Chapter 6 Exercises

Derek Chiu

August 11, 2016

6.1.2 Exercises

1. 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.function()` tells you if an object is a function. `is.primitive()` tells you if a funciton is a primitive function.

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

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

    Use it to answer the following questions:

    a. Which base function has the most arguments?
    b. How many base functions have no arguments? What's special about those functions?
    c. How could you adapt the code to find all primitive functions?

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

# a.
funs.nargs <- sapply(funs, function(x) length(formals(x)))
which.max(funs.nargs)
```

```
## scan
##  938
```

```
# b.
length(which(funs.nargs == 0))
```

```
## [1] 225
```

```
# Most of these functions are primitive functions.

# c.
prims <- Filter(is.primitive, objs)
```

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

    > Body, arguments, and environment.

4. When does printing a function not show what environment it was created in?

    > The function is a primitive function.

6.2.5 Exercises

1. What does the following code return? Why? What does each of the three c's mean?

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

    The code returns a vector of length one named "c" with the value 10. The first c is the concatenation function c() that outputs the value, the second c is naming the value given to c(), the third c is the variable 10.

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

    Name masking, functions vs. 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) {
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

The function should return 202. It starts from the inner-most function, trying to find the value of x in its own environment, and then searches outward, while evaluating. The output is obtained by (((10 ˆ 2) + 1)) * 2) = 202.

6.4.6 Exercises

1. Clarify the following list of odd function calls:

```
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

The function call to x takes the vector `c(1:10, NA)`, and randomly samples 20 values from it with replacement. The function call to y randomly samples 20 values from the standard uniform distribution. The function call of `cor` takes the vectors `x` and `y`, and computes the correlation using `method = "kendall"` and `use = "pairwise.complete.obs"`.

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

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

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

The function returns 3 because the the arguments are missing, which means the default value for `y` is taken from the default value for `x`.

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

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

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

The function returns 100 because of lazy evaluation. The default value is only evaluated if used. Since f2() did not take in any arguments, the function returns the value that the argument depends on by default.

6.5.3 Exercises

1. Create a list of all the replacement functions found in the base package. Which ones are primitive functions?

```
objs <- mget(ls("package:base"), inherits = TRUE)
rplm <- objs[grep("<-", names(objs))]
rplm.prim <- Filter(is.primitive, rplm)
names(rplm.prim)
```

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

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

   Names must start and end with %, with anything in between, except % itself.

3. Create an infix `xor()` operator.

```
`%xor%` <- function(x, y) {
  (x | y) & !(x & y)
}
x <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
y <- c(FALSE, TRUE, FALSE, TRUE, TRUE)
xor(x, y)
```

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

```
x %xor% y
```

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

4. Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`.

```
`%intersect%` <- function(x, y) {
  x & y
}
`%union%` <- function(x, y) {
  x | y
}
`%setdiff%` <- function(x, y) {
  x & !y
}
x %intersect% y
```

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

```
x %union% y
```

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

```
x %setdiff% y
```

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

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

```
`replace_random<-` <- function(x, value) {
  pos <- sample(1:length(x), 1)
  x[pos] <- value
  x
}
x <- 1:10
set.seed(2)
replace_random(x) <- 12
x
```

```
##  [1]  1 12  3  4  5  6  7  8  9 10
```

```
replace_random(x) <- 11
x
```

```
##  [1]  1 12  3  4  5  6  7 11  9 10
```

6.6.2 Exercises

1. How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?

    > The `chdir` parameter allows the working directory to be temporarily changed. I might prefer `chdir` if I don't know the name of the working directory as it takes in only a logical, whereas `in_dir()` the directory needs to be specified.

2. What function undoes the action of `library()`? How do you save and restore the values of `options()` and `par()`?

    > The function `detach(unload = TRUE)` undoes the action of `library()`. Store `options()` and/or `par()` to a variable, and run the variable at `on.exit()`.

3. 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 worked).

```
plot_pdf <- function(x) {
  pdf(tempfile())
  plot(x)
  on.exit(dev.off())
}
plot_pdf(1:100)
```

4. 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)

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

  force(code)
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
```

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

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

Compare `capture.output()` to `capture.output2()`. How do the functions differ? What features have I removed to make the key ideas easier to see? How have I rewritten the key ideas to be easier to understand?

> Features removed include file and argument parsing, key ideas rewritten include `sink()` and `readLines()`.