Practising R Programming

Team B

1/28/2022

Contents

1 Days of the week in Shire calendar

a) Write a function shire_day_from_gregorian() that converts a Gregorian day of the week into the name of the corresponding Shire day

```
shire_day_from_gregorian_if_else <- function(day){
  dplyr::if_else(day == "Monday",
          "Sterday",
          if_else(day =="Tuesday",
                  "Sunday",
                  if_else(day == "Wednesday",
                          "Monday",
                          if_else(
                            day =="Thursday",
                            "Trewsday",
                            if_else(
                              day =="Friday",
                              "Hevensday",
                              if_else(
                                day == "Saturday",
                                "Mersday",
                                if_else(
                                  day == "Sunday",
                                  "Highday",
                                  NA_character_
                                )
                              )
                            )
                          ))))
}
```

using if_else

```
shire_day_from_gregorian_case_when <- function(day) {
  dplyr::case_when(
    day == "Monday" ~ "Sterday",
    day == "Tuesday" ~ "Sunday",
    day == "Wednesday" ~ "Monday",
```

```
    day == "Thursday" ~ "Trewsday",
    day == "Friday" ~ "Hevensday",
    day == "Saturday" ~ "Mersday",
    day == "Sunday" ~ "Highday",
    TRUE ~ NA_character_
  )
}
```

using case__when

```
shire_day_from_gregorian_recode <- function(day) {
  dplyr::recode(day,
                Monday = "Sterday",
                Tuesday = "Sunday",
                Wednesday = "Monday",
                Thursday = "Trewsday",
                Friday = "Hevensday",
                Saturday = "Mersday",
                Sunday = "Highday",
                .default = NA_character_
                )
  }
```

using recode()

If the function argument is not the name of a Gregorian weekday, the function should return `NA`. Which of these three functions would you stylistically prefer? Our preference is `recode()`, as it requires the least number of keystrokes, and looks the cleanest. Further, it also has the least amount of repetition. Ideally, a simpler function would be where we give two arrays, and based on the position matched in the first array, the function would return the same position in the second array. Thus, we came up with a simple function here.

```
gregorian <- c(
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
)
shire <- c(
  "Sterday",
  "Sunday",
  "Monday",
  "Trewsday",
  "Hevensday",
  "Mersday",
  "Highday"
)

recode_array <- function(element, find_in, replace_from) {
  if (element %in% find_in) {
```

```
    replace_from[match(element, find_in)]
  } else {
    NA_character_
  }
}

shire_day_from_gregorian_match <- function(day) {
  recode_array(day, gregorian, shire)
}
```

b) The `bsts` package contains a vector called `weekday.names`. Install the package and test whether `shire_day_from_gregorian(weekday.names)` returns the correct result.

```
shire_day_from_gregorian_if_else(weekday.names)
```

Testing the result

```
## [1] "Highday"   "Sterday"   "Sunday"    "Monday"    "Trewsday"  "Hevensday"
## [7] "Mersday"
```

```
shire_day_from_gregorian_case_when(weekday.names)
```

```
## [1] "Highday"   "Sterday"   "Sunday"    "Monday"    "Trewsday"  "Hevensday"
## [7] "Mersday"
```

```
shire_day_from_gregorian_recode(weekday.names)
```

```
## [1] "Highday"   "Sterday"   "Sunday"    "Monday"    "Trewsday"  "Hevensday"
## [7] "Mersday"
```

```
shire_day_from_gregorian_match(weekday.names)
```

```
## [1] "Highday"   "Sterday"   "Sunday"    "Monday"    "Trewsday"  "Hevensday"
## [7] "Mersday"
```

```
shire_day_from_gregorian_if_else("Hello")
```

Testing NA values

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

```
shire_day_from_gregorian_case_when("Yay")
```

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

```
shire_day_from_gregorian_recode("Hi")
```

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

```
shire_day_from_gregorian_match("Nope")
```

