Functions: Part 2

Daniel Anderson

Week 6, Class 2

Agenda

- Review take–home midterm (quickly)
- Purity (quickly)
- Function conditionals
    - `if (condition) {}`
    - embedding warnings, messages, and errors
- Return values

Learning objectives

- Understand the concept of purity, and why it is often desirable
    - And be able to define a side effect
- Be able to change the behavior of a function based on the input
- Be able to embed warnings/messages/errors

Take–home midterm review

Purity

A function is pure if

1. Its output depends *only* on its inputs
2. It makes no changes to the state of the world

Any behavior that changes the state of the world is referred to as a *side–effect*

Note – state of the world is not a technical term, just the way I think of it

Common side effect functions

- We've talked about a few... what are they?

A couple examples

- `print`
- `plot`
- `write.csv`
- `read.csv`
- `Sys.time`
- `options`
- `library`
- `install.packages`

Conditionals

Example

From an old lab:

> Write a function that takes two vectors of the same length and returns the total number of instances where the value is `NA` for both vectors. For example, given the following two vectors

```
c(1, NA, NA, 3, 3, 9, NA)
c(NA, 3, NA, 4, NA, NA, NA)
```

> The function should return a value of `2`, because the vectors are both `NA` at the third and seventh locations. Provide at least one additional test that the function works as expected.

How do you *start* to solve this problem?

~~Start with writing a function~~

Solve it on a test case, then generalize!

Use the vectors to solve!

```
a <- c(1, NA, NA, 3, 3, 9, NA)
b <- c(NA, 3, NA, 4, NA, NA, NA)
```

One approach

```
is.na(a)
```

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

```
is.na(b)
```

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

```
is.na(a) & is.na(b)
```

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

```
sum(is.na(a) & is.na(b))
```

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

Generalize to function

```
both_na <- function(x, y) {
    sum(is.na(x) & is.na(y))
}
```

What happens if not same length?

Test it

```
both_na(a, b)
```

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

```
both_na(c(a, a), c(b, b))
```

```
## [1] 4
```

```
both_na(a, c(b, b)) # ???
```

```
## [1] 4
```

What's going on here?

Recycling

- R will *recycle* vectors if they are divisible

```
data.frame(nums = 1:4,
           lets = c("a", "b"))
```

```
##   nums lets
## 1    1    a
## 2    2    b
## 3    3    a
## 4    4    b
```

- This will not work if they are not divisible

```
data.frame(nums = 1:3,
           lets = c("a", "b"))
```

```
## Error in data.frame(nums = 1:3, lets = c("a", "b")): arguments imply dif
```

Unexpected results

- In the `both_na` function, recycling can lead to unexpected results, as we saw
- What should we do?
- Check that they are the same length, return an error if not

Check lengths

- Stop the evaluation of a function and return an error message with `stop`, but only if a condition has been met.

Basic structure

```
both_na <- function(x, y) {
    if(condition) {
        stop("message")
    }
    sum(is.na(x) & is.na(y))
}
```

Challenge

Modify the code below to check that the vectors are of the same length. Return a *meaningful* error message if not. Test it out to make sure it works!

```
both_na <- function(x, y) {
    if(condition) {
        stop("message")
    }
    sum(is.na(x) & is.na(y))
}
```

02:00

Attempt 1

- Did yours look something like this?

```
both_na <- function(x, y) {
    if(length(x) != length(y)) {
        stop("Vectors are of different lengths")
    }
    sum(is.na(x) & is.na(y))
}
both_na(a, b)
```

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

```
both_na(a, c(b, b))
```

```
## Error in both_na(a, c(b, b)): Vectors are of different lengths
```

More meaningful error message?

What would make it more meaningful?

State the lengths of each

```
both_na <- function(x, y) {
    if(length(x) != length(y)) {
        v_lngths <- paste0("x = ", length(x),
                           ", y = ", length(y))
        stop("Vectors are of different lengths:", v_lngths)
    }
    sum(is.na(x) & is.na(y))
}
both_na(a, c(b, b))
```

```
## Error in both_na(a, c(b, b)): Vectors are of different lengths:x = 7, y
```

Quick error messages

- For quick checks, with usually less than optimal messages, use `stopifnot`
- Often useful if the function is just for you

```
z_score <- function(x) {
  stopifnot(is.numeric(x))
  x <- x[!is.na(x)]
  (x - mean(x)) / sd(x)
}
z_score(c("a", "b", "c"))
```

```
## Error in z_score(c("a", "b", "c")): is.numeric(x) is not TRUE
```

```
z_score(c(100, 115, 112))
```

```
## [1] -1.1338934  0.7559289  0.3779645
```

warnings

If you want to embed a warning, just swap out `stop` for `warning`

Challenge

This is a tricky one

Modify your prior code to so it runs, but returns a warning, if the vectors are recyclable, and returns a meaningful error message if they're different lengths and *not* recylable.

