Introduction to R Statistics 506

About R

As described by The R Foundation, "R is a language and environment for statistical computing and graphics." Importantly, R is open-source, free software distributed under a GNU GPL-3 license.

It is also easily extensible through contributed packages that cover much of modern statistics and data science.

RStudio

RStudio is an "integrated development environment" (IDE) for working with R that simplifies many tasks and makes for a friendlier introduction to R. It provides a nice interface via Quarto or RMarkdown for integrating R code with text for creating documents. RStudio is distributed by Posit that also offers a number of related products for working with data: Shiny for interactive graphics along with enterprise and server editions. I suggest you use RStudio when feasible to streamline your workflow.

R Basics

Objects

Nearly everything in **R** is an object that can be referred to by name.

Assignment

We generally create objects by assigning values to them:

```
# This is a comment ignored by R
username <- "William"
x <- 10
```

```
y <- 32
z <- c(x, y) # Form vectors by combining or concatenating elements.

9 -> w # This works, but is bad style.
theAnswer = 42 # Most other languages use = for assignment.
```

While both `<-` and `=` can be used exchangeable when assigning values to objects (in almost all situations), it is recommended to use `<-`. `=` is used when passing arguments to functions, and the distinction between `<-` assigning objects and `=` passing arguments makes code easier to read.

The values can be referred to by the object name:

```
theAnswer
```

```
[1] 42
```

Naming Objects

Object names can be any syntactically valid name. You should, however, avoid clobbering built in `R` names such as `pi`, `mean`, `sum`, etc.

You also should not use reserved words when naming objects.

You should generally use meaningful, descriptive names for objects you create. There are various styles for creating multi-word object names:

- `snake_case` : separate words with an underscore
- `camelCase` : capitalize subsequent words
- `dot.case` : separate words with a period

Object names are case sensitive:

```
q <- 3
Q <- 4
c(q, Q)
```

```
[1] 3 4
```

Style notes

Generally speaking, you do not need to adopt my coding style. However, you should develop and use a *consistent style* of your own. See the document on Developing R style for more discussion.

Value vs Reference

It is important to remember that in **R** objects are typically stored by value and **not** by reference:

```
x <- 10
y <- 32
z <- c(x, y)
c(x, y, z)
```

```
[1] 10 32 10 32
```

```
y <- theAnswer
c(x, y, z)
```

```
[1] 10 42 10 32
```

In contrast, if `z <- c(x,y)` were a reference to the contents of `x` and `y` then changing `y` would change `z` and the value referred to by `z` as well. At a more technical level, *R* has *copy on modify semantics* and uses *lazy evaluation* meaning objects at times are stored by reference. We will learn about these later in the course, but for reasoning about R programs it is generally sufficient to think of objects holding values rather than references.

Later in this lesson we will learn how to create objects without assigning values to them.

Global Environment

An **environment** is a context in which the names we use to refer to **R** objects have meaning. For now, it is sufficient to know that the default environment where objects are assigned is a workspace called the global environment. You can view objects in the global environment using the function `ls()` and remove objects using `rm()`.

```
ls()
```

```
[1] "has_annotations" "q"               "Q"               "theAnswer"
[5] "username"        "w"               "x"               "y"
[9] "z"
```

```
rm(theAnswer)
ls()
```

```
[1] "has_annotations" "q"               "Q"               "username"
[5] "w"               "x"               "y"               "z"
```

We can remove multiple objects as well:

```
rm(username, w)
ls()
```

```
[1] "has_annotations" "q"               "Q"               "x"
[5] "y"               "z"
```

To clear the entire workspace use `rm(list=ls())`.

There are three reasons you might want to remove objects from your environment:

1. Space considerations. Obviously the memory required to store the value `4` is trivial, but as we get into loading large data sets, if you create multiple distinct versions of the data, you may run into memory limits.
2. Clear environment, clear mind. If you're proactive in removing objects you don't need, those you do need are easily to locate.
3. Avoiding unintended effects. If you have objects hanging around that are old, you may miss errors that occur and not understand why your code is performing a certain way. Consider the following

```
y <- 4
x < -7
```

```
[1] FALSE
```

```
z <- x + y
```

What value will `z` take on?

```
z
```

```
[1] 14
```

You'd expect `z` to be 11. If you noticed the error in the line `x < -7` (which is comparing `x` and `-7`, not assigning a value), you'd expect an error there, and perhaps on the line creating `z`. However, since we created `x` much earlier in the notes and never removed it, the value of `x` from then is used unexpectedly!

Programmatic assignment

Occasionally, you may write a program where you need to access or assign an object whose name you do not know ahead of time. In these cases you may wish to use the `get()` or `assign()` functions.

```
# Ex 1
rm(list = ls())
ls()
```

```
character(0)
```

```
assign("new_int", 9) # i.e. new_int = 9
ls()
```

```
[1] "new_int"
```

```
get("new_int")
```

```
[1] 9
```

```
new_int
```

```
[1] 9
```

```
# Ex 2
rm(list = ls())
obj <- "obj_name"
val <- 42
assign(obj, value = val) # assign the value in 'val' to the value in 'obj'
ls()
```

```
[1] "obj"      "obj_name" "val"
```

```
obj
```

```
[1] "obj_name"
```

```
obj_name
```

```
[1] 42
```

Arithmetic operations

R can do arithmetic with objects that contain numeric types.

```
x <- 10
y <- 32
z <- x + y
x + y
```

```
[1] 42
```

```
z / x
```

```
[1] 4.2
```

```
z^2
```

```
[1] 1764
```

```
11 %% 2   # Modular arithmetic (i.e. what is the remainder)
```

```
[1] 1
```

```
11 %/% 2  # Integer division discards the remainder
```

```
[1] 5
```

R also supports vectorized arithmetic.

```
z + 2 * c(y, x)
```

```
[1] 106  62
```

Since `c(y, x)` is a vector of length two, the operations `z + 2 *` is performed separately on each element of the vector and a vector of the same length is returned.