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

2 Measuring run-times with the microbenchmark package

```r
abs_with_if_else <- function(x){
  # return x if x is 0 or positive
  # return -x if x is negative
  dplyr::if_else(x >= 0, x, -x)
}

abs_with_subsetting <- function(x){
  # extract indices of x that are negative
  neg <- (x < 0)
  # replace them with -x[i]
  x[neg] <- -x[neg]
  # return x
  x
}

abs_with_data_type_conversion <- function(x){
  # if x is 0 or positive,
  # ((x>0) - (x<0)) yields TRUE - FALSE,
  # which is converted to 1 - 0 = 1
  # 1 * x returns x

  # if x is negative,
  # ((x>0) - (x<0)) yields FALSE - TRUE,
  # which is converted to 0 - 1 = -1
  # (-1) * x returns -x
  ((x > 0) - (x < 0)) * x
}

abs_with_for_loop <- function(x) {
  # for every element in x
  for (i in seq_along(x)) {
    # if x[i] is negative
    if (x[i] < 0) {
      # replace it with -x[i]
      x[i] <- -x[i]
    }
  }
  x
}

nums <- rnorm(1e6)

microbenchmark::microbenchmark(
  abs(nums),
  abs_with_if_else(nums),
  abs_with_subsetting(nums),
  abs_with_data_type_conversion(nums),
  abs_with_for_loop(nums)
)
```

b) Compare the run-times of five abs() functions

```
## Unit: microseconds
##                                  expr       min         lq      mean    median
```

```
##                           abs(nums)   707.759  1097.962  4005.13  2542.378
##              abs_with_if_else(nums) 43527.725 53094.702 65051.83 57904.257
##           abs_with_subsetting(nums) 14557.960 18062.408 22151.46 20511.992
##   abs_with_data_type_conversion(nums)  7653.933  9844.735 12739.94 11471.897
##               abs_with_for_loop(nums) 48870.720 50621.876 53279.00 52360.822
##          uq       max neval   cld
##    4030.104  56782.71   100  a
##   64305.731 134711.14   100     e
##   24799.613  78515.04   100   c
##   13894.822  77997.63   100  b
##   53357.245 116442.52   100    d
```

c) Comment on your results in (b). It is clear that the default abs function provided in base R is the fastest by a wide margin (the median seems to be a factor 10 better than it's closest competitor: `abs_with_data_type_conversion()`). Thus, it does not seem to be worth it to write our own function to replace `abs()`, no matter what strategy we use. However, if we do write our own function for a less common operation, the `data_type_conversion` seems to be the best bet, in terms of speed, as well as readability to some extent (in temrs of lines of code). We should definitely avoid `for_loop`, as it is the slowest out of the bunch, and requires the most keystrokes. That being said, if `data_type_conversion` is not suitable for the problem, `subetting` seems to be the next best bet.

3 Comparing two functions for calulating Pythagorean sums

```r
pythag_1 <- function(a, b) {
  # applying the normal theorem
  sqrt(a**2 + b**2)
}


pythag_2 <- function(a, b) {
  # absolute values of two inputs
  absa <- abs(a)
  absb <- abs(b)
  # pmax()/pmin() takes multiple vectors as arguments
  # return maximum for pmax() and minimum for pmin()
  p <- pmax(absa, absb)
  q <- pmin(absa, absb)
  ratio <- q / p
  ratio[is.nan(ratio)] <- 1
  # justification given below
  p * sqrt(1.0 + ratio**2)
}
```

$$p \times \sqrt{1.0 + \left(\frac{q}{p}\right)^2} = \sqrt{p^2 \times \left(1.0 + \frac{q^2}{p^2}\right)} = \sqrt{p^2 + q^2}$$

a) What is the purpose of `is.nan(ratio)` in the second-to-last line of `pythag_2()`'s function body? In the case where both `a` and `b` are 0, `p` and `q` will also be 0. Then, for the ratio, $\frac{q}{p}$ will yield `NaN`. Any operation involving `NaN` outputs `NaN`, so we need to replace it with an integer. The integer does not necessarily have to be 1, as it will get multiplied to 0 (by being multiplied by `p`) before being returned anyway.

```r
a <- rnorm(1e5)
b <- rnorm(1e5)

microbenchmark::microbenchmark(
  pythag_1(a, b),
  pythag_2(a, b)
)
```

b) Compare the run-times of `pythag_1()` and `pythag_2()` with the microbenchmark package. Use identical input consisting of long numeric vectors. Summarise your observation as a comment in the R script.

```
## Unit: microseconds
##            expr      min       lq     mean   median       uq        max neval
##  pythag_1(a, b)  540.795 1023.185 1199.732 1101.439 1274.083   7053.523   100
##  pythag_2(a, b) 2883.658 4174.286 6508.225 4487.481 5055.293 140393.033   100
##  cld
##   a
##    b
```

The summary statistics reveal that `pythag_1()` outperforms `pythag_2()` on every benchmark measure (e.g. mean, median, max, min runtime).

