Debugging Techniques

PSS SUMMER SCHOOL

Anton Badev, Daniel Nikolic, Justin Skillman

June 23, 2017

Outline

- Introduction
  - What are bugs?
  - What is debugging?
  - Bug prevention
- Naive techniques vs debugging tools
- Best practices
- Debugging in R
  - `debug()`
  - `browser()`
  - `setbreakpoint()`
  - `trace()`
  - Debugging in RStudio

What are code “bugs”?

- Parts of a program that operate or return results in an unintended fashion
- Types
  - Syntax (= vs ==)
  - Design or structure of code
  - Arithmetic errors
  - Computer resource usage

Syntax bug

- Goal: To perform the `f_analyze` vector-function on each element along with all preceding elements in a list object, `my_list`.

Bugged Code

```
my_list <- as.list(rnorm(10,0,1))
n_list <- length(my_list)
res <- c()
for (ji in 1:n_list) {
  res[ji] <- f_analyze(my_list[[1:ji]])
}
```

Syntax bug

- Goal: To perform the `f_analyze` vector-function on each element along with all preceding elements in a list object, `my_list`.

Bugged Code

```
my_list <- as.list(rnorm(10,0,1))
n_list <- length(my_list)
res <- c()
for (ji in 1:n_list) {
  res[ji] <- f_analyze(my_list[[1:ji]])
}
```

Fixed Code

```
my_list <- as.list(rnorm(10,0,1))
n_list <- length(my_list)
res <- c()
for (ji in 1:n_list) {
  res[ji] <- f_analyze(unlist(my_list[1:ji]))
}
```

- Misuse of indexing into a list
- Though `my_list[[1]]` returns a scalar, `my_list[[1:2]]` *does not* return a vector
- `unlist`: list $\rightarrow$ vector

Structure bug

- Goal: Loop through a vector `my_vec`, applying `func_1` if the prior element is negative and `func_2` if it is non-negative. Both functions are scalar input and output.

Bugged Code

```
n_elem <- 1000
my_vec <- rnorm(n_elem,0,1)
res <- matrix(NA,n_elem,1)
for (ji in 1:n_elem) {
  if (my_vec[ji-1] < 0) {
    res[ji] <- func_1(my_vec[ji])
  } else {
    res[ji] <- func_2(my_vec[ji])
  }
}
```

Structure bug

- Goal: Loop through a vector `my_vec`, applying `func_1` if the prior element is negative and `func_2` if it is non-negative. Both functions are scalar input and output.

Bugged Code

```
n_elem <- 1000
my_vec <- rnorm(n_elem,0,1)
res <- matrix(NA,n_elem,1)
for (ji in 1:n_elem) {
  if (my_vec[ji-1] < 0) {
    res[ji] <- func_1(my_vec[ji])
  } else {
    res[ji] <- func_2(my_vec[ji])
  }
}
```

Fixed Code

```
n_elem <- 1000
my_vec <- rnorm(n_elem,0,1)
res <- matrix(NA,n_elem-1,1)
for (ji in 2:n_elem) {
  if (my_vec[ji-1] < 0) {
    res[ji-1] <- func_1(my_vec[ji])
  } else {
    res[ji-1] <- func_2(my_vec[ji])
  }
}
```

- Results must be smaller by **1** element as it is dependent on the previous iteration.

Arithmetic bug

- Setup: We have a panel dataset on weather and need insight into the relationship of the variables
- Goal: To regress `log(chng_temp)` and `humidity` on our LHS variable `rainfall` (ignore econometric issues)

Data

```
> weather
  rainfall humidity chng_temp
1        6     0.40        10
2        5     0.30        -3
3        2     0.90         5
4        1     0.10        -2
5        5     0.02         7
6       20     0.04         0
```

Bugged Code

```
> lm(rainfall ~ humidity + log(chng_temp))
```

Arithmetic bug

- Setup: We have a panel dataset on weather and need insight into the relationship of the variables
- Goal: To regress `log(chng_temp)` and `humidity` on our LHS variable `rainfall` (ignore econometric issues)

Data

```
> weather
  rainfall humidity chng_temp
1        6     0.40        10
2        5     0.30        -3
3        2     0.90         5
4        1     0.10        -2
5        5     0.02         7
6       20     0.04         0
```

Bugged Code

```
> lm(rainfall ~ humidity + log(chng_temp))
```

- $\log(\mathbb{R}_{<0}) \rightarrow$ `NaN`
- No coding fix, you must change model specification

Computer resource usage bug

- Our data is observed daily for the years 2010-2016 and includes all trades taken in a specific derivatives market
- The flat files files amount to 451 GB
- Bad: Read all of these all into memory in R
- Good: Devise a strategy to *chunk process* the data
- Good: Only read in variables you need for each step in procedure, then remove from memory
- Good: Convert chars to ints by pre-processing in R or bash

What is debugging?