If you operate on two vectors, you get an element-wise result:

```
a <- c(1, 5)
b <- c(2, -3)
a * b
```

```
[1]   2 -15
```

Be careful about mixing vectors of different lengths as **R** will generally recycle values:

```
x <- 4:6
y <- c(0, 1)
x * y
```

```
Warning in x * y: longer object length is not a multiple of shorter object
length
```

```
[1] 0 5 0
```

There are a number of common mathematical functions already in **R**:

```
mean(x) # average
```

```
[1] 5
```

```
sum(x)  # summation
```

```
[1] 15
```

```
sd(x)   # Standard deviation
```

```
[1] 1
```

```
  var(x)  # Variance
```

```
[1] 1
```

```
  exp(x)  # Exponential
```

```
[1]  54.59815 148.41316 403.42879
```

```
  sqrt(x) # Square root
```

```
[1] 2.000000 2.236068 2.449490
```

```
  log(x)  # Natural log
```

```
[1] 1.386294 1.609438 1.791759
```

```
  sin(x)  # Trigonometric functions
```

```
[1] -0.7568025 -0.9589243 -0.2794155
```

```
  cos(pi + 1) # R even contains pi, but only does finite arithmetic
```

```
[1] -0.5403023
```

```
  floor(x/2) #The nearest integer below
```

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

```
  ceiling(x/2) #The nearest integer above
```

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

Notice that some of these functions take in a vector as an input and return a scalar, whereas others are vectorized - they take in a vector and return a vector of the same length. Read the documentation (`help(mean)`) or test the function before assuming which way it works.

We can also nest functions.

```
mean(log(floor(x)))
```

```
[1] 1.595831
```

When doing math in **R** or another computing language, be cognizant of the fact that numeric doubles have finite precision - also known as the floating point problem. This can sometimes lead to unexpected results as seen here:

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

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

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

```
[1] FALSE
```

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

```
[1] TRUE
```

The `all.equal` function attempts to workaround this precision so use it if you're concerned about running into this issue. Note that `all.equal` does **not** return logical if it fails:

```
all.equal(sqrt(2)^3, 2)
```

```
[1] "Mean relative difference: 0.2928932"
```

Instead, you should always check if the output of `all.equal` is `TRUE` directly:

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

```
[1] TRUE
```

```
all.equal(sqrt(2)^3, 2) == TRUE
```

```
[1] FALSE
```

The R package system

Much of the utility of **R** is derived from an extensive collection of user and domain-expert contributed packages. Packages are simply a standardized way for people to share *documented* code and data. There are nearly 20,000 officially distributed through the CRAN alone!

Packages are primarily distributed through three sources:

- CRAN
- Github
- Bioconductor

Installing packages

The primary way to install a package is using `install.packages("pkg")`.

```
install.packages("lme4") # the package name should be passed as a character string
```

You can find the default location for your **R** packages using the `.libPaths()` function. If you don't have write permission to this folder, you can set this directory to a personal library instead.

```
.libPaths() ## The default library location
.libPaths("/Users/myname/Rlib") #Create the directory first!
```

To install a package to a personal library use the 'lib' option.

```
install.packages("haven", lib = "/Users/myname/Rlib")
```

Use the above with caution and only when necessary.

If your computer has the necessary tools, packages can also be installed from source by downloading the package file and pointing directly to the source tar ball ('.tgz') or Windows binary.

Using packages in R

Installing a package does not make it available to **R**. There are two ways to use things from a package:

- calling `library("pkg")` to add it to the search path,
- using the `pkg::obj` construction to access a package's exported objects,
- using the `pkg:::obj` to access non-exported objects - use with caution!

These methods are illustrated below using the data set `InstEval` distributed with the 'lme4' package.

```
# install.packages("lme4")
head(InstEval)
```

```
Error in eval(expr, envir, enclos): object 'InstEval' not found
```

```
## Using the pkg::function construction
head(lme4::InstEval)
```

```
  s    d studage lectage service dept y
1 1 1002       2       2       0    2 5
2 1 1050       2       1       1    6 2
3 1 1582       2       2       0    2 5
4 1 2050       2       2       1    3 3
5 2  115       2       1       0    5 2
6 2  756       2       1       0    5 4
```

```
library(lme4)
```

```
Loading required package: Matrix
```

```
head(InstEval)
```

```
  s    d studage lectage service dept y
1 1 1002       2       2       0    2 5
2 1 1050       2       1       1    6 2
3 1 1582       2       2       0    2 5
4 1 2050       2       2       1    3 3
5 2  115       2       1       0    5 2
6 2  756       2       1       0    5 4
```

The `library("pkg")` command adds a package to the search path.

```
search()
```

```
 [1] ".GlobalEnv"        "package:lme4"      "package:Matrix"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "MyFunctions"
[10] "package:methods"   "Autoloads"         "package:base"
```

```
library(foreign)
search()
```

```
 [1] ".GlobalEnv"        "package:foreign"   "package:lme4"
 [4] "package:Matrix"    "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "MyFunctions"       "package:methods"   "Autoloads"
[13] "package:base"
```

```
head(InstEval)
```

```
  s    d studage lectage service dept y
1 1 1002       2       2       0    2 5
2 1 1050       2       1       1    6 2
3 1 1582       2       2       0    2 5
4 1 2050       2       2       1    3 3
5 2  115       2       1       0    5 2
6 2  756       2       1       0    5 4
```

To remove a library from the search path use `detach("package:pkg", unload = TRUE)`.

```
detach(package:foreign, unload = TRUE)
search()
```

```
 [1] ".GlobalEnv"        "package:lme4"      "package:Matrix"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "MyFunctions"
[10] "package:methods"   "Autoloads"         "package:base"
```

Input and output

R is primarily an *in-memory* language, meaning it is designed to work with objects stored in working memory (i.e. *RAM*) rather than on disk. Therefore, it is essential to know how to read and write data from disk.

