Vectors

The basic data structure in R is the vector. Vectors come in two flavours: atomic vectors and lists. They have three common properties:

- Type, `typeof()`, what it is.
- Length, `length()`, how many elements it contains.
- Attributes, `attributes()`, additional arbitrary metadata.

They differ in the types of their elements: all elements of an atomic vector must be the same type, whereas the elements of a list can have different types.

NB: `is.vector()` does not test if an object is a vector. Instead it returns `TRUE` only if the object is a vector with no attributes apart from names. Use `is.atomic(x) || is.list(x)` to test if an object is actually a vector.

Atomic vectors

There are four common types of atomic vectors that I'll discuss in detail: logical, integer, double (often called numeric), and character. There are two rare types that I will not discuss further: complex and raw.

Atomic vectors are usually created with `c()`, short for combine:

```
dbl_var <- c(1, 2.5, 4.5)
# With the L suffix, you get an integer rather than a double
int_var <- c(1L, 6L, 10L)
# Use TRUE and FALSE (or T and F) to create logical vectors
log_var <- c(TRUE, FALSE, T, F)
chr_var <- c("these are", "some strings")
```

Atomic vectors are always flat, even if you nest `c()`'s:

```
c(1, c(2, c(3, 4)))
#> [1] 1 2 3 4
# the same as
c(1, 2, 3, 4)
#> [1] 1 2 3 4
```

Missing values are specified with `NA`, which is a logical vector of length 1. `NA` will always be coerced to the correct type if used inside `c()`, or you can create `NA`s of a specific type with `NA_real_` (a double vector), `NA_integer_` and `NA_character_`.

Types and tests

Given a vector, you can determine its type with `typeof()`, or check if it's a specific type with an "is" function: `is.character()`, `is.double()`, `is.integer()`,`is.logical()`, or, more generally, `is.atomic()`.

```
int_var <- c(1L, 6L, 10L)
```

```
typeof(int_var)
#> [1] "integer"
is.integer(int_var)
#> [1] TRUE
is.atomic(int_var)
#> [1] TRUE

dbl_var <- c(1, 2.5, 4.5)
typeof(dbl_var)
#> [1] "double"
is.double(dbl_var)
#> [1] TRUE
is.atomic(dbl_var)
#> [1] TRUE
```

NB: `is.numeric()` is a general test for the "numberliness" of a vector and returns `TRUE` for both integer and double vectors. It is not a specific test for double vectors, which are often called numeric.

```
is.numeric(int_var)
#> [1] TRUE
is.numeric(dbl_var)
#> [1] TRUE
```

Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be **coerced** to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

For example, combining a character and an integer yields a character:

```
str(c("a", 1))
#>  chr [1:2] "a" "1"
```

When a logical vector is coerced to an integer or double, `TRUE` becomes 1 and `FALSE` becomes 0. This is very useful in conjunction with `sum()` and `mean()`

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1

# Total number of TRUEs
sum(x)
#> [1] 1

# Proportion that are TRUE
mean(x)
```

```
#> [1] 0.3333333
```

Coercion often happens automatically. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to a double or integer, and most logical operations (`&`, `|`, `any`, etc) will coerce to a logical. You will usually get a warning message if the coercion might lose information. If confusion is likely, explicitly coerce with `as.character()`,`as.double()`, `as.integer()`, or `as.logical()`.

Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using `list()` instead of `c()`:

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

Lists are sometimes called **recursive** vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
is.recursive(x)
#> [1] TRUE
```

`c()` will combine several lists into one. If given a combination of atomic vectors and lists, `c()` will coerce the vectors to lists before combining them. Compare the results of `list()` and `c()`:

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(y)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

The `typeof()` a list is `list`. You can test for a list with `is.list()` and coerce to a list with `as.list()`. You can turn a list

into an atomic vector with `unlist()`. If the elements of a list have different types, `unlist()` uses the same coercion rules as `c()`.

Lists are used to build up many of the more complicated data structures in R. For example, both data frames (described in data frames) and linear models objects (as produced by `lm()`) are lists:

```
is.list(mtcars)
#> [1] TRUE

mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
```

Data frames

A data frame is the most common way of storing data in R, and if used systematicallymakes data analysis easier. Under the hood, a data frame is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has `names()`, `colnames()`, and`rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`; `nrow()`gives the number of rows.

