

# Week 11 Optimization, Performance Enhancement, and Debugging

```
knitr::opts_knit$set(root.dir = "D:/Introduction-to-R/Data")
library(validate)
library(magrittr)
```

No course on programming would be complete without an in-depth discussion on optimization, performance enhancement, and debugging. R-studio has some excellent debugging tools available, such as setting breakpoints and resuming, that students should use when creating R programs. This week will focus primarily on quality of code, both from a troubleshooting standpoint as well as optimization.

## Weekly Learning Objectives

Create an R script with the following components:

1. Evaluate the execution time of a function in order to optimize the code
2. Create a function with error handling
3. Create a function that uses the validate and magrittr package to validate data
4. Using a previous simulation built in class, perform an optimization on it

### 1. Evaluate the execution time of a function in order to optimize the code

One of the easiest ways to optimize code is to check for execution time. Using the `system.time` function, you can check for the amount of time it takes a block of code to execute.

#### Full Points

Build a simple function that will check the execution time starting and ending.

```
simplyDoSomething <- function(){      # Example function to print out 100 times
  for(j in 1:50){
    print("Do Something Here")
  }
}

system.time(simplyDoSomething())      # Timer call
```

```
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
## [1] "Do Something Here"
```

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

2. `stop()` - Generates errors
3. `suppressWarnings(expr)` - Evaluates the expression and ignores any warnings from it
4. `tryCatch()` - Evaluates code and assigns exceptions

Of these four, the most common in other languages is the `tryCatch()` function.

For this example, I'm going to use the more manual approach and build my own function `error_test()`. It will report if there is an error in the form of having a vector that is below a certain threshold. For testing, I will use the rivers preloaded data set.

## Full Points

You can use either the packaged library method or write your own function. The final product has to use an error handling technique.

```
summary(rivers)           # Generate Summary of Rivers

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    135.0   310.0   425.0   591.2   680.0   3710.0

error_test <- function(vector, threshold) {
  if (min(vector) < threshold) {
    cat("Error: At least one number is below the established threshold")
  } else {
    cat("No Error")
  }
}

# I will use it to test if any of the major North American rivers are less than 100 miles.
error_test(rivers,100)

## No Error

# No Error

# Now I will test with 1,000 miles.
error_test(rivers,1000)
```

```
## Error: At least one number is below the established threshold
```

```
# Error: At least one number is below the established threshold
```

## 3. Validate and Magrittr package to validate data

These are two common libraries used for validating and manipulating data. We have already used packages built on magrittr in previous assignments, so the syntax should be relatively straightforward.

The validate package is extremely useful when validating data. A detailed discussion on the proper use of validate is found in the documentation:

<https://cran.r-project.org/web/packages/validate/vignettes/introduction.html>

Out of the available functions, the most useful are:

1. `check_that()` - Allows multiple parameters to be entered. It will run checks on each to see if the condition is met
2. `validator()` - Store a rule set to compare against