Native R Binaries

There are two common formats for writing binaries containing native R objects: `.RData` and `.rds`.

To save one or more R objects to a file, use `save()`.

```
df_desc <- "The lme4::InstEval data."
save(InstEval, df_desc, file = "data/InstEval.RData")
```

To restore these objects to the global environment use `load()`.

```
rm(list = ls())
ls()
```

```
character(0)
```

```
load("data/InstEval.RData")
ls()
```

```
[1] "df_desc"  "InstEval"
```

Serialized R Data

Generally, it is best to use `save()` and `load()` for R objects. However, there are times when it can be helpful to save the *data* an R object contains without also saving its name. In these cases, you can use `saveRDS()` and `readRDS()`.

```
saveRDS(InstEval, file = "data/InstEval.rds")
df <- readRDS("data/InstEval.rds")
```

Style note

You may come across the extensions `.Rdata` or `.rda` from time to time. These are generally synonyms for `.RData` and can usually be loaded using `load()`. However, the standard is to use `.RData` and this should be considered course style.

Delimited Data

Data is commonly shared as flat (maybe compressed) text files often delimited by commas (e.g. `.csv`), tab or `'\t'` (e.g. `.tsv`, `.tab`, `.txt`), or one or more white-space characters (e.g. `.data`, `.txt`).

In the base `R` packages, these can be read using `read.table` and its wrappers like `read.csv`. To read the file at `../data/recs2015_public_v4.csv`, use `read.table()` and assign the input to an object.

```
recs <- read.table("data/recs2015_public_v4.csv", sep = ",", header = TRUE)
dim(recs)
class(recs)
```

These functions return `data.frames` which are special *lists* whose members all have the same length (i.e. the number of rows.)

Likewise, you can write delimited files using `write.table` or `write.csv`.

```
write.csv(InstEval, file = "../../data/InstEval.txt",
            row.names = FALSE)
```

Other formats

You may find the following packages useful for reading and writing data from other formats:

- `readxl` for reading Excel files
- `haven` for reading other common data formats
- `foreign` an alternative to `haven` that supports additional formats.

Vectors

Vectors are the basic building blocks of many R data structures including rectangular data structures or *data.frames* most commonly used for analysis.

There are two kinds of vectors: *atomic vectors* and *lists*. Every vector has two defining properties aside from the values it holds:

- A *type* (or *mode*) referring to what type of data it holds, and
- A *length* referring to the number of elements it points to.

Use `typeof()` to determine a vector's type and `length()` to determine its length.

In **R**, *scalars* are just vectors with length one.

Atomic Vectors

The elements in atomic vectors must all be of the same *type*. Here are the most important types:

- Logical: `TRUE`, `FALSE`
- Integer: `1L`
- Double: `2.4`
- Character: `"some words"`

Two less commonly used types are *raw* and *complex*. The *mode* of a vector of integers or doubles is *numeric*, the *mode* of other types is the same as the type. In most cases, double is the default and easier type to use.

There are three special scalars: `NA`, `NaN`, and `Inf`.

`NaN` represents 0/0 and `Inf` represents infinity (for example, c/0).

```
0/0
```

```
[1] NaN
```

```
3/0
```

```
[1] Inf
```

```
typeof(NaN)
```

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

```
typeof(Inf)
```

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

Both are doubles, so can be included in a numeric vector:

```
x <- c(3, Inf, NaN, 4)
x
```

```
[1]   3 Inf NaN   4
```

The third special scalar, `NA`, represents missing data and is logical, not a numeric - but can still be stored in a numeric vector:

```
typeof(NA)
```

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

```
x <- c(NA, 2)
x
```

```
[1] NA  2
```

Other logical are converted to numeric:

```
x <- c(TRUE, FALSE, 3)
x
```

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

Empty vectors

To create new vectors without assigning values to them use `c()` or `vector(mode, length)`.

```
cvec <- c()
cvec
```

```
NULL
```

```
length(cvec)
```

```
[1] 0
```

```
typeof(cvec)
```

```
[1] "NULL"
```

```r
vvec <- vector("character", length = 4)
length(vvec)
```

```
[1] 4
```

```r
typeof(vvec)
```

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

```r
vvec
```

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

Subsetting

Access elements of a vector `x` using `x[]`. You can put numeric values into the square brackets to extract the requested positions. A negative (`-`) can eliminate the requested positions. Note that R vectors start at position 1.

```r
x <- c(6, 2, 1, 0, NA)
x[3]
```

```
[1] 1
```

```r
x[3:5]
```

```
[1]  1  0 NA
```

```r
x[c(1, 3, 5)]
```

```
[1]  6  1 NA
```

```r
x[-4]
```

```
[1]  6  2  1 NA
```

```
x[-(2:4)]
```

```
[1]  6 NA
```

```
x[x > 1]
```

```
[1]  6  2 NA
```

The last uses a logical statement that generates a series of logical values and drops any `FALSE` entries.

```
x > 1
```

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

Lists

Lists provide a more flexible data structure which can hold elements of multiple types, including other vectors. This can be useful for bringing together multiple object types into a single place.