As described in subsetting, you can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).

Creation

You create a data frame using `data.frame()`, which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Beware `data.frame()`'s default behaviour which turns strings into factors. Use`stringAsFactors = FALSE` to suppress this behaviour:

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

Testing and coercion

Because a `data.frame` is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use `class()` or test explicitly with `is.data.frame()`:

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

You can coerce an object to a data frame with `as.data.frame()`:

- A vector will create a one-column data frame.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows as the matrix.

Combining data frames

You can combine data frames using `cbind()` and `rbind()`:

```
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>    x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number and names of columns must match. Use `plyr::rbind.fill()` to combine data frames that don't have the same columns.

It's a common mistake to try and create a data frame by `cbind()`ing vectors together. This doesn't work because `cbind()` will create a matrix unless one of the arguments is already a data frame. Instead use `data.frame()` directly:

```
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
str(bad)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: Factor w/ 2 levels "1","2": 1 2
#>  $ b: Factor w/ 2 levels "a","b": 1 2
good <- data.frame(a = 1:2, b = c("a", "b"),
  stringsAsFactors = FALSE)
str(good)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: int  1 2
#>  $ b: chr  "a" "b"
```

The conversion rules for `cbind()` are complicated and best avoided by ensuring all inputs are of the same type.

Special columns

Since a data frame is a list of vectors, it is possible for a data frame to have a column that is a list:

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```

However, when a list is given to `data.frame()`, it tries to put each item of the list into its own column, so this fails:

```
data.frame(x = 1:3, y = list(1:2, 1:3, 1:4))
#> Error in data.frame(1:2, 1:3, 1:4, check.names = FALSE, stringsAsFactors = TRUE): arguments imply diffe
ring number of rows: 2, 3, 4
```

A workaround is to use `I()`, which causes `data.frame()` to treat the list as one unit:

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(dfl)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y:List of 3
#>   ..$ : int  1 2
#>   ..$ : int  1 2 3
#>   ..$ : int  1 2 3 4
#>   ..- attr(*, "class")= chr "AsIs"
dfl[2, "y"]
#> [[1]]
#> [1] 1 2 3
```

`I()` adds the `AsIs` class to its input, but this can usually be safely ignored.

Similarly, it's also possible to have a column of a data frame that's a matrix or array, as long as the number of rows matches the data frame:

```
dfm <- data.frame(x = 1:3, y = I(matrix(1:9, nrow = 3)))
str(dfm)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: 'AsIs' int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
dfm[2, "y"]
#>      [,1] [,2] [,3]
```

```
#> [1,]    2    5    8
```

Use list and array columns with caution: many functions that work with data frames assume that all columns are atomic vectors.

Lexical scoping

Scoping is the set of rules that govern how R looks up the value of a symbol. In the example below, scoping is the set of rules that R applies to go from the symbol `x` to its value `10`:

```
x <- 10
x
#> [1] 10
```

Understanding scoping allows you to:

- build tools by composing functions, as described in functional programming.
- overrule the usual evaluation rules and do non-standard evaluation, as described in non-standard evaluation.

R has two types of scoping: **lexical scoping**, implemented automatically at the language level, and **dynamic scoping**, used in select functions to save typing during interactive analysis. We discuss lexical scoping here because it is intimately tied to function creation. Dynamic scoping is described in more detail in scoping issues.

Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called. With lexical scoping, you don't need to know how the function is called to figure out where the value of a variable will be looked up. You just need to look at the function's definition.

The "lexical" in lexical scoping doesn't correspond to the usual English definition ("of or relating to words or the vocabulary of a language as distinguished from its grammar and construction") but comes from the computer science term "lexing", which is part of the process that converts code represented as text to meaningful pieces that the programming language understands.

There are four basic principles behind R's implementation of lexical scoping:

- name masking
- functions vs. variables
- a fresh start
- dynamic lookup

You probably know many of these principles already, although you might not have thought about them explicitly. Test your knowledge by mentally running through the code in each block before looking at the answers.

