Defensive programming with stopifnot

## Learning Objectives

* Check conditions with stopifnot
* Send warnings with warning

As we saw in the debugging section, working functions can still produce unexpected errors. And tracking down the cause of the errors can be difficult even with a debugging tool. This is why the concept of defensive programming is important. Defensive programming encourages us to frequently check conditions and throw an error if something is wrong. These checks are referred to as assertion statements because we want to assert some condition is TRUE before proceeding. This makes it easier to debug because we have a better idea of where the error originated. Recall the uninformative error we received from sum\_calc\_stat: "Error in apply(df\_sub, 1, mean): dim(X) must have a positive length", and the problem was actually introduced at the step before during the subsetting. While it will involve more work when initially writing the code, it will save time in the future when others (including your future self) use the code.

### Checking conditions with stopifnot

To create an error, we can use the function stop. For example, if a variable x needed to be a character vector, we could check for this condition with an if statement and throw an error if the condition was violated.

x <- 1  
if (!is.character(x)) {  
 stop("x must be a character vector.")  
}

Error in eval(expr, envir, enclos): x must be a character vector.

If we had multiple conditions to check, it would take many lines of code to check all of them. Luckily R provides the convenience function stopifnot. We can list as many requirements that should evaluate to TRUE, and stopifnot throws an error if it finds one that is FALSE.

stopifnot(is.character(x))

Error: is.character(x) is not TRUE

If x also had to have a length of one, we can add that condition.

stopifnot(length(x) == 1, is.character(x))

Error: is.character(x) is not TRUE

Listing these conditions also serves a secondary purpose as documentation for the code.

Let's try out defensive programming by adding assertions to check the input to our function mean\_metric\_per\_var.

mean\_metric\_per\_var <- function(metric, variable) {  
 if (!is.factor(variable)) {  
 variable <- as.factor(variable)  
 }  
 variable <- droplevels(variable)  
 result <- numeric(length = length(levels(variable)))  
 names(result) <- levels(variable)  
 for (v in levels(variable)) {  
 result[v] <- mean(metric[variable == v], na.rm = TRUE)  
 }  
 return(result)  
}

We want to assert the following:

* metric is a numeric vector
* metric and variable have the same length

mean\_metric\_per\_var <- function(metric, variable) {  
 stopifnot(is.numeric(metric),  
 length(metric) == length(variable))  
 if (!is.factor(variable)) {  
 variable <- as.factor(variable)  
 }  
 variable <- droplevels(variable)  
 result <- numeric(length = length(levels(variable)))  
 names(result) <- levels(variable)  
 for (v in levels(variable)) {  
 result[v] <- mean(metric[variable == v], na.rm = TRUE)  
 }  
 return(result)  
}

It still works when given proper input.

mean\_metric\_per\_var(counts\_raw$backtweetsCount, counts\_raw$year)

2003 2004 2005 2006 2007 2008   
0.000000000 0.009578544 0.054976303 0.016170763 0.040122277 0.047532408   
 2009 2010   
0.351047202 0.704338789

But fails instantly if given improper input.

# Metric is a factor instead of numeric  
mean\_metric\_per\_var(counts\_raw$journal, counts\_raw$year)

Error: is.numeric(metric) is not TRUE

# The variable vector is shorter than metric  
mean\_metric\_per\_var(counts\_raw$backtweetsCount, counts\_raw$year[1:20])

Error: length(metric) == length(variable) is not TRUE

We can also check the output. Let's add an assertion to ensure that the result does not contain any NAs.

mean\_metric\_per\_var <- function(metric, variable) {  
 stopifnot(is.numeric(metric),  
 length(metric) == length(variable))  
 if (!is.factor(variable)) {  
 variable <- as.factor(variable)  
 }  
 variable <- droplevels(variable)  
 result <- numeric(length = length(levels(variable)))  
 names(result) <- levels(variable)  
 for (v in levels(variable)) {  
 result[v] <- mean(metric[variable == v], na.rm = TRUE)  
 stopifnot(!is.na(result[v]))  
 }  
 return(result)  
}