Lists are R's most flexible object structure.

```
x <- list(c(3, 4, 5), c("a", "b"), head(lme4::InstEval))
x
```

```
[[1]]
[1] 3 4 5

[[2]]
[1] "a" "b"

[[3]]
  s    d studage lectage service dept y
1 1 1002       2       2       0    2 5
2 1 1050       2       1       1    6 2
3 1 1582       2       2       0    2 5
4 1 2050       2       2       1    3 3
5 2  115       2       1       0    5 2
6 2  756       2       1       0    5 4
```

The `length` of a list is the number of elements inside a list, not the length of the elements themselves.

```
length(x)
```

```
[1] 3
```

You can obtain information about the elements within the list via the `sapply` function:

```
sapply(x, typeof)
```

```
[1] "double"    "character" "list"
```

We will talk more about the "`apply`" family of functions at a later point.

Subsetting lists

Lists can be subset via `[]` as with vectors, or with the double brackets, `[[]]`.

A single bracket returns another list:

```
x[1]
```

```
[[1]]
[1] 3 4 5
```

```
length(x[1])
```

```
[1] 1
```

A double bracket returns the object inside the list

```
x[[1]]
```

```
[1] 3 4 5
```

```
length(x[[1]])
```

```
[1] 3
```

You can stack the subsetting brackets to pull individual items out of an object inside a list:

```
x[[1]][3:4]
```

```
[1]  5 NA
```

Attributes

Attributes are metadata associated with a vector.

Some of the most important and commonly used attributes we will learn about are:

- names
- class
- dim

Each of these can be accessed and set using functions of the same name.

Names

There are a few ways to create a vector with names.

```
## assign with names()
x <- 1:3
names(x) <- c("Uno", "Dos", "Tres")
x
```

```
 Uno  Dos Tres
   1    2    3
```

```
## quoted names
x <- c( "Uno" = 1, "Dos" = 2, "Tres" = 3 )
x
```

```
 Uno  Dos Tres
   1    2    3
```

```
## bare names - names cannot contain spaces or most special characters
x <- c( Uno = 1, Dos = 2, Tres = 3)
x
```

```
 Uno  Dos Tres
   1    2    3
```

```
names(x)
```

```
[1] "Uno"  "Dos"  "Tres"
```

class

The *class* of an object plays an important role in R's *S3* object oriented system and determines how an object is treated by various functions.

The class of a vector is generally the same as its *mode* or *type*.

```
class(FALSE)
```

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

```
class(0L)
```

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

```
class(1)
```

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

```
class("Two")
```

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

```
class(list())
```

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

Note the difference after we change the *class* of x.

```
x
```

```
 Uno  Dos Tres
   1    2    3
```

```
class(x) <- "character"
x
```

```
 Uno  Dos Tres
 "1"  "2"  "3"
```

It's generally better to use explicit conversion functions such as `as.character()` to change an object's class as simplifying modifying the attribute will not guarantee the object is a valid member of that class.

dim

We use `dim()` to access or set the *dimensions* of an object. Three special classes where dimensions matter are *matrix*, *array*, and *data.frame*. We will examine just the first two here.

Both matrices and arrays are really just long vectors with this additional *dim* attribute.

```
# Matrix class
x <- matrix(1:10, nrow = 5, ncol = 2)
dim(x)
```

```
[1] 5 2
```

```
class(x)
```

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

```
dim(x) = c(2, 5)
x
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

The above demonstrates that **R** matrices are stored in *column-major order*. The matrix class is the two-dimensional special cases of the array class.

```
dim(x) <- c(5, 1, 2)
class(x)
```

```
[1] "array"
```

```
x
```

```
, , 1

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

, , 2

     [,1]
[1,]    6
[2,]    7
[3,]    8
[4,]    9
[5,]   10
```

```
dim(x) <- c(5, 2, 1)
x
```

```
, , 1

     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

```
class(x)
```

```
[1] "array"
```

Arbitrary attributes

See a list of all attributes associated with a vector using `attributes()`. Access and assign specific attributes using `attr()`.

```
# Assign an a new attribute color
attr(x, "color") <- "green"
attributes(x)
```

```
$dim
[1] 5 2 1

$color
[1] "green"
```

```
class(attributes(x))
```

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

```
# Access or assign specific attributes
attr(x, "dim")
```

```
[1] 5 2 1
```

```
  attr(x, "dim") <- c(5, 2)
  class(x)
```

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

```
  x
```

```
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
attr(,"color")
[1] "green"
```

```
  # Assign NULL to remove attributes
  attr(x, "color") <- NULL
  attributes(x)
```

```
$dim
[1] 5 2
```

The *data.frame* class

The *data.frame* class in **R** is a *list* whose elements (columns) all have the same length. This provide a major benefit over a matrix, as each object in the list can have it's own type. In other words, while a matrix must have all entries as numeric, a data.frame can have a numeric column next to a character column.

A *data.frame* has attributes *names* with elements of the list constituting its columns, *row.names* with values uniquely identifying each row, and also has *class data.frame*.

