Exercises - Class 4: Functions

Sander Bossier

2023-10-22

1.

Harmonic numbers are sums of the reciprocals of the positive integers. More specifically, the n'th harmonic number is:

$$\sum_{i=1}^{n} \frac{1}{i}$$

Create a function `Harmonic(n)` which computes the nth harmonic number. The body of this function can consist of a single line. An example is:

```
Harmonic <- function(n) sum(1 / (1:n))

Harmonic(5)
```

```
## [1] 2.283333
```

2.

Harmonic numbers are recursive:

$$H_n = H_{n-1} + \frac{1}{n}$$

Write a recursive function rHarmonic(n) which computes the nth harmonic number recursively (in other words, the result should be the same as in the previous exercise).

```
rHarmonic <- function(n) {
  if (n == 1) {
    1
  }
  else{
    rHarmonic(n - 1) + (1 / n)
  }
}
rHarmonic(5)
```

```
## [1] 2.283333
```

3.

Compare the computation time of `Harmonic(n)` and `rHarmonic(n)` for the 1000th harmonic number (using R's core timing function).

```
system.time(
  Harmonic(1000)
)
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(
  rHarmonic(1000)
)
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(
  Harmonic(1e6)
)
```

```
##    user  system elapsed
##    0.03    0.00    0.00
```

4.

Create a binary infix operator **%fill%** which replaces all **NA** values in the vector on the left-hand side by the value specified on the right-hand side.

```
`%fill%` <- function(vec, value) {
  vec[is.na(vec)] <- value # or
  #replace(vec, is.na(vec1), value)
  vec
}

vec1 <- c(5, 55, NA, -9, -99, NA, NA, -5, -9, 999)
vec2 <- vec1 %fill% 0
vec2
```

```
##  [1]   5  55   0  -9 -99   0   0  -5  -9 999
```

5.

Create a replacement function fill() which replaces all NA values in a vector by the value specified on the right-hand side.

```
`fill<-` <- function(vec, value) {
  vec[is.na(vec)] <- value
  vec
}

vec1 <- c(5, 55, NA, -9, -99, NA, NA, -5, -9, 999)
fill(vec1) <- 0
vec1
```

```
##  [1]   5  55   0  -9 -99   0   0  -5  -9 999
```

6.

Compare the computation time of the %fill% operator and the fill() replacement function on the following large vector (you can copy-paste these two lines):

```
library(microbenchmark)
set.seed(123)
vec3 <- sample(c(5, -9, NA), size = 1e6, replace = TRUE)
```

Do this using both R's core function and the package microbenchmark (replace the NA's by 0).

```
system.time(
  tmp <- vec3 %fill% 0
)
```

```
##    user  system elapsed
##    0.00    0.00    0.02
```

```
system.time(
  fill(vec3) <- 0
)
```

```
##    user  system elapsed
##    0.01    0.00    0.03
```

```
microbenchmark(
  tmp <- vec3 %fill% 0,
  unit = "us"
)
```

```
## Unit: microseconds
##                  expr  min      lq     mean   median       uq     max neval
##  tmp <- vec3 %fill% 0 3543 3694.65 4880.555 4207.551 5128.851 11381.2   100
```

```
microbenchmark(
  fill(vec3) <- 0,
  unit = "us"
)
```

```
## Unit: microseconds
##             expr      min       lq     mean   median       uq     max neval
##  fill(vec3) <- 0 3542.601 3744.651 4958.441 4363.201 5046.901 12335.3   100
```

7.

Create two functions **`full1()`** and **`full2()`** which both extract the non-NA elements from a vector. One of both functions needs to make use of the functional `Filter()` (see last week). Compare their computation times on the `vec3` object from the previous exercises. You can use both R's core function and the **microbenchmark** package but for the latter you can specify a number of `times` <= 100.

