Debugging: Find the mistakes

Debugging and Software Testing

Two ways to find/prevent bugs:

1. **Debugging**
   - Finds the source of a programming flaw
   - Helps understand program execution
2. **Software Testing**
   - Standardized means for quality and correctness checks
   - Can be used to specify requirements
   - Assess usability of program interfaces

Rule of thumb: debugging consumes about 2/3 of the development time!

Debugging

- … is recommended when the return value (e.g., of a unit test) is incorrect and the error is not obvious
- … uses tools that examine the control flow and values of variables in a program
- Many programming environments support line-by-line execution debugging, where only one line of code at a time is executed

Debugging strategy

1. Realize that you have a bug
2. Reproduce/generate input values that cause the bug
3. Isolate the flawed component with a binary search
4. Fix it
5. Confirm its successful resolution using the previous input (when using unit testing, create an automated test

Debugging in R

Fixing bugs

- Output variables to the screen

  (e.g., `print(...)` or `browser()` for an interactive session)
- Using built-in commands in R (e.g., `traceback()` for the call stack)
- Interactive debugger inside R Studio

Preventing bugs

- Exception handling

Preventing those pesky bugs browser() and Asserts

`browser()` interrupts the execution of an expression and allows you to inspect the environment of where `browser()` was called from.

```
for (i in 1:5){
  x = i + 1
  print(i)
  browser()
}
```

For each break, you can call a variable, i.e., *i* or *x*. In the end, R will always return the result.

Useful if you want to see values of variables at certain times during code execution

```
[1] 1
Called from: top level
Browse[1]> i
[1] 1
Browse[1]> x
[1] 2
Browse[1]>
[1] 2
Called from: top level
Browse[1]> i
[1] 2
Browse[1]>
[1] 3
Called from: top level
Browse[1]> x
[1] 4
Browse[1]>
[1] 4
```

Preventing those pesky bugs
Exception handling and traceback()

`traceback()` prints the call stack of the last uncaught error (i.e., the sequence of calls that lead to the error).

```
> printUpTo("0")
Error in printUpTo("0") :
is.numeric(num) is not TRUE
```

Show Traceback
Rerun with Debug

```
> traceback()
3: stop(simpleError(msg, call = sys.call(-1)))
2: stopifnot(is.numeric(num)) at #3
1: printUpTo("0")
```

`try()` is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery (won't stop execution of the program).

```
try("a" + 5)
```

Here, you want to try to add two things together, and if they can't be added together (error is thrown), the program will keep running instead of crashing

Debugger inside R Studio

1. Insert breakpoints (where R will pause running the program)

2. Debug file

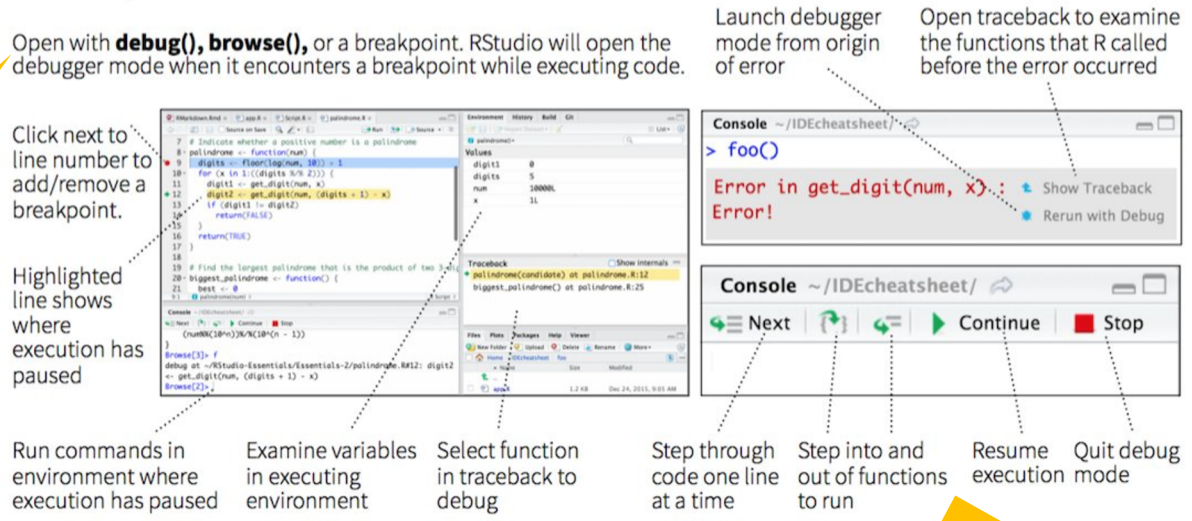
Useful commands in base R

```
debug(), undebug()
debugonce()
browser()
traceback()
options(error = browser), options(error = NULL)
```

Debugging Shiny applications

Automatic `traceback()` in error output, in RStudio and application log
Can set a breakpoint in the server function
Use `browser()` everywhere else (ui, sourced file, etc.)
`options(shiny.error = browser)`
Tracing: `cat(file=stderr(),...), options(shiny.reactlog=TRUE)`

Debug mode in RStudio (from the IDE cheat sheet)



https://rstudio.com/resources/cheatsheets/

Resources

Debugging with RStudio
https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio

Debugging Shiny applications
https://shiny.rstudio.com/articles/debugging.html

"Debugging, condition handling, and defensive programming" in *Advanced R*
http://adv-r.had.co.nz/Exceptions-Debugging.html

3. To navigate through program, use buttons or commands: next (n), step info (s), continue (c) and Stop (Q)

For hardcore command line users: debug ()

- The function `debug()` in R allows the user to step through the execution of a function, line by line. At any point, we can print out values of variables or produce a graph of the results within the function.

- While debugging, we can simply type "`c`" to continue to the end of the current section of code. `traceback()` does not tell us where the error occurred in the function. In order to know which line causes the error, we will have to step through the function using `debug()`.

Now it’s your turn!