### Full Points

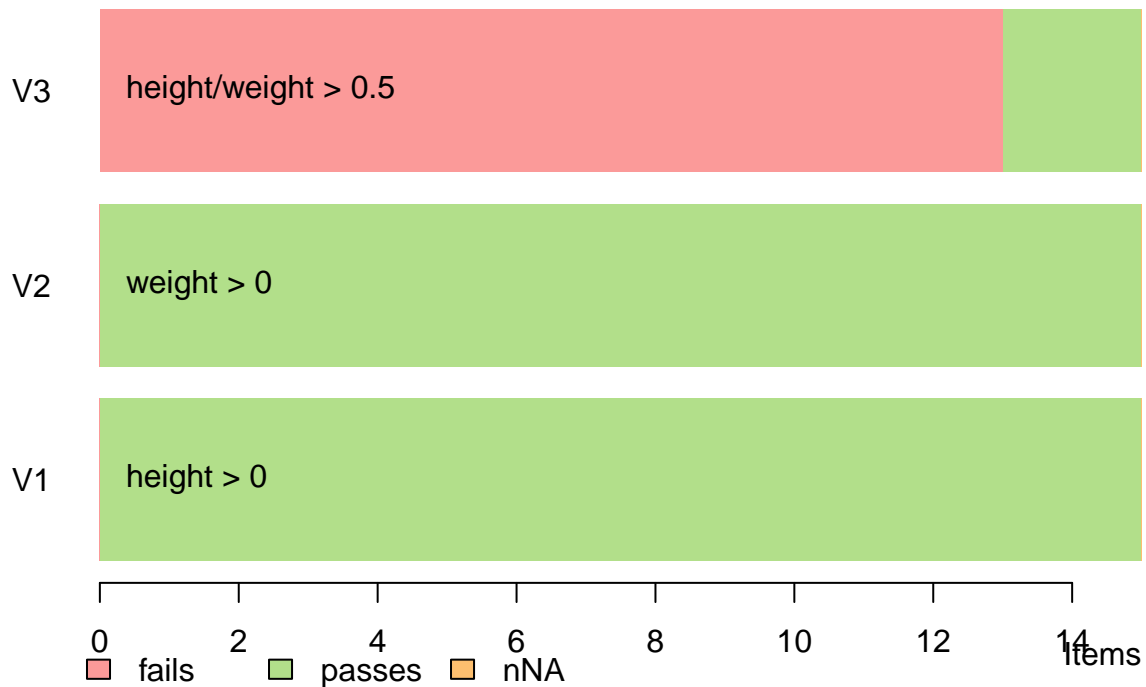
The result should be proper execution of functions in both `validate` and `magrittr`.

```
cf <- check_that(women, height > 0, weight > 0, height/weight > 0.5)
summary(cf)
```

```
##   name items passes fails nNA error warning      expression
## 1  V1    15     15     0   0 FALSE  FALSE      height > 0
## 2  V2    15     15     0   0 FALSE  FALSE      weight > 0
## 3  V3    15      2    13   0 FALSE  FALSE height/weight > 0.5
```

```
barplot(cf)
```

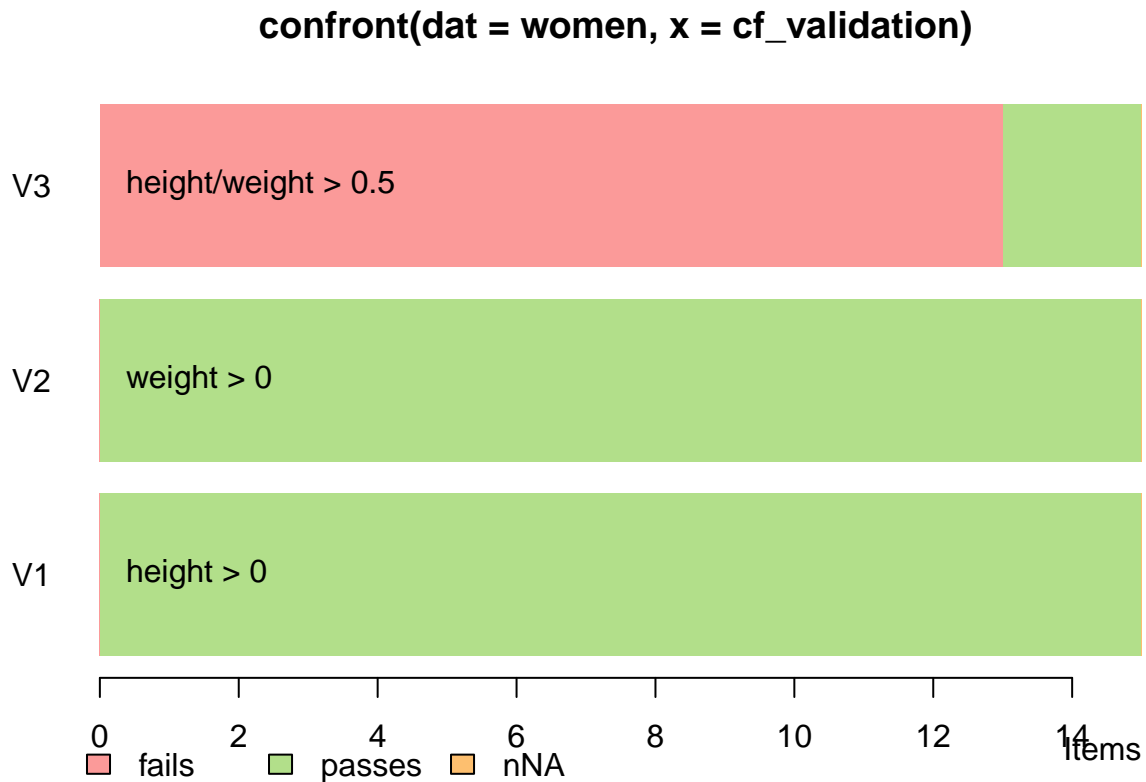
### `check_that(women, height > 0, weight > 0, height/weight > 0.5)`



```
# Using the validate package to set rules.
cf_validation <- validator(height > 0, weight > 0, height/weight > 0.5)
cf_validation
```

```
## Object of class 'validator' with 3 elements:
## V1: height > 0
## V2: weight > 0
## V3: height/weight > 0.5
```

```
# Testing the rules.
confront_data <- confront(women, cf_validation)
barplot(confront_data)
```