For example, on average, the runtime for `pythag_1()` is approximately 4 times shorter (on our R) than that of `pythag2()`. The median runtime for `pythag_1()` is also approximately 4 times shorter than that of `pythag_2()`.

c) By performing numerical tests, find out under which conditions the functions numerically overflow. When do the functions underflow? Comment on the observed differences between `pythag_1()` and `pythag_2()`. This works with both functions.

```r
x_large <- 3e153
y_large <- 4e153

pythag_1(x_large, y_large)
```

```
## [1] 5e+153
```

```r
pythag_2(x_large, y_large)
```

```
## [1] 5e+153
```

However, `pythag_1()` stops working at e154.

```r
x_large <- 3e154
y_large <- 4e154

pythag_1(x_large, y_large)
```

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

```r
pythag_2(x_large, y_large)
```

```
## [1] 5e+154
```

`pythag_2()` still goes strong until e306.

```r
x_large <- 3e306
y_large <- 4e306
```

```r
pythag_1(x_large, y_large)
```

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

```r
pythag_2(x_large, y_large)
```

```
## [1] 5e+306
```

But, after e307, both seem to stop working.

```r
x_large <- 3e307
y_large <- 4e307

pythag_1(x_large, y_large)
```

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

```r
pythag_2(x_large, y_large)
```

```
## [1] 5e+307
```

Similarly, `pythag_1()` fails before `pythag_2()` for lower numbers too.

At e-159, both functions seem to be relatively accurate.

```r
x_small <- 3e-159
y_small <- 4e-159

pythag_1(x_small, y_small)
```

```
## [1] 5e-159
```

```r
pythag_2(x_small, y_small)
```

```
## [1] 5e-159
```

However, `pythag_1()` quickly loses accuracy until e-162

```r
x_small <- 3e-160
y_small <- 4e-160

pythag_1(x_small, y_small)
```

```
## [1] 4.999972e-160
```

```r
pythag_2(x_small, y_small)
```

```
## [1] 5e-160
```

```r
x_small <- 3e-162
y_small <- 4e-162

pythag_1(x_small, y_small)
```

```
## [1] 4.97024e-162
```

```r
pythag_2(x_small, y_small)
```

```
## [1] 5e-162
```

It finally gives up at e-163

```r
x_small <- 3e-163
y_small <- 4e-163

pythag_1(x_small, y_small)
```

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

```r
pythag_2(x_small, y_small)
```

```
## [1] 5e-163
```

`pythag_2()` goes strong until e-324 (with some precision already lost),

```r
x_small <- 3e-324
y_small <- 4e-324

pythag_1(x_small, y_small)
```

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

```r
pythag_2(x_small, y_small)
```

```
## [1] 4.940656e-324
```

before giving up beyond e-325.

```r
x_small <- 3e-325
y_small <- 4e-325

pythag_1(x_small, y_small)
```

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

```r
pythag_2(x_small, y_small)
```

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

d) Is `pythag_1()` or `pythag_2()` better as a general-purpose method? The two functions have their respective advantages, which should be considered in the context of specific use cases.

Advantages of `pythag_1()`: compared to `pythag_2()`, it completes faster and takes up less memory, making it a better choice if a very large number of Pythagorean sums need to be calculated. Its code is also easier to understand and implement.

Advantages of `pythag_2()`: compared to `pythag_1()`, it has better thresholds for both numerical overflow and underflow, making it a better choice if Pythagorean sums for numbers with large magnitude need to be calculated.

For a general-purpose method, it is rare that numbers with very large magnitude need to be dealt with, and thus `pythag_1()` is generally better as it is simpler and faster. However, it is important for us to check that the threshold for numerical overflow or underflow.

Alternatively, we may also consider a simple helper function that takes in the two values, checks the limits to see whether using `pythag_1()` may overflow or underflow, and use `pythag_2()` if that is the case. However, the overhead for checking the overflow and underflow limits must be lesser than the time taken by `pythag_2()` normally for such a function to be considered viable.

4 Floating-point accuracy

```r
0.1 + 0.2
```

```
## [1] 0.3
```

```r
0.1 + 0.2 == 0.3
```

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

```r
round(0.1 + 0.2) == 0.3
```

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

```r
all.equal(0.1 + 0.2, 0.3)
```

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

```r
all.equal(0.1 + 0.2, 0.3, tolerance = .Machine$double.eps)
```

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

```r
all.equal(0.2, 0.3, tolerance = 0.2)
```

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

```r
all.equal(1.1, 10.1, tolerance = 20)
```

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

The default value of tolerance is close to 1.5e-8 [1]. By modifying this value, we can have any two numbers considered equal. Thus, since `(0.1 + 0.2)` cannot be represented accurately in binary, all that `all.equal` does is compare the two values, and ensuring they are within the tolerance.

[1] https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/all.equal