Name masking

The following example illustrates the most basic principle of lexical scoping, and you should have no problem predicting the output.

```
f <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
```

```
f()
rm(f)
```

If a name isn't defined inside a function, R will look one level up.

```
x <- 2
g <- function() {
  y <- 1
  c(x, y)
}
g()
rm(x, g)
```

The same rules apply if a function is defined inside another function: look inside the current function, then where that function was defined, and so on, all the way up to the global environment, and then on to other loaded packages. Run the following code in your head, then confirm the output by running the R code.

```
x <- 1
h <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
h()
rm(x, h)
```

The same rules apply to closures, functions created by other functions. Closures will be described in more detail in functional programming; here we'll just look at how they interact with scoping. The following function, `j()`, returns a function. What do you think this function will return when we call it?

```
j <- function(x) {
  y <- 2
  function() {
    c(x, y)
  }
}
k <- j(1)
k()
rm(j, k)
```

This seems a little magical (how does R know what the value of `y` is after the function has been called). It works because `k` preserves the environment in which it was defined and because the environment includes the value

of `y`. Environments gives some pointers on how you can dive in and figure out what values are stored in the environment associated with each function.

Functions vs. variables

The same principles apply regardless of the type of associated value — finding functions works exactly the same way as finding variables:

```
l <- function(x) x + 1
m <- function() {
  l <- function(x) x * 2
  l(10)
}
m()
#> [1] 20
rm(l, m)
```

For functions, there is one small tweak to the rule. If you are using a name in a context where it's obvious that you want a function (e.g., `f(3)`), R will ignore objects that are not functions while it is searching. In the following example `n` takes on a different value depending on whether R is looking for a function or a variable.

```
n <- function(x) x / 2
o <- function() {
  n <- 10
  n(n)
}
o()
#> [1] 5
rm(n, o)
```

However, using the same name for functions and other objects will make for confusing code, and is generally best avoided.

A fresh start

What happens to the values in between invocations of a function? What will happen the first time you run this function? What will happen the second time? (If you haven't seen `exists()` before: it returns `TRUE` if there's a variable of that name, otherwise it returns `FALSE`.)

```
j <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
```

```
  print(a)
}
j()
rm(j)
```

You might be surprised that it returns the same value, `1`, every time. This is because every time a function is called, a new environment is created to host execution. A function has no way to tell what happened the last time it was run; each invocation is completely independent. (We'll see some ways to get around this in mutable state.)

Dynamic lookup

Lexical scoping determines where to look for values, not when to look for them. R looks for values when the function is run, not when it's created. This means that the output of a function can be different depending on objects outside its environment:

```
f <- function() x
x <- 15
f()
#> [1] 15

x <- 20
f()
#> [1] 20
```

You generally want to avoid this behaviour because it means the function is no longer self-contained. This is a common error — if you make a spelling mistake in your code, you won't get an error when you create the function, and you might not even get one when you run the function, depending on what variables are defined in the global environment.

One way to detect this problem is the `findGlobals()` function from `codetools`. This function lists all the external dependencies of a function:

```
f <- function() x + 1
codetools::findGlobals(f)
#> [1] "+" "x"
```

Another way to try and solve the problem would be to manually change the environment of the function to the `emptyenv()`, an environment which contains absolutely nothing:

```
environment(f) <- emptyenv()
f()
#> Error in f(): could not find function "+"
```

This doesn't work because R relies on lexical scoping to find *everything*, even the `+`operator. It's never possible to make a function completely self-contained because you must always rely on functions defined in base R or other packages.

You can use this same idea to do other things that are extremely ill-advised. For example, since all of the standard

operators in R are functions, you can override them with your own alternatives. If you ever are feeling particularly evil, run the following code while your friend is away from their computer:

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
#> [36] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
rm("(")
```

This will introduce a particularly pernicious bug: 10% of the time, 1 will be added to any numeric calculation inside parentheses. This is another good reason to regularly restart with a clean R session!

Every operation is a function call

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call."
>
> — John Chambers

The previous example of redefining `(` works because every operation in R is a function call, whether or not it looks like one. This includes infix operators like `+`, control flow operators like `for`, `if`, and `while`, subsetting operators like `[]` and `$`, and even the curly brace `{`. This means that each pair of statements in the following example is exactly equivalent. Note that `` ` ``, the backtick, lets you refer to functions or variables that have otherwise reserved or illegal names:

```
x <- 10; y <- 5
x + y
#> [1] 15
`+`(x, y)
#> [1] 15

for (i in 1:2) print(i)
#> [1] 1
#> [1] 2
`for`(i, 1:2, print(i))
#> [1] 1
#> [1] 2

if (i == 1) print("yes!") else print("no.")
#> [1] "no."
`if`(i == 1, print("yes!"), print("no."))
#> [1] "no."

x[3]
#> [1] NA
`[`(x, 3)
#> [1] NA

{ print(1); print(2); print(3) }
#> [1] 1
#> [1] 2
#> [1] 3
`{`(print(1), print(2), print(3))
```

```
#> [1] 1
#> [1] 2
#> [1] 3
```

It is possible to override the definitions of these special functions, but this is almost certainly a bad idea. However, there are occasions when it might be useful: it allows you to do something that would have otherwise been impossible. For example, this feature makes it possible for the `dplyr` package to translate R expressions into SQL expressions. Domain specific languages uses this idea to create domain specific languages that allow you to concisely express new concepts using existing R constructs.

It's more often useful to treat special functions as ordinary functions. For example, we could use `sapply()` to add 3 to every element of a list by first defining a function `add()`, like this:

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
#>  [1]  4  5  6  7  8  9 10 11 12 13
```

But we can also get the same effect using the built-in `+` function.

```
sapply(1:5, `+`, 3)
#> [1] 4 5 6 7 8
sapply(1:5, "+", 3)
#> [1] 4 5 6 7 8
```

Note the difference between `` `+` `` and `"+"`. The first one is the value of the object called `+`, and the second is a string containing the character `+`. The second version works because `sapply` can be given the name of a function instead of the function itself: if you read the source of `sapply()`, you'll see the first line uses `match.fun()` to find functions given their names.

A more useful application is to combine `lapply()` or `sapply()` with subsetting:

```
x <- list(1:3, 4:9, 10:12)
sapply(x, "[", 2)
#> [1]  2  5 11

# equivalent to
sapply(x, function(x) x[2])
#> [1]  2  5 11
```

Remembering that everything that happens in R is a function call will help you inmetaprogramming.

Function arguments

It's useful to distinguish between the formal arguments and the actual arguments of a function. The formal arguments are a property of the function, whereas the actual or calling arguments can vary each time you call the function. This

section discusses how calling arguments are mapped to formal arguments, how you can call a function given a list of arguments, how default arguments work, and the impact of lazy evaluation.

Calling functions

When calling a function you can specify arguments by position, by complete name, or by partial name. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

```
f <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(f(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(f(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Can abbreviate long argument names:
str(f(2, 3, a = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# But this doesn't work because abbreviation is ambiguous
str(f(1, 3, b = 1))
#> Error in f(1, 3, b = 1): argument 3 matches multiple formal arguments
```

Generally, you only want to use positional matching for the first one or two arguments; they will be the most commonly used, and most readers will know what they are. Avoid using positional matching for less commonly used arguments, and only use readable abbreviations with partial matching. (If you are writing code for a package that you want to publish on CRAN you can not use partial matching, and must use complete names.) Named arguments should always come after unnamed arguments. If a function uses `...` (discussed in more detail below), you can only specify arguments listed after `...` with their full name.

These are good calls:

```
mean(1:10)
mean(1:10, trim = 0.05)
```

This is probably overkill:

```
mean(x = 1:10)
```

And these are just confusing:

```
mean(1:10, n = T)
mean(1:10, , FALSE)
mean(1:10, 0.05)
mean(, TRUE, x = c(1:10, NA))
```

Calling a function given a list of arguments

Suppose you had a list of function arguments:

```
args <- list(1:10, na.rm = TRUE)
```

How could you then send that list to `mean()`? You need `do.call()`:

```
do.call(mean, list(1:10, na.rm = TRUE))
#> [1] 5.5
# Equivalent to
mean(1:10, na.rm = TRUE)
#> [1] 5.5
```

Default and missing arguments

Function arguments in R can have default values.

```
f <- function(a = 1, b = 2) {
  c(a, b)
}
f()
#> [1] 1 2
```

