# 6_Functions

2022-10-03

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.8
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

# 6. Functions

## 6.1 Introduction

### Quiz

1. What are the three components of a function?

arguments, body, and environment

2. What does the following code return?

```
x <- 10
f1 <- function(x) {
  function() {
    x + 10
  }
}
f1(1)()
```

```
## [1] 11
```

3. How would you usually write this code?

```
`+`(1, `*`(2, 3))
```

```
## [1] 7
```

4. How could you make this call easier to read?

```
mean(, TRUE, x = c(1:10, NA))
```

```
## [1] 5.5
```

5. Does the following code throw an error when executed? Why or why not?

```
f2 <- function(a, b) {
  a * 10
}
f2(10, stop("This is an error!"))
```

```
## [1] 100
```

6. What is an infix function? How do you write it? What's a replacement function? How do you write it?
7. How do you ensure that cleanup action occurs regardless of how a function exits?

## 6.2 Function fundamentals

- Functions can be broken down into three components: arguments, body, and environment.
- Functions are objects, just as vectors are objects.

### 6.2.1 Function components

A function has three parts:

- The formals(), the list of arguments that control how you call the function.
- The body(), the code inside the function.
- The environment(), the data structure that determines how the function finds the values associated with the names.

```
f02 <- function(x, y) {
  # A comment
  x + y
}

formals(f02)
```

```
## $x
##
##
## $y
```

```
body(f02)
```

```
## {
##     x + y
## }
```

```
environment(f02)
```

```
## <environment: R_GlobalEnv>
```

```
attr(f02, "srcref")
```

```
## function(x, y) {
##   # A comment
##   x + y
## }
```

### 6.2.2 Primitive functions

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
`[`
```

```
## .Primitive("[")
```

```
typeof(sum)
```

```
## [1] "builtin"
```

```
typeof(`[`)
```

```
## [1] "special"
```

```
formals(sum)
```

```
## NULL
```

```
body(sum)
```

```
## NULL
```

```
environment(sum)
```

```
## NULL
```

### 6.2.3 First-class functions

```r
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```

```r
lapply(mtcars, function(x) length(unique(x)))
```

```
## $mpg
## [1] 25
##
## $cyl
## [1] 3
##
## $disp
## [1] 27
##
## $hp
## [1] 22
##
## $drat
## [1] 22
##
## $wt
## [1] 29
##
## $qsec
## [1] 30
##
## $vs
## [1] 2
##
## $am
## [1] 2
##
## $gear
## [1] 3
##
## $carb
## [1] 6
```

```r
Filter(function(x) !is.numeric(x), mtcars)
```

```
## data frame with 0 columns and 32 rows
```

```r
integrate(function(x) sin(x) ^ 2, 0, pi)
```

```
## 1.570796 with absolute error < 1.7e-14
```

```r
funs <- list(
  half = function(x) x / 2,
  double = function(x) x * 2
```

```
)

funs$double(10)
```

```
## [1] 20
```

### 6.2.4 Invoking a function

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

```
do.call(mean, args)
```

```
## [1] 5.5
```

### 6.2.5 Exercises

1. Given a name, like `"mean"`, `match.fun()` lets you find a function. Given a function, can you find its name? Why doesn't that make sense in R?

It's a lot easier to match a name then it is to match all the code in a function to a name

2. It's possible (although typically not useful) to call an anonymous function. Which of the two approaches below is correct? Why?

```
function(x) 3()
```

```
## function(x) 3()
```

```
(function(x) 3)()
```

```
## [1] 3
```

2nd?

3. A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use {}. Review your code. Where could you have used an anonymous function instead of a named function? Where should you have used a named function instead of an anonymous function?

Should have used an anonymous on smaller quick functions. Use named functions on larget multi line chunks that need comments

4. What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function?

```
is.primitive(sum)
```

```
## [1] TRUE
```

```
is.function(mean)
```

```
## [1] TRUE
```

5. This code makes a list of all functions in the base package.

```
objs <- mget(ls("package:base", all = TRUE), inherits = TRUE)
funs <- Filter(is.function, objs)
```

a. Which base function has the most arguments?

```
sort(unlist((lapply(
  lapply(funs, FUN = formals), length
))), decreasing = T) %>% head()
```

```
##             scan   format.default           source          formatC
##               22               16               16               15
##          library merge.data.frame
##               13               13
```

b. How many base functions have no arguments? What's special about those functions?

```
a <- unlist((lapply(
  lapply(funs, FUN = formals), length
)))
sum(a == 0)
```

```
## [1] 253
```

```
head(a[a == 0])
```

```
##   -   !  !=   $ $<-  %%
##   0   0   0   0   0   0
```

```
is.primitive(`-`)
```

```
## [1] TRUE
```

```
is.primitive("scan")
```

```
## [1] FALSE
```

```
# Are primitive functions
rm(a)
```

c. How could you adapt the code to find all primitive functions?

```
objs <- mget(ls("package:base", all = TRUE), inherits = TRUE)
funs <- Filter(is.primitive, objs)
```

