Session 9
Debugging 1

R for Social Data Science

Jeffrey Ziegler, PhD

Assistant Professor in Political Science & Data Science
Trinity College Dublin

Fall 2022

Road map for today

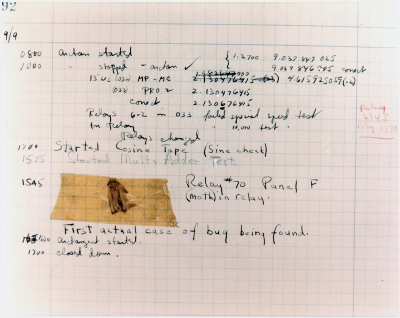
Last week: Functions

- Decomposition and abstraction
- Function definition and function call
- Functionals
- Scoping in R

This time:

- Software bugs
- Debugging

Computer bugs before



Grace Murray Hopper popularised the term *bug* after in 1947 her team traced an error in Mark II to a moth trapped in a relay

Source: US Naval History and Heritage Command

Computer bugs today

```r
even_or_odd <- function(num) {
  if (num %% 2 == 0) {
  return("even")
  } else {
  return("odd")
  }
}
even_or_odd(42.7)
```

```
[1] "odd"
```

```r
even_or_odd('42')
```

```
Error in num%%2: non-numeric argument to binary operator
Traceback:1. even_or_odd("42")
```

Explicit expectations

- Make explicit what kind of input your function expects
- Conditional statements (or type conversion) at beginning help check that

```r
even_or_odd <- function(num) {
  num <- as.integer(num) # We expect input to be integer or convertible into one}
  if (num %% 2 == 0) {
    return("even")
  } else {
    return("odd")
  }
}
even_or_odd(42.7)
```

```
[1] "even"
```

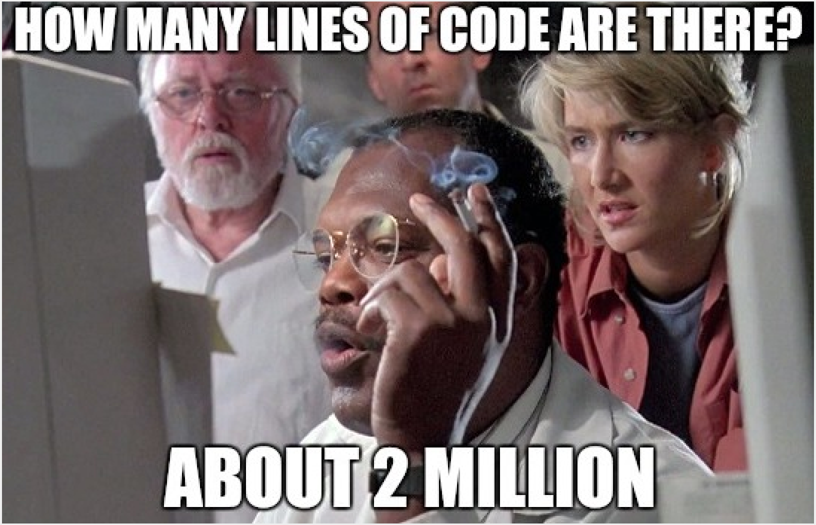
```r
even_or_odd('42')
```

```
[1] "even"
```

Types of bugs

- *Overt* vs *covert*
  - Overt bugs have obvious manifestation (e.g. premature program termination, crash)
  - Covert bugs manifest themselves in wrong (unexpected) results
- *Persistent* vs *intermittent*
  - Persistent bugs occur for every run of the program with same input
  - Intermittent bugs occur occasionally even given the same input and other conditions

Debugging



Debugging

Fixing a buggy program is a process of confirming, one by one, that the many things you believe to be true about the code actually are true. When you find that one of your assumptions is not true, you have found a clue to the location (if not the exact nature) of a bug.

- Norman Matloff

When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth.

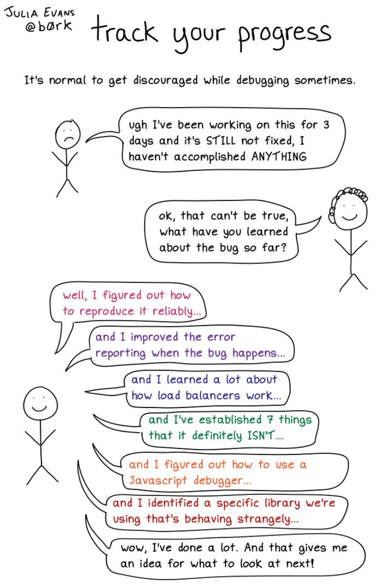
- Arthur Conan Doyle

- Process of finding, isolating and fixing an existing problem in computer program

Debugging process

1. Realise that you have a bug
   - Could be non-trivial for covert and intermittent bugs
2. Make it reproducible
   - Extremely important step that makes debugging easier
   - Isolate the smallest snippet of code that repeats the bug
   - Test with different inputs/objects
   - Will also be helpful if you are seeking outside help
   - Provides a case that can be used in automated testing
3. Figure out where it is
   - Formulate hypotheses, design experiments
   - Test hypotheses on a reproducible example
   - Keep track of the solutions that you have attempted
4. If it worked:
   - Fix the bug and test use-case
5. Otherwise:
   - Sleep on it