Construct a new *data.frame* using `data.frame()`. Here is an example.

```
  df = data.frame(id = 1:10,
                  group = sample(0:1, 10, replace = TRUE),
                  var1 = rnorm(10),
                  var2 = seq(0, 1, length.out = 10),
```

```
                  var3 = rep(c("a", "b"), each = 5))
  names(df)
```

```
[1] "id"    "group" "var1"  "var2"  "var3"
```

```
  dim(df)
```

```
[1] 10  5
```

```
  length(df)
```

```
[1] 5
```

```
  nrow(df)
```

```
[1] 10
```

We can access the values of a data frame both like a list:

```
  df$id # This is the most common way of accessing columns in a data.frame
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
  df[["var3"]]
```

```
 [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b"
```

or like a matrix:

```
  df[1:5,]
```

```
  id group       var1      var2 var3
1  1     0 -0.8204684 0.0000000    a
2  2     1  0.4874291 0.1111111    a
3  3     0  0.7383247 0.2222222    a
4  4     0  0.5757814 0.3333333    a
5  5     1 -0.3053884 0.4444444    a
```

```
df[, "var2"]
```

```
[1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
[8] 0.7777778 0.8888889 1.0000000
```

This works because the subset operator `[` has behavior that depends on the class of the object being subset.

Logicals

R has three reserved words of type 'logical':

```
typeof(TRUE)
```

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

```
typeof(FALSE)
```

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

```
typeof(NA)
```

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

```
TRUE == T
```

```
[1] TRUE
```

```
FALSE == F
```

```
[1] TRUE
```

While `T` and `F` are equivalent to `TRUE` and `FALSE`, **you should never use `T` or `F`**. The primary reason being that you can name variables `T` or `F`, but not `TRUE` or `FALSE`:

```
T <- 2
x <- 14:17
x[c(T, T, T, F)]
```

```
[1] 15 15 15
```

```
x[c(TRUE, TRUE, TRUE, FALSE)]
```

```
[1] 14 15 16
```

```
TRUE <- 3
```

```
Error in TRUE <- 3: invalid (do_set) left-hand side to assignment
```

Boolean comparisons

Boolean operators are useful for generating values conditionally on other values. Here are the basics:

Operator	Meaning
`==`	equal
`!=`	not equal
`>`, `>=`	greater than (or equal to)
`<`, `<=`	less than (or equal to)
`&`	vectorized AND
`|`	vectorized OR
`&&`	scalar AND
`||`	scalar OR
`!`	negation (`!TRUE == FALSE` and `!FALSE == TRUE`)
`any()`	are ANY of the elements true
`all()`	are ALL of the elements true

Logicals are created by Boolean comparisons:

```
2*3 == 6     # test equality with ==
```

```
[1] TRUE
```

```
2+2 != 5     # use != for 'not equal'
```

```
[1] TRUE
```

```
sqrt(70) > 8   # comparison operators: >, >=, <, <=
```

```
[1] TRUE
```

```
sqrt(64) >= 8
```

```
[1] TRUE
```

```
!(2 == 3)        # Use not to negate or 'flip' a logical - don't forget PEMBAS
```

```
[1] TRUE
```

Comparison operators are vectorized:

```
1:10 > 5
```

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

You can combine operators using "and" (`&`) or "or" (`|`):

```
2 + 2 == 4 | 2 + 2 == 5 # An or statement asks if either statement is true
```

```
[1] TRUE
```

```
2 + 2 == 4 & 2 + 2 == 5 # And requires both to be true
```

```
[1] FALSE
```

"and" and "or" are vectorized:

```r
c(TRUE, FALSE) | c(FALSE, FALSE)
```

```
[1]  TRUE FALSE
```

The double version of and (`&&`) and (`||`) will throw an error if you attempt to use it with a vector:

```r
c(TRUE, FALSE) || c(FALSE, FALSE)
```

```
Error in c(TRUE, FALSE) || c(FALSE, FALSE): 'length = 2' in coercion to 'logical(1)'
```

If both sides of the statement have length 1, `&` and `&&` (and `|` and `||`) function identically.

It is recommended to use `&&` and `||` when you want to ensure that the objects passed into the conditional are length 1. (Note that earlier versions of R did not produce errors in this case, yielding undesired results. Older documents you may find may discuss the distinction between the single and double versions differently.)

Use `any` or `all` to efficiently check for the presence of any `TRUE` or `FALSE`.

```r
any(c(TRUE, FALSE, FALSE))
```

```
[1] TRUE
```

```r
all(c(TRUE, FALSE, FALSE))
```

```
[1] FALSE
```

`na.rm` argument

Many functions take an optional `na.rm` argument which determines whether `NA` is ignored during the functions execution.

```r
all(c(TRUE, NA))
```

```
[1] NA
```

```
all(c(TRUE, NA), na.rm = TRUE)
```

```
[1] TRUE
```

This also applies to `NA`s in numerical objects:

```
mean(c(2, 5, 7, NA, 4))
```

```
[1] NA
```

```
mean(c(2, 5, 7, NA, 4), na.rm = TRUE)
```

```
[1] 4.5
```

While `na.rm` is the most common version of this, sometimes you'll see a different argument with a similar functionality, for example:

```
table(c(2, 4, 5, NA, 6))
```

```
2 4 5 6
1 1 1 1
```

```
table(c(2, 4, 5, NA, 6), useNA = "always")
```

```
   2    4    5    6 <NA>
   1    1    1    1    1
```

```
table(c(2, 4, 5, NA, 6), useNA = "ifany")
```

```
   2    4    5    6 <NA>
   1    1    1    1    1
```

```r
table(c(2, 4, 5, 6), useNA = "always")
```

```
   2    4    5    6 <NA>
   1    1    1    1    0
```

```r
table(c(2, 4, 5, 6), useNA = "ifany")
```

```
2 4 5 6
1 1 1 1
```

Finally, modeling commands (such as `lm` for linear regression models) often take an `na.action` argument, which takes in one of:

- `na.omit`, the default, drop any observations with `NA`
- `na.fail`, if there's an `NA` in the data passed into the model, produce an error
- `na.pass`, do nothing special to the data, which may or may not lead to an error or unexpected results from the model.