6. What are the three important components of a function?

Formals, Body, Environment

7. When does printing a function not show the environment it was created in?

Primitives and functions created in the global environment

## 6.3 Function composition

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)
```

```
x <- runif(100)

sqrt(mean(square(deviation(x))))
```

```
## [1] 0.2937235
```

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
```

```
## [1] 0.2937235
```

```
library(magrittr)
```

```
##
## Attaching package: 'magrittr'

## The following object is masked from 'package:purrr':
##
##     set_names

## The following object is masked from 'package:tidyr':
##
##     extract
```

```
x %>%
  deviation() %>%
  square() %>%
  mean() %>%
  sqrt()
```

```
## [1] 0.2937235
```

## 6.4 Lexical scoping

```
x <- 10
g01 <- function() {
  x <- 20
  x
}

g01()
```

```
## [1] 20
```

R's lexical scoping follows four primary rules:

- Name masking
- Functions versus variables
- A fresh start
- Dynamic lookup

### 6.4.1 Name masking

```
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
```

```
## [1] 1 2
```

```
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
```

```
## [1] 2 1
```

```
y
```

```
## [1] 20
```

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

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

### 6.4.2 Functions versus variables

```
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
```

```
## [1] 110
```

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
```

```
## [1] 110
```

### 6.4.3 A fresh start

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
```

```
## [1] 1
```

```
g11()
```

```
## [1] 1
```

### 6.4.4 Dynamic lookup

```
g12 <- function() x + 1
x <- 15
g12()
```

```
## [1] 16
```

```
x <- 20
g12()
```

```
## [1] 21
```

```
codetools::findGlobals(g12)
```

```
## [1] "+" "x"
```

```
environment(g12) <- emptyenv()
g12()
```

```
## Error in x + 1: could not find function "+"
```

### 6.4.5 Exercises

1. What does the following code return? Why? Describe how each of the three c's is interpreted.

```
c <- 10
c(c = c)
```

```
##  c
## 10
```

Creates a vector with one named element "c" with a value of 10

2. What are the four principles that govern how R looks for values?

Name masking, Functions versus variables, A fresh start, Dynamic lookup

3. What does the following function return? Make a prediction before running the code yourself.

```
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x ^ 2
    }
    f() + 1
  }
  f(x) * 2
}
f(10)
```

```
## [1] 202
```

202

## 6.5 Lazy evaluation

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
```

```
## [1] 10
```

### 6.5.1 Promises

```
y <- 10
h02 <- function(x) {
  y <- 100
  x + 1
}

h02(y)
```

```
## [1] 11
```

```
h02(y <- 1000)
```

```
## [1] 1001
```

```
y
```

```
## [1] 1000
```

```
double <- function(x) {
  message("Calculating...")
  x * 2
}

h03 <- function(x) {
  c(x, x)
}

h03(double(20))
```

```
## Calculating...

## [1] 40 40
```

### 6.5.2 Default arguments

```
h04 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}

h04()
```

```
## [1]   1   2 110
```

```
h05 <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside h05:
h05()
```

```
## [1] "a" "x"
```

```
# ls() evaluated in global environment:
h05(ls())
```

```
##  [1] "args"      "c"         "deviation" "double"    "f"         "f01"
##  [7] "f02"       "f1"        "f2"        "funs"      "g01"       "g02"
## [13] "g03"       "g04"       "g07"       "g08"       "g09"       "g10"
## [19] "g11"       "g12"       "h01"       "h02"       "h03"       "h04"
## [25] "h05"       "objs"      "out"       "square"    "x"         "y"
```

### 6.5.3 Missing arguments

```
h06 <- function(x = 10) {
  list(missing(x), x)
}
str(h06())
```

```
## List of 2
##  $ : logi TRUE
##  $ : num 10
```

```
str(h06(10))
```

```
## List of 2
##  $ : logi FALSE
##  $ : num 10
```

```
args(sample)
```

```
## function (x, size, replace = FALSE, prob = NULL)
## NULL
```

```
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  if (is.null(size)) {
    size <- length(x)
  }

  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

```
`%||%` <- function(lhs, rhs) {
  if (!is.null(lhs)) {
    lhs
  } else {
    rhs
  }
}

sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  size <- size %||% length(x)
  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

### 6.5.4 Exercises

1. What important property of && makes x_ok() work?

```
x_ok <- function(x) {
  !is.null(x) && length(x) == 1 && x > 0
}

x_ok(NULL)
```

```
## [1] FALSE
```

```
x_ok(1)
```

```
## [1] TRUE
```

```
x_ok(1:3)
```

```
## [1] FALSE
```

Processed from left to right. Prevents error later on by requiring the first tests to work.

```
x_ok <- function(x) {
  !is.null(x) & length(x) == 1 & x > 0
}

x_ok(NULL)
```

```
## logical(0)
```

```
x_ok(1)
```

```
## [1] TRUE
```

```
x_ok(1:3)
```

```
## [1] FALSE FALSE FALSE
```

Runs each logical test, no control flow. Provide values to tests that should not accept them

2. What does this function return? Why? Which principle does it illustrate?

```
f2 <- function(x = z) {
  z <- 100
  x
}
f2()
```

```
## [1] 100
```

100, lazy evaluation

3. What does this function return? Why? Which principle does it illustrate?

```
y <- 10
f1 <- function(x = {y <- 1; 2}, y = 0) {
  c(x, y)
}
f1()
```

```
## [1] 2 1
```

```
y
```

```
## [1] 10
```

(2, 1), 10. name masking

4. In hist(), the default value of xlim is range(breaks), the default value for breaks is “Sturges”, and

```
range("Sturges")
```

```
## [1] "Sturges" "Sturges"
```

Explain how hist() works to get a correct xlim value.

Sturges is a function called which calculates bins for you

5. Explain why this function works. Why is it confusing?

