A Function for R Session Scripting

Ross Ihaka

Abstract

This document describes an R command called "`script`" that fullfils the same role as the Unix command of the same name. The command diverts a copy of all the activity that takes place on the console in an R session (user input, R output and error messages) to a file specified as the argument to the function.

The document describes both the use of the command and also provides a literate version of the function itself.

1 The "`script`" Function

Once the "`script`" function is loaded,[1] it can be invoked either with no arguments or with a single argument giving the name of a file. If no argument is specified, the R session is recorded in a file called "`transcript.txt`." If a filename is specified the session is recorded in that file.

When "`script`" is invoked, a sub-interpreter is run to process the user's commands. When this sub-interpreter is running, the the R command prompt is changed to "`script> `" and the continuation prompt to "`script+ `". The sub-interpreter is exited by typing the command "`q()`".

```
> script()
Script started, file is "transcript.txt"
script> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
script> max(rnorm(100))
[1] 2.592984
script> q()
Script done, file is "transcript.txt">
```

A record of everything that transpired between "`script()`" and "`q()`" is stored in the file "`transcript.txt`" along with timestamps indicating when scripting started and finished.

This is essentially all there is to know about using "`script`," other than to note that a "`script`" command cannot be run from a `script` sub-interpreter.[2]

[1]It seems like overkill to provide this function in an R package. It can easily be loaded with the "`source`" function.

[2]There is nothing technical that prevents this; it's just likely to be confusing for users to try to use multiple script levels.

2 Implementation of the "`script`" Function

The "`script`" function is implemented as a *closure*. The support functions it uses are encapsulated in a private environment, visible only to that function. The mechanism used is as follows.

2a ⟨*script.R* 2a⟩≡
```
  ⟨comments-and-copyright 13b⟩
  script =
      local({
          ⟨warning state variables 11b⟩
          ⟨support functions 9a⟩
          ⟨read-eval-print loop 4⟩
          ⟨main function 2b⟩
      })
```
This code is written to file `script.R`.

2.1 The Main Function

The main function takes a single argument that specifies the name of the file used to store the transcript. It carries out basic housekeeping tasks and also calls the main workhorse function that provides a read-eval-print-loop.

2b ⟨*main function* 2b⟩≡
```
  function(file = "transcript.txt") {
      ⟨open file connections 2c⟩
      ⟨print startup messages 2d⟩
      ⟨call the read-eval-print-loop 3a⟩
      ⟨print shutdown messages 3b⟩
      ⟨close file connections 3c⟩
      ⟨return an invisible value 3d⟩
  }
```

The function opens two file connections. The first of these is the connection to the transcript file. The second is to a "scratch" file used to capture R output and error messages. Using an empty name for this file means that no cleanup is required when transcript recording finishes.

2c ⟨*open file connections* 2c⟩≡
```
  transcon = file(file, "w")
  outcon = file("")
```
Defines:
  `outcon`, used in chunks 3–8 and 11a.
  `transcon`, used in chunks 2–8, 10c, and 11a.

If the creation of the file connections is successful, the function writes two messages; one to the terminal and one to the transcript file. The second message contains a time stamp. This can be useful when the record is viewed at a later time.

2d ⟨*print startup messages* 2d⟩≡
```
  cat("Script started, file is \"", file, "\"\n", sep = "")
  cat("Script started on", date(), "\n", file = transcon)
```
Uses `transcon` 2c.

Now, everything is ready to call the read-eval-print loop. The first argument passed to the"`repl`" function is the environment that the "`script`" function was called from. This will almost always be the R global enviroment. The other two arguments are the file connections opened above.

3a ⟨*call the read-eval-print-loop* 3a⟩≡
```
  repl(sys.parent(), transcon, outcon)
```
Uses `outcon` 2c, `repl` 4, and `transcon` 2c.

When the read-eval-print loop terminates, messages are again printed to the transcript and the terminal. The transcript message is timestamped.

3b ⟨*print shutdown messages* 3b⟩≡
```
  cat("Script done on", date(), "\n", file = transcon)
  cat("Script done, file is \"", file, "\"\n", sep = "")
```
Uses `transcon` 2c.