Using which

The `which()` function returns the elements of a logical vector that return true:

```r
which((1:5)^2 > 10)
```

```
[1] 4 5
```

A combination of `which` and logicals can be used to subset data frames:

```r
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
mtcars[which(mtcars$mpg>30), ]
```

```
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Fiat 128      32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic   30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9  4 71.1  65 4.22 1.835 19.90  1  1    4    1
Lotus Europa  30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

Functions

Functions in **R** operate as `name(arguments)` - a name of a function, and a comma-separated list of arguments. Functions are simply collections of commands that do something. Functions **arguments** can be used to specify *which* objects to operate on and *what* values of parameters are used. You can use `help(func)` to see what a function is used for and what arguments it expects, i.e. `help(mean)`.

Arguments

Functions will often have multiple arguments. Some arguments have default values, others do not. All arguments without default values must be passed to a function. Arguments can be passed by name or position. For instance,

```
x <- runif(n = 5, min = 0, max = 1)
y <- runif(5, 0, 1)
z <- runif(5)
round(cbind(x, y, z), 1)
```

```
      x   y   z
[1,] 0.5 0.7 0.8
[2,] 0.6 0.8 0.6
[3,] 0.5 0.1 0.8
[4,] 0.2 0.7 0.6
[5,] 0.8 0.4 0.5
```

all generate 5 numbers from a Uniform(0,1) distribution.

Arguments passed by name need not be in order:

```
w <- runif(min = 0, max = 1, n = 5)
```

```
round(rbind(x, y, z, w = w), 1)
```

```
  [,1] [,2] [,3] [,4] [,5]
x  0.5  0.6  0.5  0.2  0.8
y  0.7  0.8  0.1  0.7  0.4
z  0.8  0.6  0.8  0.6  0.5
w  0.8  0.0  0.5  0.7  0.7
```

General style advice is to: 1. Pass "common" arguments (no more than 2-3) by position and do not re-order them. 2. Once you pass an argument by name, all further arguments to the right should also be by name

```
# Good style
runif(5, 0, 1)
```

```
[1] 0.47761962 0.86120948 0.43809711 0.24479728 0.07067905
```

```
runif(5, min = 0, max = 1)
```

```
[1] 0.09946616 0.31627171 0.51863426 0.66200508 0.40683019
```

```
# Bad style
runif(n = 5, min = 0, 1)
```

```
[1] 0.9128759 0.2936034 0.4590657 0.3323947 0.6508705
```

```
runif(min = 0, max = 1, n = 5) # despite it's use above
```

```
[1] 0.25801678 0.47854525 0.76631067 0.08424691 0.87532133
```

This becomes more important with commands such as `lm` for linear regression fits which take in 13 arguments, of which arguably 2 are common.

The ... argument

Some functions include an optional `...` argument. This is a special argument that accepts any number of arguments with any names (not conflicting with established names). This is used for two purposes:

- A function that takes in an arbitary number of objects, e.g. `sum` (see `help(sum)`).
- To pass additional arguments to a lower-level function. We'll see examples of this usage with the `apply` family of functions in a future set of notes.

In the first case, objects passed into `...` may be named. In the second case, objects **must** be named for the lower-level functions to identify them.

Writing Functions

You can create your own functions in **R**. Use functions for tasks that you repeat often in order to make your scripts more easily readable and modifiable. A good rule of thumb is never to copy and paste more than twice; use a function instead. It can also be a good practice to use functions to break complex processes into parts, especially if these parts are used with control flow statements such as loops or conditionals.

```
# function to compute z-scores
zScore1 <- function(x) {
  #inputs: x - a numeric vector
  #outputs: the z-scores for x
  xbar <- mean(x)
  s <- sd(x)
  z <- (x - xbar) / s
  return(z)
}
```

The return statement is not strictly necessary, but can make complex functions more readable. If excluded, the result of the last line is returned.

```
x = rnorm(10, 3, 1) ## generate some normally distributed values
round(cbind(x = x, z = zScore1(x)), 1)
```

```
      x    z
 [1,] 2.6 -0.9
 [2,] 2.6 -0.8
 [3,] 2.9 -0.3
 [4,] 4.1  1.6
 [5,] 3.8  1.1
```

```
 [6,] 2.8 -0.5
 [7,] 2.7 -0.6
 [8,] 3.7  1.0
 [9,] 3.6  0.7
[10,] 2.3 -1.3
```

Default parameters

We can set default values for parameters using the construction `parameter = xx` in the function definition.

```
# function to compute z-scores
zScore2 = function(x, na.rm = TRUE){
  xbar <- mean(x, na.rm = na.rm)
  s <- sd(x, na.rm = na.rm)
  (x - xbar) / s
}
```

```
x = c(NA, x, NA)
round(cbind(x,
            z1 = zScore1(x),
            z2 = zScore2(x)
          , z2FALSE = zScore2(x, FALSE)), 1)
```

```
       x z1   z2 z2FALSE
 [1,]  NA NA   NA      NA
 [2,] 2.6 NA -0.9      NA
 [3,] 2.6 NA -0.8      NA
 [4,] 2.9 NA -0.3      NA
 [5,] 4.1 NA  1.6      NA
 [6,] 3.8 NA  1.1      NA
 [7,] 2.8 NA -0.5      NA
 [8,] 2.7 NA -0.6      NA
 [9,] 3.7 NA  1.0      NA
[10,] 3.6 NA  0.7      NA
[11,] 2.3 NA -1.3      NA
[12,]  NA NA   NA      NA
```

Documenting functions

You'll see above in `zScore` that we introduced two comments: `#inputs` and `#outputs`. These are not anything formal, but they do represent good practice - document what goes into a function, and document what comes out of a function.

For more formal documentation, the **roxygen2** package comes in handy. It is primarily used during the creation of custom R packages (which we will cover later) but provides a good framework for documenting any function.

You can manually write roxygen documentation (any comments starting with `#'` immediately preceeding a function defintion) or use the Code -> Insert Roxygen Skeleton option in RStudio.

