinction is anyway a bad idea.

```
mean
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000024458480da8>
## <environment: namespace:base>
mean <- mean(1:5)
mean
## [1] 3
mean(8:9)
## [1] 8.5
getAnywhere("mean")$where
## [1] ".GlobalEnv"      "package:base"      "namespace:base"
rm(mean)
getAnywhere("mean")$where
## [1] "package:base"      "namespace:base"
```

In this last example I removed with `rm(mean)` the variable we had assigned a value to. Package namespaces also prevent deletion or overwriting of objects defined in the package. This is different to defining a new object with the same name, which is allowed. The two statements below trigger errors and are not evaluated when typesetting the book.

```
datasets::cars <- "my car is green"
rm(datasets::cars)
```

We looked at only one member of the value returned by `getAnywhere()`, do have a look at its help page for more details as it contains additional information.

1.5 Further reading

Several books describe in detail the different class systems available and how to use them in R. For an in-depth treatment of the subject please consult the books *Advanced R* (Wickham 2019) and *Extending R* (Chambers 2016).

The development of R packages is accessibly explained in the book *R for Data Science* (Wickham et al. 2023), using a practical approach and tools developed by the author and his collaborators. The book *Extending R* (Chambers 2016) has its focus on R itself, how it works, and how to develop extensions both with simple and with challenging goals.

Bibliography

- Aho, A. V. and J. D. Ullman (1992). *Foundations of computer science*. Computer Science Press. ISBN: 0716782332.
- Burns, P. (1998). *S Poetry* (cit. on p. 8).
- Chambers, J. M. (2016). *Extending R*. The R Series. Chapman and Hall/CRC. ISBN: 1498775713 (cit. on p. 16).
- Pinheiro, J. C. and D. M. Bates (2000). *Mixed-Effects Models in S and S-Plus*. New York: Springer (cit. on p. 11).
- Ram, K., C. Boettiger, S. Chamberlain, N. Ross, M. Salmon, and S. Butland (Mar. 2019). “A Community of Practice Around Peer Review for Long-Term Research Software Sustainability”. In: *Computing in Science & Engineering* 21.2, pp. 59–65. DOI: 10.1109/mcse.2018.2882753 (cit. on p. 14).
- Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. 1st ed. Springer, p. 268. ISBN: 0387759689 (cit. on p. 11).
- Wickham, H. (2019). *Advanced R*. 2nd ed. Chapman and Hall/CRC. 588 pp. ISBN: 0815384572 (cit. on p. 16).
- Wickham, H., M. Cetinkaya-Rundel, and G. Grolemund (2023). *R for Data Science. Import, Tidy, Transform, Visualize, and Model Data*. O’Reilly Media. ISBN: 9781492097402 (cit. on p. 16).
- Wickham, H. and C. Sievert (2016). *ggplot2: Elegant Graphics for Data Analysis*. 2nd ed. Springer. XVI + 260. ISBN: 978-3-319-24277-4 (cit. on p. 11).



General Index

- 'base', 15
- Bioconductor, 11
- Bitbucket, 12
- C, 12
- C++, 12
- classes, 8
 - S3 class system, 8
- CRAN, 11, 14
- 'datasets', 15
- extensions to R, 10
- FORTTRAN, 12
- functions
 - arguments, 4
 - defining new, 1, 6
- further reading
 - object oriented programming in
 - R, 16
 - package development, 16
- Git, 12
- GitHub, 12
- languages
 - C, 12
 - C++, 12
 - FORTTRAN, 12
- 'learnrbook', 12
- methods, 8
 - S3 class system, 8
- MiKTeX, 12
- MS-Windows, 11, 12
- names and scoping, 5
- namespaces, 5
- object-oriented programming, 8
- objects, 8
- operators
 - defining new, 1, 7
- OS X, 12
- packages
 - 'base', 15
 - 'datasets', 15
 - 'learnrbook', 12
 - 'pak', 12
 - 'remotes', 12
 - 'stats', 7, 15
 - using, 11
- 'pak', 12
- programmes
 - Git, 12
 - MiKTeX, 12
 - MS-Windows, 11, 12
 - OS X, 12
 - RGUI, 11
 - RStudio, 11, 12
 - RTools, 12
- 'remotes', 12
- RGUI, 11
- ROpenScience, 11
- RStudio, 11, 12
- RTools, 12
- S3 class system, 8
- scoping rules, 5
- 'stats', 7, 15



Alphabetic Index of R Names

<code><<-</code> , 5	<code>lm</code> , 4
<code>assign()</code> , 5	<code>lm()</code> , 4
<code>citation()</code> , 13	<code>methods()</code> , 9
<code>detach()</code> , 15	<code>pak::pkg_install()</code> , 13
<code>environment()</code> , 6	<code>plot()</code> , 9
<code>exists()</code> , 5	<code>print()</code> , 4
<code>function()</code> , 2	<code>return()</code> , 4
<code>getAnywhere()</code> , 16	<code>search()</code> , 15
<code>install.packages()</code> , 11, 12	<code>SEM()</code> , 6, 7
<code>library</code> , 15	<code>setRepositories()</code> , 13
<code>library()</code> , 11, 13, 15	<code>sprintf()</code> , 10
<code>list</code> , 4	<code>sum()</code> , 7
	<code>update.packages()</code> , 11, 12
	<code>var()</code> , 6



Index of R Names by Category

R names and symbols grouped into the categories ‘classes and modes’, ‘constant and special values’, ‘control of execution’, ‘data objects’, ‘functions and methods’, ‘names and their scope’, and ‘operators’.

classes and modes

- `list`, 4

- `lm`, 4

control of execution

- `return()`, 4

functions and methods

- `assign()`, 5

- `citation()`, 13

- `environment()`, 6

- `function()`, 2

- `getAnywhere()`, 16

- `install.packages()`, 11, 12

- `library`, 15

- `library()`, 11, 13, 15

- `lm()`, 4

- `methods()`, 9

- `pak::pkg_install()`, 13

- `plot()`, 9

- `print()`, 4

- `search()`, 15

- `SEM()`, 6, 7

- `setRepositories()`, 13

- `sprintf()`, 10

- `sum()`, 7

- `update.packages()`, 11, 12

- `var()`, 6

names and their scope

- `detach()`, 15


- `exists()`, 5

operators

- `<<-`, 5



Frequently Asked Questions

Frequently asked questions and their answers appear in the body of the book preceded by the icon  and highlighted by a marginal bar of the same colour as the icon.

How to change the repository used to install packages?, 13

How to install an R package from GitHub?, 12

How to install or update a package from CRAN?, 11

How to use an installed package?, 13