```
show_time <- function(x = stop("Error!")) {
  stop <- function(...) Sys.time()
  print(x)
}
show_time()
```

```
## [1] "2022-10-13 11:51:06 PDT"
```

The default value for x is run the function stop with the argument “Error!”. Which should stop and prevent an error message. The stop function is then redefined in the function to provide the system time instead. They evaluation is then applied to the value of x. Last x is print given its environmental value which is the system time. It’s confusing because we are masking functions and lazy evaluating calls.

6. How many arguments are required when calling library()?

None

## 6.6 Dot-Dot-Dot

```
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
```

```
## List of 2
##  $ y: num 2
##  $ z: num 3
```

```
i03 <- function(...) {
  list(first = ..1, third = ..3)
}
str(i03(1, 2, 3))
```

```
## List of 2
##  $ first: num 1
##  $ third: num 3
```

```
i04 <- function(...) {
  list(...)
}
str(i04(a = 1, b = 2))
```

```
## List of 2
##  $ a: num 1
##  $ b: num 2
```

```
x <- list(c(1, 3, NA), c(4, NA, 6))
str(lapply(x, mean, na.rm = TRUE))
```

```
## List of 2
##  $ : num 2
##  $ : num 5
```

```
print(factor(letters), max.levels = 4)
```

```
##  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
## 26 Levels: a b c ... z
```

```
print(y ~ x, showEnv = TRUE)
```

```
## y ~ x
## <environment: R_GlobalEnv>
```

```
sum(1, 2, NA, na_rm = TRUE)
```

```
## [1] NA
```

### 6.6.1 Exercises

1. Explain the following results:

```
sum(1, 2, 3)
```

```
## [1] 6
```

```r
mean(1, 2, 3)
```

```
## [1] 1
```

```r
sum(1, 2, 3, na.omit = TRUE)
```

```
## [1] 7
```

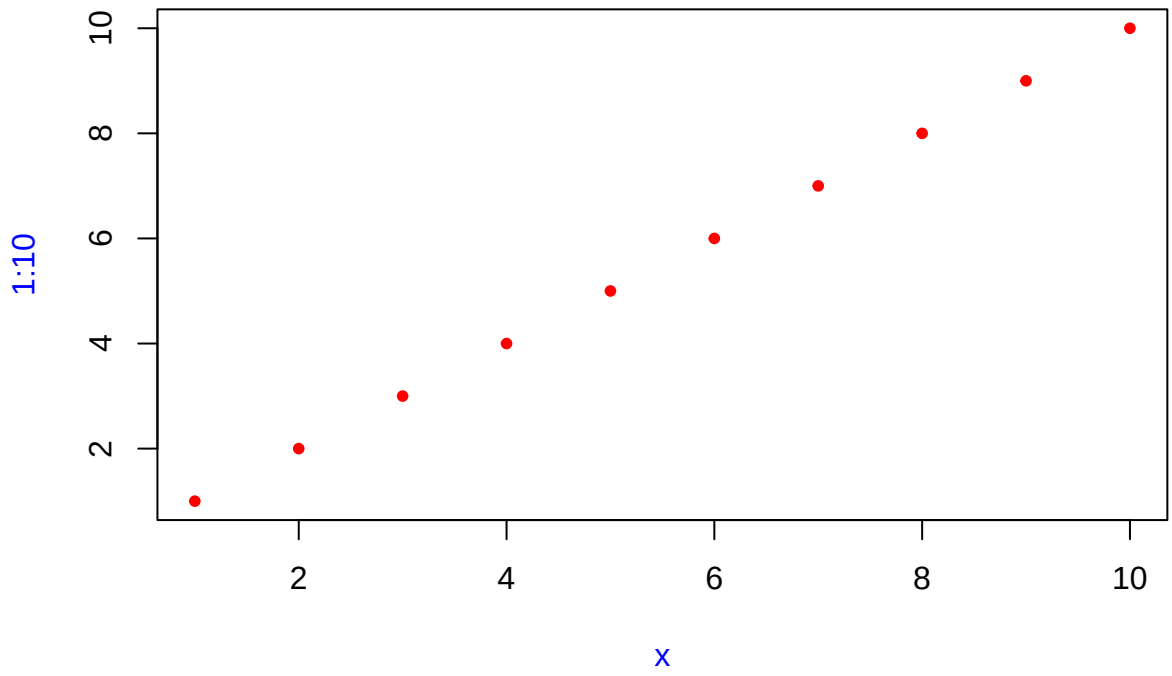
```r
mean(1, 2, 3, na.omit = TRUE)
```

```
## [1] 1
```

As expected. The first argument of mean is the vector to calculate over which in this case is the single element vector of 1, the 2 and 3 are additional arguments. The na.omit = TRUE is sum is equal to 1 and a typo of the argument na.rm so it get added. In the last it's not included because it's an argument with a typo that gets passed on.

2. Explain how to find the documentation for the named arguments in the following function call:

```r
plot(1:10, col = "red", pch = 20, xlab = "x", col.lab = "blue")
```




Follow the help documentation

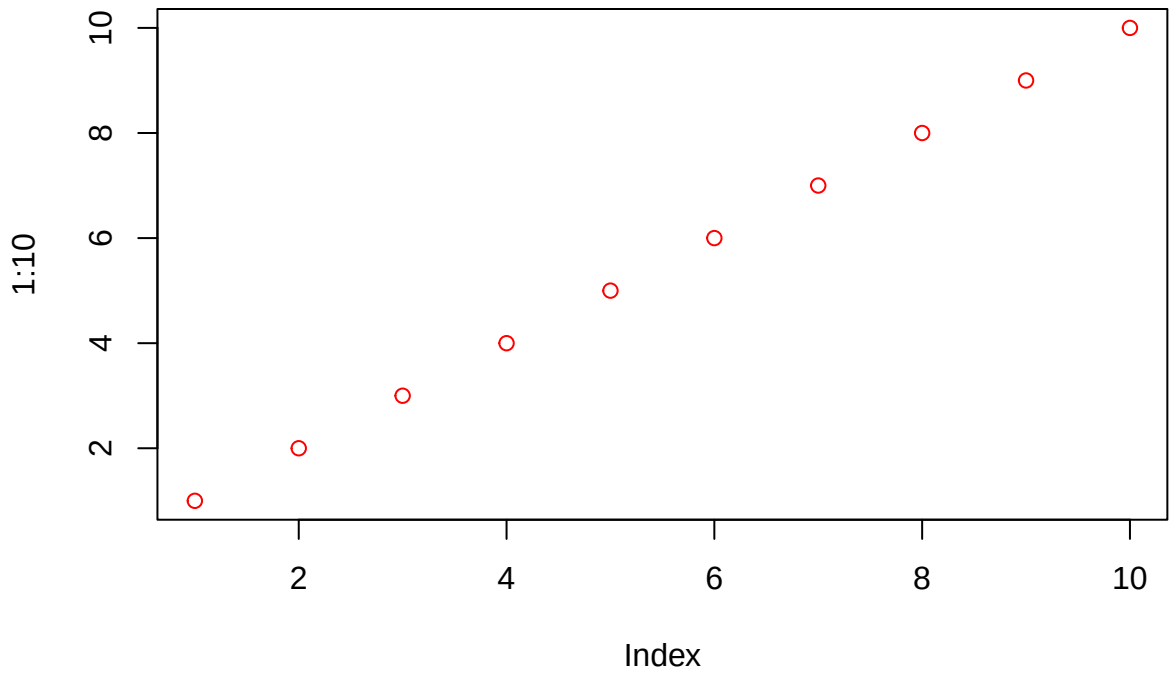
```
str(plot)
```

```
## function (x, y, ...)
```

```
#?plot
```

3. Why does plot(1:10, col = “red”) only colour the points, not the axes or labels? Read the source code of plot.default() to find out.

```
plot(1:10, col = "red")
```




```
plot.default
```

```
## function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
##     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
##     ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
##     panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,
##     ...)
## {
##     localAxis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)
##     localBox <- function(..., col, bg, pch, cex, lty, lwd) box(...)
##     localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)
##     localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)
```

```
##     xlabel <- if (!missing(x))
##         deparse1(substitute(x))
##     ylabel <- if (!missing(y))
##         deparse1(substitute(y))
##     xy <- xy.coords(x, y, xlabel, ylabel, log)
##     xlab <- if (is.null(xlab))
##         xy$xlab
##     else xlab
##     ylab <- if (is.null(ylab))
##         xy$ylab
##     else ylab
##     xlim <- if (is.null(xlim))
##         range(xy$x[is.finite(xy$x)])
##     else xlim
##     ylim <- if (is.null(ylim))
##         range(xy$y[is.finite(xy$y)])
##     else ylim
##     dev.hold()
##     on.exit(dev.flush())
##     plot.new()
##     localWindow(xlim, ylim, log, asp, ...)
##     panel.first
##     plot.xy(xy, type, ...)
##     panel.last
##     if (axes) {
##         localAxis(if (is.null(y))
##             xy$x
##         else x, side = 1, gap.axis = xgap.axis, ...)
##         localAxis(if (is.null(y))
##             x
##         else y, side = 2, gap.axis = ygap.axis, ...)
##     }
##     if (frame.plot)
##         localBox(...)
##     if (ann)
##         localTitle(main = main, sub = sub, xlab = xlab, ylab = ylab,
##             ...)
##     invisible()
## }
## <bytecode: 0x0000000024993cb0>
## <environment: namespace:graphics>
```

It is also expected. We set the color for the points, but did not for the labels.

## 6.7 Exiting a function

### 6.7.1 Implicit versus explicit returns

```
# Implicitly, where the last evaluated expression is the return value:
j01 <- function(x) {
  if (x < 10) {
    0
```

```
  } else {
    10
  }
}
j01(5)
```

```
## [1] 0
```

```
j01(15)
```

```
## [1] 10
```

```
# Explicitly, by calling return():
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
j02(5)
```

```
## [1] 0
```

```
j02(15)
```

```
## [1] 10
```

### 6.7.2 Invisible values

```
j03 <- function() 1
j03()
```

```
## [1] 1
```

```
j04 <- function() invisible(1)
j04()
```

```
print(j04())
```

```
## [1] 1
```

```
(j04())
```

```
## [1] 1
```

```
str(withVisible(j04()))
```

```
## List of 2
##  $ value  : num 1
##  $ visible: logi FALSE
```

```
a <- 2
(a <- 2)
```

```
## [1] 2
```

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

### 6.7.3 Errors