Track your progress



Debugging with ‘print()‘

- ‘print()‘ statement can be used to check the internal state of a program during evaluation
- Can be placed in critical parts of code (before or after loops/function calls/objects loading)
- Can be combined with function ‘ls()‘ (and ‘get()‘/‘mget()‘) to reveal all local objects
- For harder cases switch to R debugging functions(‘debug()‘/‘debugonce()‘)

Bug example

```r
calculate_median <- function(a) {
  a <- sort(a)
  n <- length(a)
  m <- (n + 1) %/% 2
  if (n %% 2 == 1) {
    med <- a[m]
  } else {
    med <- mean(a[m:m+1])
  }
  return(med)
}
v1 <- c(1, 2, 3)
v2 <- c(0, 1, 2, 2)
calculate_median(v1)
```

```
[1] 2
```

```r
calculate_median(v2)
```

```
[1] 2
```

Debugging with 'print()'

```r
calculate_median <- function(a) {
  a <- sort(a)
  n <- length(a)
  m <- (n + 1) %/% 2
  print(m)
  if (n %% 2 == 1) {
    med <- a[m]
  } else {
    med <- mean(a[m:m+1])
  }
  return(med)
}
calculate_median(v1) # v1 <- c(1, 2, 3)
```

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

```r
calculate_median(v2) # v1 <- c(1, 2, 3, 2)
```

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

Debugging with ‘print()‘ and ‘ls()‘

```r
calculate_median <- function(a) {
  a <- sort(a)
  n <- length(a)
  m <- (n + 1) %/% 2
  # Print all objects in function environment
  print(mget(ls()))
  if (n %% 2 == 1) {
    med <- a[m]
  } else {
    med <- mean(a[m:m+1])
  }
  return(med)
}
calculate_median(v1)
```

```
$a
[1] 1 2 3
$m
[1] 2
$n
[1] 3
[1] 2
```

Debugging with ‘print()‘ and ‘ls()‘

```
calculate_median(v2)
```

```
$a
[1] 0 1 2 2
$m
[1] 2
$n
[1] 4
[1] 2
```

Debugging with 'print()'

```r
calculate_median <- function(a) {
  a <- sort(a)
  n <- length(a)
  m <- (n + 1) %/% 2
  if (n %% 2 == 1) {
    med <- a[m]
  } else {
    print(m-1:m)
    med <- mean(a[m:m+1])
  }
  return(med)
}
calculate_median(v1)
```

```
[1] 2
```

```r
calculate_median(v2)
```

```
[1] 1 0
[1] 2
```

Fixing a bug and confirming

```r
calculate_median <- function(a) {
  a <- sort(a)
  n <- length(a)
  m <- (n + 1) %/% 2
  if (n %% 2 == 1) {
    med <- a[m]
  } else {
    med <- mean(a[m:(m+1)])
  }
  return(med)
}
calculate_median(v1)
```

```
[1] 2
```

```r
calculate_median(v2)
```

```
[1] 1.5
```

R Debugging Facilities

- The core of R debugging process is stepping through the code as it gets executed
- This permits the inspection of the environment where a problem occurs
- Three functions provide the the main entries into the debugging mode:
  - ‘browser()‘ - pauses the execution at a dedicated line in code (breakpoint)
  - ‘debug()‘/‘undebug()‘ - (un)sets a flag to run function in a debug mode (setting through)
  - ‘debugonce()‘ - triggers single stepping through a function

Breakpoints

```r
calculate_median <- function(a) {
  a <- sort(a)
          n <- length(a)
  m <- (n + 1) %/% 2
   if (n %% 2 == 1) {
    med <- a[m]
          } else {
    browser()
            med <- mean(a[m:m+1])
   }
          return(med)
}
calculate_median(v2)
```

```
Called from: calculate_median(v2)
debug at <text>#9: med <- mean(a[m:m + 1])
debug at <text>#11: return(med)
```

Debugger commands

Command	Description
‘n(ext)‘	Execute next line of the current functiom
‘s(tep)‘	Execute next line, stepping inside the function (if present)
‘c(ontinue)‘	Continue execution, only stop when breakpoint in encountered
‘f(inish)‘	Finish execution of the current loop or function
‘Q(uit)‘	Quit from the debugger, executed program is aborted

More resources

Debug a function

- ‘debugonce()‘ function allows to run and step through the function

```
debugonce(<function_name>, <*args>, <**kwargs>)
```

Tutorial - Debugging with ‘print()‘

- ‘print()‘ statement can be used to check the internal state of a program during evaluation
- Can be placed in critical parts of code (before or after loops/function calls/objects loading)
- For harder cases switch to R debugger

Exercise: Debug function - Pearson correlation

- See function for calculating Pearson correlation below)
- Recall that sample correlation can be calculated using this formula:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{x}$ and $\bar{y}$ are the means (averages) of variable x and y, respectively)

- What do you think is correlation coefficient between vectors ‘c(1, 2, 3, 4, 5)‘ and ‘c(-3, -5, -7, -9, -11)‘?
- Check output of function, is it correct?
- Find and fix any problems that you encounter

Exercise: Use debugger to fix function

- Look at the problematic ‘calculate_sd‘ function
- Run R debugger and step through it
- While inside function print out values of deviations and result of stand_dev
- Fix bug(s)

Overview

This time:

- Software bugs
- Debugging

Next time:

- Handling conditions
- Testing
- Defensive programming