```
#' Function to create z-scores
#'
#' @param x a numeric vector
#' @return The z-score for `x`
zScore1 <- function(x) {
  xbar <- mean(x)
  s <- sd(x)
  z <- (x - xbar) / s
  return(z)
}
```

The `@keywords` are called "tags". Each argument should have a corresponding `@param` tag and there should be a `@return` tag.Any lines above these can be used to further describe what's going on in the function.

The skeleton created by RStudio has an `@export`; this is only relevant during package creation and can be ignored (or deleted) when working interactively. It also has an `@examples` which you can use if needed.

Formally, during package creation, these roxygen comments are used to generate the documentation found when using `help()`. You can see a full list of tags here but almost all the rest are only relevant during package creation.

Sanitizing input

It's usually helpful to make sure the input to R functions are what you expect.

```
#' Function to create z-scores
#'
#' @param x a numeric vector
#' @return The z-score for `x`
zScore2 <- function(x) {
```

```
  if (!is.numeric(x)) {
    stop("x must be numeric")
  }

  xbar <- mean(x)
  s <- sd(x)
  z <- (x - xbar) / s
  return(z)
}

zScore1(c("a", "b"))
```

```
Warning in mean.default(x): argument is not numeric or logical: returning NA
```

```
Warning in var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm =
na.rm): NAs introduced by coercion
```

```
Error in x - xbar: non-numeric argument to binary operator
```

```
zScore2(c("a", "b"))
```

```
Error in zScore2(c("a", "b")): x must be numeric
```

While both error appropriarely, the second provides a much more useful output. You could of course do something more useful than creating an error.

```
#' Function to create z-scores
#'
#' @param x a numeric vector
#' @return The z-score for `x`
zScore3 <- function(x) {
  if (!is.numeric(x)) {
    warning("Input must be numeric. Attempting to convert to numeric...")
    x <- as.numeric(x)
    if (all(is.na(x))) {
      stop("Conversion to numeric failed")
    }
  }
```

```
  xbar <- mean(x)
  s <- sd(x)
  z <- (x - xbar) / s
  return(z)
}

zScore3(c("1", "2"))
```

```
Warning in zScore3(c("1", "2")): Input must be numeric. Attempting to convert
to numeric...
```

```
[1] -0.7071068  0.7071068
```

```
zScore3(c("a", "b"))
```

```
Warning in zScore3(c("a", "b")): Input must be numeric. Attempting to convert
to numeric...
```

```
Warning in zScore3(c("a", "b")): NAs introduced by coercion
```

```
Error in zScore3(c("a", "b")): Conversion to numeric failed
```

Scope

Scoping refers to how **R** looks up the value associated with an object referred to by name. There are two types of scoping – lexical and dynamic – but we will concern ourselves only with lexical scoping here. There are four keys to understanding scoping:

- environments
- name masking
- variables vs functions
- dynamic lookup and lazy evaluation.

An environment can be thought of as context in which a name for an object makes sense. Each time a function is called, it generates a new environment for the computation.

Consider the following examples:

```
rm(list = ls())
x <- 3
```

```
f1 <- function(){
  print(paste("First call to x:", x))
  x <- 5
  print(paste("Second call to x:", x))
}
f1()
```

```
[1] "First call to x: 3"
[1] "Second call to x: 5"
```

```
x
```

```
[1] 3
```

Name masking refers to *where* and *in what order* **R** looks for object names. When we call `f1` above, **R** first looks in the current environment which happens to be the global environment. The first `print`finds the `x` in the Global Environment and prints it. Next, we redefine `x` and when it looks for it in the second `print`, finds that `x` as it exists within the local environment.

However, despite us changing `x` inside `f1`, after we execute the function, `x` in the Global Environment remains unchanged.

When an environment is created, it gets nested within the current environment referred to as the "parent environment". When an object is referenced we first look in the current environment and move recursively up through parent environments until we find a value bound to that name.

R also uses dynamic lookup, meaning *values are searched for when a function is called*, not when it is *created*.

Finally, lazy evaluation means **R** only evaluates function arguments if and when they are actually used.

```
f2 = function(x){
  #x
  45
}

f2( x=stop("Let's pass an error.") )
```

```
[1] 45
```

Uncomment `x` to see what happens if we evaluate it.

<- vs = for arguments

Consider the following:

```
rm(list = ls())
mean(x = c(5, 2))
```

```
[1] 3.5
```

```
x
```

```
Error in eval(expr, envir, enclos): object 'x' not found
```

Now, look at the version where we use <- instead:

```
rm(list = ls())
mean(x <- c(5, 2))
```

```
[1] 3.5
```

```
x
```

```
[1] 5 2
```

Unlike assigning values to objects, where in almost all situations = and <- are equivalent, when it comes to arguments, = assigns the values to the object *in the function's scope* whereas <- assigns it in the calling environment.

Creating side effects

In general, functions should take in any objects they need as arguments, and return any objects that the need to (whether they are newly created or modified versions). Lists can be handy here to return several different types of objects.

There may be *extremely limited* situations in which you do want a side effect - usually I use this only when debugging an issue. The double arrow, <<-, may be used in this case. Let's replicate `f1` from above, but use the double arrow.