```
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
```

```
## Error in j05(): I'm an error
```

### 6.7.4 Exit handlers

```
j06 <- function(x) {
  cat("Hello\n")
  on.exit(cat("Goodbye!\n"), add = TRUE)

  if (x) {
    return(10)
  } else {
    stop("Error")
  }
}

j06(TRUE)
```

```
## Hello
## Goodbye!
```

```
## [1] 10
```

```
j06(FALSE)
```

```
## Hello
```

```
## Error in j06(FALSE): Error

## Goodbye!
```

```
cleanup <- function(dir, code) {
  old_dir <- setwd(dir)
  on.exit(setwd(old_dir), add = TRUE)

  old_opt <- options(stringsAsFactors = FALSE)
  on.exit(options(old_opt), add = TRUE)
}
```

```
with_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  force(code)
}

getwd()
```

```
## [1] "D:/Advanced_R_Hadley_2nd_Ed"
```

```
with_dir("~", getwd())
```

```
## [1] "C:/Users/johnt/OneDrive/Documents"
```

```
getwd()
```

```
## [1] "D:/Advanced_R_Hadley_2nd_Ed"
```

```
j08 <- function() {
  on.exit(message("a"), add = TRUE)
  on.exit(message("b"), add = TRUE)
}
j08()
```

```
## a

## b
```

```
j09 <- function() {
  on.exit(message("a"), add = TRUE, after = FALSE)
  on.exit(message("b"), add = TRUE, after = FALSE)
}
j09()
```

```
## b

## a
```

### 6.7.5 Exercises

1. What does load() return? Why don't you normally see these values?

```
load
```

```
## function (file, envir = parent.frame(), verbose = FALSE)
## {
##     if (is.character(file)) {
##         con <- gzfile(file)
##         on.exit(close(con))
##         magic <- readChar(con, 5L, useBytes = TRUE)
##         if (!length(magic))
##             stop("empty (zero-byte) input file")
##         if (!grepl("RD[ABX][2-9]\n", magic)) {
##             if (grepl("RD[ABX][2-9]\r", magic))
##                 stop("input has been corrupted, with LF replaced by CR")
##             warning(sprintf("file %s has magic number '%s'\n",
##                 sQuote(basename(file)), gsub("[\n\r]*", "", magic)),
##                 "  ", "Use of save versions prior to 2 is deprecated",
##                 domain = NA, call. = FALSE)
##             return(.Internal(load(file, envir)))
##         }
##     }
##     else if (inherits(file, "connection")) {
##         con <- if (inherits(file, "gzfile") || inherits(file,
##             "gzcon"))
##             file
##         else gzcon(file)
##     }
##     else stop("bad 'file' argument")
##     if (verbose)
##         cat("Loading objects:\n")
##     .Internal(loadFromConn2(con, envir, verbose))
## }
## <bytecode: 0x0000000015d7b4e0>
## <environment: namespace:base>
```

Load will load the objects from a file and add them to the environment. Nothing is shown since verbose is set to false.

2. What does write.table() return? What would be more useful?

```
write.table
```

```
## function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
##     eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
##     qmethod = c("escape", "double"), fileEncoding = "")
## {
##     qmethod <- match.arg(qmethod)
##     if (is.logical(quote) && (length(quote) != 1L || is.na(quote)))
##         stop("'quote' must be 'TRUE', 'FALSE' or numeric")
```

```
##     quoteC <- if (is.logical(quote))
##         quote
##     else TRUE
##     qset <- is.logical(quote) && quote
##     if (!is.data.frame(x) && !is.matrix(x))
##         x <- data.frame(x)
##     makeRownames <- isTRUE(row.names)
##     makeColnames <- is.logical(col.names) && !identical(FALSE,
##         col.names)
##     if (is.matrix(x)) {
##         p <- ncol(x)
##         d <- dimnames(x)
##         if (is.null(d))
##             d <- list(NULL, NULL)
##         if (is.null(d[[1L]]) && makeRownames)
##             d[[1L]] <- seq_len(nrow(x))
##         if (is.null(d[[2L]]) && makeColnames && p > 0L)
##             d[[2L]] <- paste0("V", 1L:p)
##         if (qset)
##             quote <- if (is.character(x))
##                 seq_len(p)
##             else numeric()
##     }
##     else {
##         if (qset)
##             quote <- if (length(x))
##                 which(unlist(lapply(x, function(x) is.character(x) ||
##                   is.factor(x))))
##             else numeric()
##         if (any(vapply(x, function(z) length(dim(z)) == 2 &&
##             dim(z)[2L] > 1, NA))) {
##             c1 <- names(x)
##             x <- as.matrix(x, rownames.force = makeRownames)
##             d <- dimnames(x)
##             if (qset) {
##                 ord <- match(c1, d[[2L]], 0L)
##                 quote <- ord[quote]
##                 quote <- quote[quote > 0L]
##             }
##         }
##         else d <- list(if (makeRownames) row.names(x), if (makeColnames) names(x))
##         p <- ncol(x)
##     }
##     nocols <- p == 0L
##     if (is.logical(quote))
##         quote <- NULL
##     else if (is.numeric(quote)) {
##         if (any(quote < 1L | quote > p))
##             stop("invalid numbers in 'quote'")
##     }
##     else stop("invalid 'quote' specification")
##     rn <- FALSE
##     rnames <- NULL
##     if (is.logical(row.names)) {
```