### Send warnings

In general, if something is wrong, we want to produce an error as quickly as possible to make it easier to debug. However there are situations where it is appropriate to issue a warning to the user without stopping the code. For example, when a non-factor is passed to variable in mean\_metric\_per\_var, we automatically convert it to a factor. Since the user could have simply passed in the wrong vector, we should issue a warning. This alerts the user to double check their input. To do this we use the function warning.

mean\_metric\_per\_var <- function(metric, variable) {  
 stopifnot(is.numeric(metric),  
 length(metric) == length(variable))  
 if (!is.factor(variable)) {  
 warning("variable was automatically converted to a factor.")  
 variable <- as.factor(variable)  
 }  
 variable <- droplevels(variable)  
 result <- numeric(length = length(levels(variable)))  
 names(result) <- levels(variable)  
 for (v in levels(variable)) {  
 result[v] <- mean(metric[variable == v], na.rm = TRUE)  
 stopifnot(!is.na(result[v]))  
 }  
 return(result)  
}

Now when we use a non-factor, the function warns us that it was converted to a factor.

mean\_metric\_per\_var(counts\_raw$backtweetsCount, counts\_raw$year)

Warning in mean\_metric\_per\_var(counts\_raw$backtweetsCount, counts\_raw  
$year): variable was automatically converted to a factor.

2003 2004 2005 2006 2007 2008   
0.000000000 0.009578544 0.054976303 0.016170763 0.040122277 0.047532408   
 2009 2010   
0.351047202 0.704338789

### Challenge

## Practice defensive programming

Use defensive programming techniques to make the function calc\_sum\_stat more robust.

calc\_sum\_stat <- function(df, cols) {  
 df\_sub <- df[, cols, drop = FALSE]  
 sum\_stat <- apply(df\_sub, 1, mean)  
 return(sum\_stat)  
}

Some specific ideas:

* Assert that the input df is not empty (hint: use dim)
* Assert that cols is a character vector
* Assert that the columns listed in cols are in df (hint: use %in% and colnames)
* Assert that df\_sub is a data frame (hint: use is.data.frame)
* Assert that sum\_stat is not NA
* Issue a warning if cols only contains one column (since taking the mean of one column isn't very useful, the user may have made a mistake)

After you add your assertion statements, test out the following inputs to calc\_sum\_stat. Do your assertion statements catch these errors? Should we update the function based on some of these results?

# Empty data frame  
sum\_stat <- calc\_sum\_stat(data.frame(), c("wosCountThru2010", "f1000Factor"))  
# Non-character cols  
sum\_stat <- calc\_sum\_stat(counts\_raw, 1:3)  
# Bad column names  
sum\_stat <- calc\_sum\_stat(counts\_raw, c("a", "b"))  
# Issue warning since only one column  
sum\_stat <- calc\_sum\_stat(counts\_raw, "mendeleyReadersCount")  
# NA output  
sum\_stat <- calc\_sum\_stat(counts\_raw, c("wosCountThru2010", "facebookLikeCount"))

# Because of the `NA` result, use `na.rm = TRUE` with `mean`  
calc\_sum\_stat <- function(df, cols) {  
 stopifnot(dim(df) > 0,  
 is.character(cols),  
 cols %in% colnames(df))  
 if (length(cols) == 1) {  
 warning("Only one column specified. Calculating the mean will not change anything.")  
 }  
 df\_sub <- df[, cols, drop = FALSE]  
 stopifnot(is.data.frame(df\_sub))  
 sum\_stat <- apply(df\_sub, 1, mean, na.rm = TRUE)  
 stopifnot(!is.na(sum\_stat))  
 return(sum\_stat)  
}  
# Proper  
sum\_stat <- calc\_sum\_stat(counts\_raw, c("wosCountThru2010", "f1000Factor"))