Finally, the file connections are closed and the function returns an invisible (`NULL`) value.

3c ⟨*close file connections* 3c⟩≡
```
  close(outcon)
  close(transcon)
```
Uses `outcon` 2c and `transcon` 2c.

3d ⟨*return an invisible value* 3d⟩≡
```
  invisible()
```

2.2 The Read-Eval-Print Loop

The "`repl`" function takes over the role of the topmost level of functionality in R. It reads the lines of text that the user types, parses them and evaluates the results. It also has to handle exceptional conditions such as errors, warnings and user interrupts.

There are two important strategies employed in this function. The first of these is to use the "`sink`" function to divert user output and error messages to the scratch file attached to the "`outcon`" connection. These can then be read back and written to the transcript file. The logic used is as follows.

```
sink(outcon, split = TRUE)
activity that does some printing
sink()
seek(outcon, 0)
writeLines(readLines(outcon), transcon)
```

The second strategy is used to accumulate the lines the user types until a complete expression has been read. Reading the lines is easy; it is done with "`readLines`". Checking for a complete expression is trickier because parsing an incomplete expression trips an error. These must be caught using the "`tryCatch`" mechanism and this type of error discriminated from other syntax errors.

There is also the problem of user interrupts. These can occur at any point in the read-eval-print process. To protect against such interrupts the whole read-eval-print process is embedded in a loop whose sole task is to catch and process interrupts.

The general structure of the "`repl`" function is shown by the following function. The depth of the current "`sink`" diversion is recorded in "`sinkdepth`", initial values are defined for the command prompt and the current expression and then the interrupt catching loop is run.

4 ⟨*read-eval-print loop* 4⟩≡

```
repl =
    function(env, transcon, outcon) {
        sinkdepth = sink.number()
        prompt = "script> "
        cmd = character()
        repeat {
            ⟨interrupt catching 5a⟩
        }
    }
```

Defines:
- `cmd`, used in chunks 5, 6, and 10c.
- `prompt`, used in chunks 5, 6, and 9a.
- `repl`, used in chunk 3a.
- `sinkdepth`, used in chunk 5a.

Uses `outcon` 2c and `transcon` 2c.

The code inside the "`repeat`" loop, in the function above, runs the *repl* and catches any interrupts that occur with a "`tryCatch`" statement. The statement catches just interrupts and it takes one of two actions. If there is an output diversion in place (i.e. if `sink.number() > sinkdepth`) then the interrupt occurred during evaluation of an expression. In this case, the diversion is terminated and the the contents of the diversion are sent to the transcript file. If there is no diversion in place, the interrupt (probably) occured during parsing and we need to copy the (possibly incomplete) command to the transcript. (Note that this code uses the "`sink`" diversion depth recorded on entry to "`repl`" as well as the command buffer "`cmd`".)

5a ⟨*interrupt catching* 5a⟩≡
```
ans = tryCatch(repeat {
    ⟨parse and evaluate expressions 5b⟩
}, interrupt = function(x) x)
if (inherits(ans, "interrupt")) {
    if (sink.number() > sinkdepth) {
        sink()
        echoOutput(transcon, outcon)
    }
    else
        echoCommands(cmd, transcon)
    cat("\nInterrupt!\n")
    cat("Interrupt!\n", file = transcon)
    prompt = "script> "
    cmd = character()
}
else
    stop("Interrupt catcher caught non-interrupt")
```
Uses `cmd` 4, `echoCommands` 10c, `echoOutput` 11a, `outcon` 2c, `prompt` 4, `sinkdepth` 4, and `transcon` 2c.

Expressions are read and processed in a loop. A pass through the loop reads a single line of input with `readLine` and adds it to the "`cmd`" buffer. Each time a line is added, an attempt is made to parse the contents of "`cmd`" and obtain a valid expression for evaluation. The parse is wrapped in a "`tryCatch`" to trap any parsing errors that occur. The result of this attempted parse determines what happens next.

5b ⟨*parse and evaluate expressions* 5b⟩≡
```
repeat {
    cmd = c(cmd, readLine(prompt))
    ans = tryCatch(parse(text = cmd),
        error = function(e) e)
    ⟨handle the results of the parse 6a⟩
}
```
Uses `cmd` 4, `prompt` 4, and `readLine` 9a.