```r
rm(list = ls())
x <- 3
f1b <- function(){
  print(paste("First call to x:", x))
  x <<- 5
  print(paste("Second call to x:", x))
}
f1b()
```

```
[1] "First call to x: 3"
[1] "Second call to x: 5"
```

```r
x
```

```
[1] 5
```

Note that `x` now has the value of 5 *outside of the function*. The double arrow assigns in the so-called "parent environment", the one that contains the current environment. So when we call `x <<- 5` inside the function, the `x` outside the function gets changed.

Resources

Read more about functions here

You can also read much more about functions in Chapter 7 of "The Art of R Programming."

Control Statements

`for` loops

Here is the syntax for a basic for loop in **R**

```r
for (i in 1:10) {
   cat(i, "\n")
}
```

```
1
2
3
4
```

```
5
6
7
8
9
10
```

Note that the loop and the *iterator* are evaluated within the global environment.

```
for (var in names(mtcars)) {
  cat(sprintf("average %s = %4.3f", var, mean(mtcars[, var]) ), "\n")
}
```

```
average mpg = 20.091
average cyl = 6.188
average disp = 230.722
average hp = 146.688
average drat = 3.597
average wt = 3.217
average qsec = 17.849
average vs = 0.438
average am = 0.406
average gear = 3.688
average carb = 2.812
```

```
var
```

```
[1] "carb"
```

while

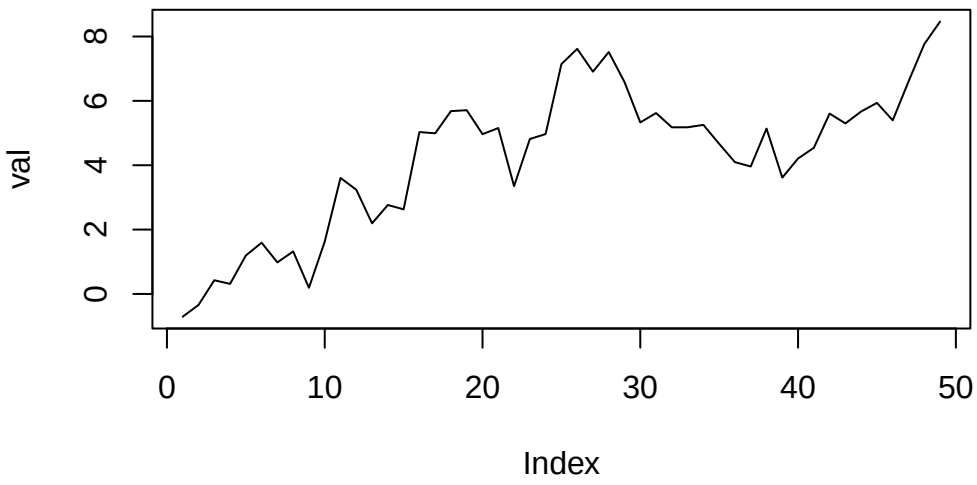
A `while` statement can be useful when you aren't sure how many iterations are needed. Here is an example that takes a random walk and terminates if the value is more than 10 units from 0.

```
max_iter <- 1e3 # always limit the total iterations allowed
val <- vector(mode = "numeric", length = max_iter)
val[1] <- rnorm(1) ## initialize

k <- 1
```

```
while ( abs(val[k]) < 10 && k <= max_iter ) {
  val[k+1] <- val[k] + rnorm(1)
  k <- k + 1
}
val <- val[1:(k-1)]

plot(val, type="l")
```



Note the generation of an empty `val` vector of appropriate length, instead of starting with a scalar and using `c(val, val[k] + rnorm(1))`. It is very inefficient due to *copy on modify* to expand the length of a vector (or the rows/columns of a matrix/data.frame) in a loop. Instead, as here, generate a complete blank object first, and fill it in as you go.

key words

The following key words are useful within loops:

- `break` - break out of the currently executing loop
- `next` - move to the next iteration immediately, without executing the rest of this iteration (`continue` in other languages such as C++)

Here is an example using `next`:

```
for (i in 1:10) {
  if (i %% 2 == 0) {
    next
  }
  cat(i, "\n")
}
```

```
1
3
5
7
9
```

Here is an example using `break`:

```
x <- vector(mode = "numeric", length = 1e1)
for (i in 1:1e1) {
  if (i %% 3 == 0) {
    break
  }
  x[i] <- i
}
print(x[1:10])
```

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

Conditionals

In programming, we often need to execute a piece of code only if some condition is true. Here are some of the **R** tools for doing this.

if statements

The workhorse for conditional execution in **R** is the `if` statement.

```
if (TRUE) {
  print("do something if true")
}
```

```
[1] "do something if true"
```

Occasionally, with short statements it can be idiomatic to include the condition on the same line without the braces:

```
if (TRUE) print("do something if true")
```

```
[1] "do something if true"
```

It is not recommended to use this as it can be harder to understand while reading the code.

Use an `else` to control the flow without separately checking the condition's negation:

```
if (2 + 2 == 5) {
  print("the statement is true")
} else {
  print("the statement is false")
}
```

```
[1] "the statement is false"
```

```
result <- c(4, 5)
report <- ifelse(2 + 2 == result, "true", "false")
report
```

```
[1] "true"  "false"
```

As you can see above, there is also a vectorized `ifelse()` function that can be useful.

For more complex cases, you may want to check multiple conditions:

```
a <- -1
b <- 1

if (a * b > 0) {
  print("Zero is not between a and b")
} else if (a < b) {
  smaller <- a
  larger <- b
} else {
```

```
  smaller <- b
  larger <- a
}

c(smaller, larger)
```

```
[1] -1  1
```