```
##         if (row.names) {
##             rnames <- as.character(d[[1L]])
##             rn <- TRUE
##         }
##     }
##     else {
##         rnames <- as.character(row.names)
##         rn <- TRUE
##         if (length(rnames) != nrow(x))
##             stop("invalid 'row.names' specification")
##     }
##     if (!is.null(quote) && rn)
##         quote <- c(0, quote)
##     if (is.logical(col.names)) {
##         if (!rn && is.na(col.names))
##             stop("'col.names = NA' makes no sense when 'row.names = FALSE'")
##         col.names <- if (is.na(col.names) && rn)
##             c("", d[[2L]])
##         else if (col.names)
##             d[[2L]]
##         else NULL
##     }
##     else {
##         col.names <- as.character(col.names)
##         if (length(col.names) != p)
##             stop("invalid 'col.names' specification")
##     }
##     if (file == "")
##         file <- stdout()
##     else if (is.character(file)) {
##         file <- if (nzchar(fileEncoding))
##             file(file, ifelse(append, "a", "w"), encoding = fileEncoding)
##         else file(file, ifelse(append, "a", "w"))
##         on.exit(close(file))
##     }
##     else if (!isOpen(file, "w")) {
##         open(file, "w")
##         on.exit(close(file))
##     }
##     if (!inherits(file, "connection"))
##         stop("'file' must be a character string or connection")
##     qstring <- switch(qmethod, escape = "\\\"", double = "\"\"")
##     if (!is.null(col.names)) {
##         if (append)
##             warning("appending column names to file")
##         if (quoteC)
##             col.names <- paste0("\"", gsub("\"", qstring, col.names,
##                 fixed = TRUE), "\"")
##         writeLines(paste(col.names, collapse = sep), file, sep = eol)
##     }
##     if (nrow(x) == 0L)
##         return(invisible())
##     if (nocols && !rn)
##         return(cat(rep.int(eol, NROW(x)), file = file, sep = ""))
```

```
##     if (is.matrix(x) && !is.atomic(x))
##         mode(x) <- "character"
##     if (is.data.frame(x)) {
##         x[] <- lapply(x, function(z) {
##             if (is.object(z) && !is.factor(z))
##                 as.character(z)
##             else z
##         })
##     }
##     invisible(.External2(C_writetable, x, file, nrow(x), p, rnames,
##         sep, eol, na, dec, as.integer(quote), qmethod != "double"))
## }
## <bytecode: 0x00000000247aa408>
## <environment: namespace:utils>
```

```
# write.table(a, "test.txt")
```

nothing is printed to the screen. Would be useful if it told you the table was correctly saved.

3. How does the chdir parameter of source() compare to with_dir()? Why might you prefer one to the other?

```
?source
```

```
## starting httpd help server ... done
```

```
?with_dir
```

source lets you give the file path and the chdir parameter let's you use the files directory as the working directory. May have to change directory back. with_dir lets you set a new working directory for the code you're going to execute and then returns back to the original working directory. Both depend on what other files you want to use for your execution

4. Write a function that opens a graphics device, runs the supplied code, and closes the graphics device (always, regardless of whether or not the plotting code works).

```
trash <- function(){
  on.exit(dev.off(), add = TRUE)
  png("test.png")
  plot(1:10)
}
trash()
```

5. We can use on.exit() to implement a simple version of capture.output().

```
capture.output2 <- function(code) {
  temp <- tempfile()
  on.exit(file.remove(temp), add = TRUE, after = TRUE)

  sink(temp)
  on.exit(sink(), add = TRUE, after = TRUE)
```

```
  force(code)
  readLines(temp)
}
capture.output(cat("a", "b", "c", sep = "\n"))
```

```
## [1] "a" "b" "c"
```

```
capture.output2(cat("a", "b", "c", sep = "\n"))
```

```
## Warning in file.remove(temp): cannot remove file 'C:
## \Users\johnt\AppData\Local\Temp\RtmpGiocJP\file310844c01a35', reason 'Permission
## denied'
```

```
## [1] "a" "b" "c"
```

Huh

## 6.8 Function forms

- prefix: the function name comes before its arguments, like foofy(a, b, c). These constitute of the majority of function calls in R.
- infix: the function name comes in between its arguments, like x + y. Infix forms are used for many mathematical operators, and for user-defined functions that begin and end with %.
- replacement: functions that replace values by assignment, like names(df) <- c("a", "b", "c"). They actually look like prefix functions.
- special: functions like [[, if, and for. While they don't have a consistent structure, they play important roles in R's syntax.

### 6.8.1 Rewriting to prefix form

```
x + y
```

```
## Error in x + y: non-numeric argument to binary operator
```

```
`+`(x, y)
```

```
## Error in x + y: non-numeric argument to binary operator
```

```
names(df) <- c("x", "y", "z")
```

```
## Error in names(df) <- c("x", "y", "z"): names() applied to a non-vector
```

