R: Debugging

Debashis Ghosh

October 7, 2019

Overview

- Debugging: how to track down errors in your code
- Conditions: errors, warnings and messages to the user (anticipated problems)
- Condition Handling: do specific actions when specific conditions are satisfied.
- Defensive programming: make code fail when something unexpected occurs.

Key R functions

- **warning()**: returns a warning but still executes function
- **stop()**: stops execution of the current expression and executes an error action
- **message()**: generates a diagnostic message from its arguments

Recipe for Debugging

- Realize that you have a bug
  1. See `http://r-pkgs.had.co.nz/tests.html` for advanced reading.
- Make the bug repeatable
- Figure out where it is
- Fix it and test it

Some concepts/functions in R

- **call stack:** sequence of function calls that gave rise to the error
- **traceback()**: R function that lists out the function calls
- **browser()**: start an interactive console in the environment where the function occurred
- Example in lec8code.R

Other functions in R

- **recover()**: allows the user to enter the environment of any calls in the call stack
- **dump.frames()**: It creates a **last.dump.rda** file in the current working directory
- Example in lec8code.R

Breakpoints

- Useful to enter interactive consoles at arbitrary locations in the code; can be done easily in RStudio by
- Related: add **browser()** to your code where you want it to pause.
- Some caveats:
  - Sometimes breakpoints won't work
  - RStudio does not support conditional breakpoints (however, you can add **browser()** inside an if statement)

Types of failure

- Returns failure
- Returns incorrect result
- Warnings (use **options(warnings=2)** to convert to an error)
- Unexpected message (use **message2error()** function in lec8code.R to convert to an error)
- A function might not return (terminate and look at the call stack)
- R crashes!

Condition handling

- Planned errors
- Three types of tools
  1. **try()** gives you the ability to continue execution even when an error occurs.
  2. **tryCatch()** lets you specify handler functions that control what happens when a condition is signalled.
  3. **withCallingHandlers()** is a variant of **tryCatch()** that runs its handlers in a different context. It?s rarely needed, but is useful to be aware of.

try()

- **try()** allows execution to continue even after an error has occurred
- To pass larger blocks of code to **try()**, wrap them in { }
- Associated with objects of class "try-error" if the command fails
- Another useful **try()** idiom is using a default value if an expression fails. Simply assign the default value outside the try block, and then run the risky code:

tryCatch()

- A very flexible tool
- Can take different actions for errors, warnings and messages
- Maps to different functions depending on the type of error/warning/message that gets invoked.
- Maps conditions to **handlers**, special functions that take the condition as the input; this can be useful for creating more informative error/warning messages
- Has a **finally** argument that will execute a block of code that will run regardless of whether the initial expression succeeds or fails

withCallingHandlers()

- Similar in spirit to **tryCatch()**
- Differences from **tryCatch()**:
  1. The return value of **tryCatch()** handlers is returned by **tryCatch()**, whereas the return value of **withCallingHandlers()** handlers is ignored
  2. The handlers in **withCallingHandlers()** are called in the context of the call that generated the condition whereas the handlers in **tryCatch()** are called in the context of **tryCatch()**
- Can also create customized classes for condition handling.

Defensive Programming

- Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs
- "Fail fast"
  1. Be strict about what you accept
  2. Avoid functions that use non-standard evaluation
  3. Avoid functions that return different types of output depending on their input