Since arguments in R are evaluated lazily (more on that below), the default value can be defined in terms of other arguments:

```
g <- function(a = 1, b = a * 2) {
  c(a, b)
}
g()
#> [1] 1 2
g(10)
```

```
#> [1] 10 20
```

Default arguments can even be defined in terms of variables created within the function. This is used frequently in base R functions, but I think it is bad practice, because you can't understand what the default values will be without reading the complete source code.

```
h <- function(a = 1, b = d) {
  d <- (a + 1) ^ 2
  c(a, b)
}
h()
#> [1] 1 4
h(10)
#> [1]  10 121
```

You can determine if an argument was supplied or not with the `missing()` function.

```
i <- function(a, b) {
  c(missing(a), missing(b))
}
i()
#> [1] TRUE TRUE
i(a = 1)
#> [1] FALSE  TRUE
i(b = 2)
#> [1]  TRUE FALSE
i(1, 2)
#> [1] FALSE FALSE
```

Sometimes you want to add a non-trivial default value, which might take several lines of code to compute. Instead of inserting that code in the function definition, you could use `missing()` to conditionally compute it if needed. However, this makes it hard to know which arguments are required and which are optional without carefully reading the documentation. Instead, I usually set the default value to `NULL` and use `is.null()` to check if the argument was supplied.

Return values

The last expression evaluated in a function becomes the return value, the result of invoking the function.

```
f <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
f(5)
#> [1] 0
f(15)
#> [1] 10
```

Generally, I think it's good style to reserve the use of an explicit `return()` for when you are returning early, such as for an error, or a simple case of the function. This style of programming can also reduce the level of indentation, and generally make functions easier to understand because you can reason about them locally.

```
f <- function(x, y) {
  if (!x) return(y)

  # complicated processing here
}
```

Functions can return only a single object. But this is not a limitation because you can return a list containing any number of objects.

The functions that are the easiest to understand and reason about are pure functions: functions that always map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no **side effects**: they don't affect the state of the world in any way apart from the value they return.

R protects you from one type of side effect: most R objects have copy-on-modify semantics. So modifying a function argument does not change the original value:

```
f <- function(x) {
  x$a <- 2
  x
}
x <- list(a = 1)
f(x)
#> $a
#> [1] 2
```

```
x$a
#> [1] 1
```

(There are two important exceptions to the copy-on-modify rule: environments and reference classes. These can be modified in place, so extra care is needed when working with them.)

This is notably different to languages like Java where you can modify the inputs of a function. This copy-on-modify behaviour has important performance consequences which are discussed in depth in profiling. (Note that the performance consequences are a result of R's implementation of copy-on-modify semantics; they are not true in general. Clojure is a new language that makes extensive use of copy-on-modify semantics with limited performance consequences.)

Most base R functions are pure, with a few notable exceptions:

- `library()` which loads a package, and hence modifies the search path.
- `setwd()`, `Sys.setenv()`, `Sys.setlocale()` which change the working directory, environment variables, and the locale, respectively.
- `plot()` and friends which produce graphical output.
- `write()`, `write.csv()`, `saveRDS()`, etc. which save output to disk.
- `options()` and `par()` which modify global settings.
- S4 related functions which modify global tables of classes and methods.
- Random number generators which produce different numbers each time you run them.

It's generally a good idea to minimise the use of side effects, and where possible, to minimise the footprint of side effects by separating pure from impure functions. Pure functions are easier to test (because all you need to worry about are the input values and the output), and are less likely to work differently on different versions of R or on different platforms. For example, this is one of the motivating principles of ggplot2: most operations work on an object that represents a plot, and only the final `print` or`plot` call has the side effect of actually drawing the plot.

Functions can return `invisible` values, which are not printed out by default when you call the function.

```
f1 <- function() 1
f2 <- function() invisible(1)

f1()
#> [1] 1
f2()
f1() == 1
#> [1] TRUE
f2() == 1
#> [1] TRUE
```

You can force an invisible value to be displayed by wrapping it in parentheses:

```
(f2())
#> [1] 1
```

The most common function that returns invisibly is <-:

```
a <- 2
(a <- 2)
#> [1] 2
```

This is what makes it possible to assign one value to multiple variables:

```
a <- b <- c <- d <- 2
```

because this is parsed as:

```
(a <- (b <- (c <- (d <- 2))))
#> [1] 2
```