Note: There is even a way to combine the `Filter()` function with the `Negate()` function. If you manage to find that one, then you can compare all three functions.

```
full1 <- function(vec) {
  vec[!is.na(vec)]
}

full2 <- function(vec) {
  Filter(function(x) !is.na(x), vec)
}

full3 <- function(vec) {
  Filter(Negate(function(x) is.na(x)), vec)
}
```

```
vec1 <- c(5, 55, NA, -9, -99, NA, NA, -5, -9, 999)
microbenchmark(
  full1(vec3),
```

```
  full2(vec3),
  full3(vec3),
  times = 100,
  unit = "ns"
)
```

```
## Unit: nanoseconds
##         expr       min         lq         mean     median         uq
##  full1(vec3)   9472500   14876601     23071048   17656051   24042951
##  full2(vec3) 553414001  913735801   1109730769 1024485602 1230508601
##  full3(vec3) 967064801 1307615051 221269689718 1423236452 1645245701
##           max neval
##  1.044511e+08   100
##  1.856980e+09   100
##  2.160360e+13   100
```

8.

One of the features of R's syntax is that a function name before the opening bracket `(` can actually be a variable. This allows uses such as the following:

```
f <- sum
f(vec2)
```

```
## [1] 937
```

```
f <- mean
f(vec2)
```

```
## [1] 93.7
```

Create a function `twofun()` which has two formal arguments:

- `x` which can be a numeric vector
- `fun` which can be any function computing some descriptive statistic

The function `twofun()` should apply the function specified as `fun` to every two consecutive elements in the vector specified as `x`. In other words, if `fun = sum` then `twofun()` will compute the sum of every two consecutive elements in `x` like we have seen last week (see the last exercise of the previous week). However, `twofun()` allows for more flexibility:

```
twofun <- function(vec, fun = sum) {
  vapply(
    seq_len(length(vec) - 1),
    FUN = function(i) fun(vec[i:(i + 1)]),
    numeric(1)
  )
}
```

```
twofun(vec1, fun = sum)
```

```
## [1]   60   NA   NA -108   NA   NA   NA  -14  990
```

```
twofun(vec1, fun = mean)
```

```
## [1]  30  NA  NA -54  NA  NA  NA  -7 495
```

```
twofun(vec1, fun = median)
```

```
## [1]  30  NA  NA -54  NA  NA  NA  -7 495
```

```
twofun(vec1, fun = var)
```

```
## [1]   1250     NA     NA   4050     NA     NA     NA      8 508032
```

```
twofun(vec1)
```

```
## [1]   60   NA   NA -108   NA   NA   NA  -14  990
```

9.

R's functions for descriptive statistics have an argument `na.rm` and `var()` even has an argument `use`. Rewrite the `twosum()` function so that it can handle all of these extra arguments without any conflict.

```
twofun <- function(vec, fun = sum, ...) {

  vapply(
    X = seq_len(length(vec) - 1),
    FUN = function(i) fun(vec[i:(i + 1)], ...),
    FUN.VALUE = numeric(1)
  )
}
```

10.

```
twofunfun <- function(fun = sum) {
  out <- function(vec, ...) {
    vapply(
      X = seq_len(length(vec) - 1),
      FUN = function(i) fun(vec[i:(i + 1)], ...),
      FUN.VALUE = numeric(1)
    )
  }
  out
}
```

```
twomean <- twofunfun(fun = mean)
twomean(vec1)
```

```
## [1]  30  NA  NA -54  NA  NA  NA  -7 495
```

```
twovar <- twofunfun(fun = var)
twovar(vec1)
```

```
## [1]   1250     NA     NA   4050     NA     NA     NA      8 508032
```

```
environment(twomean)
```

```
## <environment: 0x00000258c5b78538>
```

```
environment(twovar)
```

```
## <environment: 0x00000258c59f6e28>
```

```
exists("fun", environment(twomean))
```

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

```
get("fun", environment(twomean))
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x00000258c48b1598>
## <environment: namespace:base>
```

```
exists("fun", environment(twovar))
```

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

```
get("fun", environment(twovar))
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x00000258c260f6b8>
## <environment: namespace:stats>
```