Hint 1: You'll need two conditions

Hint 2: Check if a number is fractional with `%%`, which returns the remainder in a division problem. So `8 %% 2` and `8 %% 4` both return zero (because there is no remainder), while and `7 %% 2` returns 1 and `7 %% 4` returns 3.

06:00

One approach

```
both_na <- function(x, y) {
    if(length(x) != length(y)) {
        lx <- length(x)
        ly <- length(y)

        v_lngths <- paste0("x = ", lx, ", y = ", ly)

        if(lx %% ly == 0 | ly %% lx == 0) {
            warning("Vectors were recycled (", v_lngths, ")")
        }
        else {
            stop("Vectors are of different lengths and are not re
                 v_lngths)
        }
    }
    sum(is.na(x) & is.na(y))
}
```

Test it

```
both_na(a, c(b, b))
```

```
## Warning in both_na(a, c(b, b)): Vectors were recycled (x = 7, y = 14)
```

```
## [1] 4
```

```
both_na(a, c(b, b)[-1])
```

```
## Error in both_na(a, c(b, b)[-1]): Vectors are of different lengths and a
```


Step back
from the
ledge

How important is this?

- For most of the work you do? Not very
- Develop a package? Very!
- Develop functions that others use, even if not through a function? Sort of.

Return values

Thinking more about return values

- By default the function will return the last thing that is evaluated
- Override this behavior with `return`
- This allows the return of your function to be conditional
- Generally the last thing evaluated should be the "default", or most common return value

Pop quiz

- What will the following return?

```
add_two <- function(x) {
    result <- x + 2
}
```

Answer: Nothing! Why?

```
add_two(7)
add_two(5)
```

Specify the return value

The below are all equivalent, and all result in the same function behavior

```
add_two.1 <- function(x) {
    result <- x + 2
    result
}
add_two.2 <- function(x) {
    x + 2
}
```

```
add_two.3 <- function(x) {
    result <- x + 2
    return(result)
}
```

When to use `return`?

Generally reserve `return` for you're returning a value prior to the full evaluation of the function. Otherwise, use `.1` or `.2` methods from prior slide.

Thinking about function names

Which of these is most intuitive?

```
f <- function(x) {
    x <- sort(x)
    data.frame(value = x,
               p = ecdf(x)(x))
}

ptile <- function(x) {
    x <- sort(x)
    data.frame(value = x,
               ptile = ecdf(x)(x))
}
percentile_df <- function(x) {
    x <- sort(x)
    data.frame(value = x,
               percentile = ecdf(x)(x))
}
```

Output

- The descriptive nature of the output can also help
- Maybe a little too tricky but...

```
percentile_df <- function(x) {
    arg <- as.list(match.call())
    x <- sort(x)
    d <- data.frame(value = x,
                    percentile = ecdf(x)(x))

    names(d)[1] <- paste0(as.character(arg$x), collapse = "_")
    d
}
```

```
random_vector <- rnorm(100)
tail(percentile_df(random_vector))
```

```
##     random_vector percentile
## 95       1.951452       0.95
## 96       1.958863       0.96
## 97       2.002071       0.97
## 98       2.003271       0.98
## 99       2.151006       0.99
## 100      2.673912       1.00
```

```
head(percentile_df(rnorm(50)))
```

```
##    rnorm_50 percentile
## 1 -1.549645       0.02
## 2 -1.409034       0.04
## 3 -1.357726       0.06
## 4 -1.293475       0.08
## 5 -1.227892       0.10
## 6 -1.155356       0.12
```

How do we do this?

- I often debug functions and/or figure out how to do something within the function by changing the return value & re–running the function multiple times

[demo]

Thinking about dependencies

- What's the purpose of the function?
    - Just your use? Never needed again? Don't worry about it at all.
    - Mass scale? Worry a fair bit, but make informed decisions.
- What's the likelihood of needing to reproduce the results in the future?
    - If high, worry more.
- Consider using name spacing (`::`)

Next time

Lab 3