```
`names<-`(df, c("x", "y", "z"))
```

```
## Error in eval(expr, envir, enclos): names() applied to a non-vector
```

```
for(i in 1:10) print(i)
```

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

```
`for`(i, 1:10, print(i))
```

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

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
```

```
##  [1] 3 3 4 3 3 3 3 4 3 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3
## [39] 3 4 3 3 3 3 3 3 3 4 3 3
```

```
rm("(")
```

```
add <- function(x, y) x + y
lapply(list(1:3, 4:5), add, 3)
```

```
## [[1]]
## [1] 4 5 6
##
## [[2]]
## [1] 7 8
```

```
lapply(list(1:3, 4:5), `+`, 3)
```

```
## [[1]]
## [1] 4 5 6
##
## [[2]]
## [1] 7 8
```

### 6.8.2 Prefix form

```
k01 <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(k01(1, 2, 3))
```

```
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
```

```
str(k01(2, 3, abcdef = 1))
```

```
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
```

```
# Can abbreviate long argument names:
str(k01(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(k01(1, 3, b = 1))
```

```
## Error in k01(1, 3, b = 1): argument 3 matches multiple formal arguments
```

```
options(warnPartialMatchArgs = TRUE)
x <- k01(a = 1, 2, 3)
```

```
## Warning in k01(a = 1, 2, 3): partial argument match of 'a' to 'abcdef'
```

### 6.8.3 Infix functions

```
`%+%` <- function(a, b) paste0(a, b)
"new " %+% "string"
```

```
## [1] "new string"
```

```
`% %` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)

"a" % % "b"
```

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

```
"a" %/\% "b"
```

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

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
```

```
## [1] "((a %-% b) %-% c)"
```

```
-1
```

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

```
+10
```

```
## [1] 10
```

### 6.8.4 Replacement functions

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
```

```
x <- 1:10
second(x) <- 5L
x
```

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

```
x <- 1:10
tracemem(x)
```

```
## [1] "<0000000023FA77F8>"
```

```
second(x) <- 6L
```

```
## tracemem[0x0000000023fa77f8 -> 0x0000000026420048]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x0000000026420048 -> 0x00000000263d3cc8]: second<- eval eval eval_with_user_handlers withV
```

```
untracemem(x)
```

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
```

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

```
x <- `modify<-`(x, 1, 10)
```

```
x <- c(a = 1, b = 2, c = 3)
names(x)
```

```
## [1] "a" "b" "c"
```

```
names(x)[2] <- "two"
names(x)
```

```
## [1] "a"   "two" "c"
```

```
`*tmp*` <- x
x <- `names<-`(`*tmp*`, `[<-`(names(`*tmp*`), 2, "two"))
rm(`*tmp*`)
```

### 6.8.5 Special forms

```
`for`
```

```
## .Primitive("for")
```

### 6.8.6 Exercises

1. Rewrite the following code snippets into prefix form:

```
1 + 2 + 3
```

```
## [1] 6
```

```
`+`(1, `+`(2,3))
```

```
## [1] 6
```

```
1 + (2 + 3)
```

```
## [1] 6
```

```
`+`(1, `(`(`+`(2,3)))
```

```
## [1] 6
```

```
x <- 1:10
n <- 3

if (length(x) <= 5) x[[5]] else x[[n]]
```

```
## [1] 3
```

```
`if`(length(x) <= 5, `[[`(x,5), `[[`(x,n))
```

```
## [1] 3
```

2. Clarify the following list of odd function calls:

```
set.seed(1)
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
x
```

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

```
set.seed(1)
x <- sample(x = c(1:10, NA), size = 20, replace = TRUE)
x
```

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

```
set.seed(1)
y <- runif(min = 0, max = 1, 20)
y
```

```
##  [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968
##  [7] 0.94467527 0.66079779 0.62911404 0.06178627 0.20597457 0.17655675
## [13] 0.68702285 0.38410372 0.76984142 0.49769924 0.71761851 0.99190609
## [19] 0.38003518 0.77744522
```

```r
set.seed(1)
y <- runif(n = 20, min = 0, max = 1)
y
```

```
##  [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968
##  [7] 0.94467527 0.66079779 0.62911404 0.06178627 0.20597457 0.17655675
## [13] 0.68702285 0.38410372 0.76984142 0.49769924 0.71761851 0.99190609
## [19] 0.38003518 0.77744522
```

```r
cor(m = "k", y = y, u = "p", x = x)
```

```
## Warning in cor(m = "k", y = y, u = "p", x = x): partial argument match of 'u' to
## 'use'
```

```
## Warning in cor(m = "k", y = y, u = "p", x = x): partial argument match of 'm' to
## 'method'
```

```
## [1] 0.1776493
```

```r
cor(x = x, y = y, use = "pairwise.complete.obs", method = "kendall")
```

```
## [1] 0.1776493
```

3. Explain why the following code fails:

```r
modify <- function(x, position, value) {
  x[position] <- value
  x
}
x
```

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

```r
#modify(get("x"), 1) <- 10
modify(get("x"), 1, 10)
```

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

Modify in our case takes 3 arguments. Code shows 2

4. Create a replacement function that modifies a random location in a vector.