#### 4. Optimize a previous simulation

Using any variation of the techniques discussed, use one of your existing simulations and perform an optimization on it. The demonstrator will be a roulette simulation.

##### Full Points

Comparing the before-and-after solutions, there should be optimization between the two.

```
Roulette<-c(0:36)
sample_size<-100
repeat_samples<-10000 # I increased the repeat samples from 1,000 to 10,000 to better display
                        # the time difference in the optimization.

# I have tweaked the for loop to generate one vector combining all of the simulations.
for(i in 1:repeat_samples){
  clt_y<-sample(Roulette,size=sample_size,replace=TRUE)
  if (exists("loop_wheel")==TRUE){
    clt_z <- c(clt_y)
    loop_wheel <- rbind(loop_wheel,clt_z)
  }
}
```

```

    rm(clt_z)
  }else{
    loop_wheel <- c(clt_y)
  }
}
length(loop_wheel)          # Test to make sure length is 1,000,000

```

```
## [1] 1000000
```

```
mean(loop_wheel)           # Mean should be close to 18
```

```
## [1] 18.00711
```

```

# This simulation could be run as a vector:
vector_wheel <- c(sample(Roulette,size=sample_size*repeat_samples,replace=TRUE))
length(vector_wheel)      # Test to make sure length is 1,000,000

```

```
## [1] 1000000
```

```
mean(vector_wheel)        # Mean should be close to 18
```

```
## [1] 18.01238
```

```
# Now to show the time it takes to run each of those functions:
```

```

# First the untuned starting version:
system.time(for(j in 1:repeat_samples){
  clt_y<-sample(Roulette,size=sample_size,replace=TRUE)
  if (exists("sample_x")==TRUE){
    clt_z <- c(clt_y)
    sample_x <- rbind(sample_x,clt_z)
    rm(clt_z)
  }else{
    sample_x <- c(clt_y)
  }
})

```

```
##      user  system elapsed
##  13.29    0.50   13.80
```

```

#      user  system elapsed
#   10.72    0.15   10.90

```

```
# And the vector approach / tuned version
```

```
system.time(vector_wheel <- c(sample(Roulette,size=sample_size*repeat_samples,replace=TRUE)))
```

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

```
#  user  system elapsed
#  0.01   0.00   0.02    <--- Much faster
```