- Process of both finding *and* removing bugs so that code executes correctly
- Steps for a general debugging process
  1. Determine if bug is present
     - Test a smaller data set with known results
     - R warning or error report
  2. Determine bug's place in the code
     - Hand writing pseudo-code is helpful
     - Use R's built-in debugging tools
  3. Reproduce bug and understand its effect
     - What, how, and why things are going wrong
  4. Find a fix
     - Googling error reports
     - Pair programming
  5. Test extensively
  6. Repeat if necessary
- Top skill for programming *effectively*

Exercise

- Each group will be given a snippet of code
- At the end of **7 minutes** each group will speak briefly about their work
- Please do the following things:

1. Find the bug
2. Determine its effect
3. Create a fix and implement
4. Test new output

Bug prevention

- Modular code
- Well commented
- Clear syntax & style
  - Google's R style guide
- Defensive programming
- Code review

Modular code

- Modular code creates function that are independent and entirely self-contained
- Their inputs and outputs should be as simple as possible

Monolithic Code

```
------------FUNCTIONS----------------
main <- function(data) {
  procedure_A
  procedure_B
  procedure_C
  subset_procedure
  return(data)
}
------------MAIN CODE----------------
df <- read.csv("file")
results <- main(df)
```

Modular code

- Modular code creates function that are independent and entirely self-contained
- Their inputs and outputs should be as simple as possible

Monolithic Code

```
------------FUNCTIONS-----------------
main <- function(data) {
  procedure_A
  procedure_B
  procedure_C
  subset_procedure
  return(data)
}
------------MAIN CODE-----------------
df <- read.csv("file")
results <- main(df)
```

Modular Code

```
------------FUNCTIONS-----------------
subset_data <- function(data) {
  subset_procedure
}
det_positions <- function(trades) {
  procedure_A
  procedure_B
}
comp_exposure <- function(positions) {
  procedure_C
  sub_df <- subset_data(data)
}
main <- function(data) {
  my_trades <- det_positions(data)
  my_pos <- det_positions(my_trades)
  res <- comp_exposure(my_pos)
}
------------MAIN CODE-----------------
df <- read.csv("file")
results <- main(df)
```

Defensive programming

- Defensive programming uses techniques to handle unforeseen inputs and outputs
- Creates robustness against misuse
- There is debate about the usefulness of this practice
- Goal: Find the ten largest elements in a vector

Offensive Code

```
find_ten <- function(vec) {
  v_sort <- sort(vec)
  res <- v_sort(1:10)
  return(res)
}
```

Defensive Code

```
find_ten <- function(vec) {
  if (!is.numeric(vec)) {
    stop("Not a numeric vector!")
  }
  if (!is.vector(vec)) {
    stop("Not a vector!")
  }
  if (length(vec)<10) {
    stop("Not enough data!")
  }
  v_sort <- sort(vec)
  res <- v_sort(1:10)
  return(res)
}
```

Naive techniques

- Manual step through execution
- `print()` statements in loops
- Can be inefficient

Code

```
>n <- 100
>for (i in 1:n) {
  sub_data <- data[,i]
  res[i] <- func_1(sub_data)
}
>
```

Debugging code

```
>n <- 100
>for (i in 1:n) {
  sub_data <- data[,i]
  res[i] <- func_1(sub_data)
  print(res[i])
}
[1] 42.5
[1] 51.9
[1] 99.3
[1] NaN
>
```

Built-in functions

- `browser()`
- `trace()`
- `traceback()`
- `debug()`

Debugging with Rstudio

- Provides more interactive features than base R debugging tools
- Let's view the below example in Rstudio

```
> my_func1 <- function(x) my_func2(x + 1)
> my_func2 <- function(x) my_func3(x**2)
> my_func3 <- function(x) (sqrt(x) - 1) + y
>
> my_func1(2)
 Error in my_func3(x^2) : object 'y' not found
>
```

Debugging with Rstudio

- Debug mode
- Analogous to being at point in manual step through where error occurred

```
> my_func1 <- function(x) my_func2(x + 1)
> my_func2 <- function(x) my_func3(x**2)
> my_func3 <- function(x) (sqrt(x) - 1) + y
>
> my_func1(2)
Error in my_func3(x^2) : object 'y' not found
Called from: my_func3(x^2)
Browse[1]> x
[1] 9
```

Summary

- Bugs
- Bug Prevention
- Naive techniques vs debugging tools
- Best practices
- Debugging in R & RStudio