```r
`whoops<-` <- function(x, value){
  changed <- sample(1:length(x), size = 1)
  x[changed] <- value
  x
}

x
```

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

```
whoops(x) <- 42069
x
```

```
##  [1]     9     4     7     1     2     7    NA     2 42069     3     1     5
## [13]     5    10     6    10     7     9     5     5
```

5. Write your own version of + that pastes its inputs together if they are character vectors but behaves as usual otherwise. In other words, make this code work:

```
`%crapadd%` <- function(a, b){
  if(is.numeric(a) & is.numeric(b)) a + b else paste0(a,b)
}

1 %crapadd% 2
```

```
## [1] 3
```

```
"a" %crapadd% "b"
```

```
## [1] "ab"
```

```
1 %crapadd% "b"
```

```
## [1] "1b"
```

6. Create a list of all the replacement functions found in the base package. Which ones are primitive functions? (Hint: use apropos().)

```
(replacers <- apropos("<-$", where = T, mode = "function"))
```

```
##                  19                  19                  19                  19
##               "$<-"     ".rowNamesDF<-"              "@<-"             "[[<-"
##                  19                  19                  19                  17
##              "[<-"                "<-"             "<<-"             "as<-"
##                  19                  19                  17                  19
##           "attr<-"     "attributes<-"           "body<-"           "body<-"
##                  19                  17                  19                  19
##          "class<-"         "coerce<-"       "colnames<-"        "comment<-"
##                  12                  19                  19                  19
##      "contrasts<-"           "diag<-"            "dim<-"       "dimnames<-"
##                  17                  17                  19                  19
##             "el<-"        "elNamed<-"       "Encoding<-"    "environment<-"
##                  19                  17                  19                  17
##        "formals<-"   "functionBody<-"          "is.na<-"     "languageEl<-"
##                  19                  19                  19                   1
##         "length<-"         "levels<-"           "mode<-"         "modify<-"
##                  19                  19                  19                  17
## "mostattributes<-"          "names<-"       "oldClass<-"    "packageSlot<-"
##                  19                   6                  19                  19
```

```
##     "parent.env<-"          "pluck<-"     "regmatches<-"      "row.names<-"
##                19                 17                 17                 1
##       "rownames<-"        "S3Class<-"         "S3Part<-"         "second<-"
##                17                 19                 19                 4
##           "slot<-"          "split<-"   "storage.mode<-"        "str_sub<-"
##                19                 19                 12                19
##         "substr<-"      "substring<-"            "tsp<-"          "units<-"
##                 1                 12
##         "whoops<-"         "window<-"
```

```
# search() A character vector, starting with ".GlobalEnv", and ending with "package:base" which is R's
(base_replacers <- replacers[names(replacers) == length(search())])
```

```
##                19                 19                 19                 19
##              "$<-"    ".rowNamesDF<-"              "@<-"              "[[<-"
##                19                 19                 19                 19
##              "[<-"              "<-"             "<<-"           "attr<-"
##                19                 19                 19                 19
##     "attributes<-"          "body<-"         "class<-"       "colnames<-"
##                19                 19                 19                 19
##        "comment<-"          "diag<-"           "dim<-"       "dimnames<-"
##                19                 19                 19                 19
##       "Encoding<-"    "environment<-"        "formals<-"          "is.na<-"
##                19                 19                 19                 19
##         "length<-"         "levels<-"          "mode<-" "mostattributes<-"
##                19                 19                 19                 19
##          "names<-"       "oldClass<-"     "parent.env<-"     "regmatches<-"
##                19                 19                 19                 19
##      "row.names<-"       "rownames<-"          "split<-"   "storage.mode<-"
##                19                 19                 19
##         "substr<-"      "substring<-"          "units<-"
```

```
mget(base_replacers, envir = baseenv()) %>%
  Filter(is.primitive, .) %>%
  names()
```

```
##  [1] "$<-"            "@<-"            "[[<-"           "[<-"
##  [5] "<-"             "<<-"            "attr<-"         "attributes<-"
##  [9] "class<-"        "dim<-"          "dimnames<-"     "environment<-"
## [13] "length<-"       "levels<-"       "names<-"        "oldClass<-"
## [17] "storage.mode<-"
```

17 are primitive functions

7. What are valid names for user-created infix functions?

Anything except "%" as long as it's properly escaped

8. Create an infix xor() operator.

```
xor
```

```
## function (x, y)
## {
##     (x | y) & !(x & y)
## }
## <bytecode: 0x000000002231f940>
## <environment: namespace:base>
```

```
x <- rep(c(T,F), 5)
y <- rep(c(T,F), each = 5)

`%xor%` <- function(a, b){
  (a | b) & !(a & b)
}

xor(x,y)
```

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

```
x %xor% y
```

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

9. Create infix versions of the set functions intersect(), union(), and setdiff(). You might call them %n%, %u%, and %/% to match conventions from mathematics.

```
x <- 1:10
y <- 5:14
`%n%` <- function(a,b) intersect(a,b)
`%u%` <- function(a,b) union(a,b)
`%/%` <- function(a,b) setdiff(a,b)

intersect(x,y)
```

```
## [1]  5  6  7  8  9 10
```

```
x %n% y
```

```
## [1]  5  6  7  8  9 10
```

```
union(x,y)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

```
x %u% y
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

```
setdiff(x,y)
```

```
## [1] 1 2 3 4
```

```
x %/% y
```

```
## [1] 1 2 3 4
```