The result returned by the "`tryCatch`" is either a valid expression that can be evaluated or an error condition. We branch depending on the type of result obtained.

6a ⟨*handle the results of the parse* 6a⟩≡
```
  ⟨if there was an error deal with it 6b⟩
  ⟨otherwise handle the expression 6c⟩
```

There are two possible types of error to deal with. Errors can be caused by an incomplete parse or by some other type of syntax error. If the expression is incomplete, we change the prompt to indicate continuation and return to the top of the loop to fetch another line of input. If there was some other type of error, we emit the currently buffered input lines to the transcript. Then we deal with the error and emit any resulting output. Finally, we reset the command prompt and the state of the input buffer.

6b ⟨*if there was an error deal with it* 6b⟩≡
```
  if (inherits(ans, "error")) {
      if (incompleteParse((ans))) {
          prompt = "script+ "
      }
      else {
          echoCommands(cmd, transcon)
          sink(outcon, split = TRUE)
          handleParseError(ans)
          sink()
          echoOutput(transcon, outcon)
          prompt = "script> "
          cmd = character()
      }
  }
```
Uses `cmd` 4, `echoCommands` 10c, `echoOutput` 11a, `handleParseError` 9c, `incompleteParse` 9b, `outcon` 2c, `prompt` 4, and `transcon` 2c.

If there was no error, we have a valid expression and we echo it to the transcript. We then choose between a number of special cases (such as quitting the transcript process) and the general case of evaluating the expression typed by the user. When that is complete, we reset the command prompt and the state of the command buffer before continuing on to read the next expression.

6c ⟨*otherwise handle the expression* 6c⟩≡
```
  else {
      echoCommands(cmd, transcon)
      ⟨handle special expression cases 7⟩
      ⟨handle the general expression case 8⟩
      prompt = "script> "
      cmd = character()
  }
```
Uses `cmd` 4, `echoCommands` 10c, `prompt` 4, and `transcon` 2c.

If the expression was empty (the user idly typed the enter key) we simply go back to fetch another expression. If the user typed `q()` then we exit from the repl and return to the top-level function. If for some reason the user tried to invoke `script` when session is already being recorded we issue an error. (This probably needs further thought.)

7 ⟨*handle special expression cases* 7⟩≡
```
if (length(ans) == 0) {
    break
}
else if (isQuitCall(ans)) {
    return()
}
else if (grepl("^script\\(",
               deparse(ans[[1]], nlines = 1))) {
    sink(outcon, split = TRUE)
    cat("Error: You can’t call \"script\" while scripting\n")
    sink()
    echoOutput(transcon, outcon)
    break
}
```
Uses `echoOutput` 11a, `isQuitCall` 13a, `outcon` 2c, and `transcon` 2c.

If none of these special cases hold, we are in the general situation. We evaluate the expression that the user typed and print the answer. Note that it is possible for parsing to produce several calls in the expression returned from the parse. (Such calls are separated by semicolons.) To handle the general case, we loop over the elements of the expression evaluating and printing each one in turn.

Evaluation is carried out inside a `tryCatchWithWarnings` call. This means that any warnings that occur are recorded (in the variables `warningCalls` and `warningMessages`). After evaluation, a check is made of whether any new warnings have been issued. If there were, the warnings are transferred to the global variable `last.warning`. There, they can be accessed with calls to the function `warnings`. Finally, a call is made to `displayWarnings` to display the warning messages in the correct way.

8 ⟨*handle the general expression case* 8⟩≡

```
else {
    renewwarnings <<- TRUE
    newwarnings <<- FALSE
    for(e in ans) {
        sink(outcon, split = TRUE)
        e = tryCatchWithWarnings(withVisible(eval(e,
            envir = env)))
        if (inherits(e, "error"))
            handleError(e)
        else
            handleValue(e)
        sink()
        echoOutput(transcon, outcon)
    }
    if (newwarnings) {
        warnings = warningCalls
        names(warnings) = warningMessages
        assign("last.warning",
               warnings[1:nwarnings],
               "package:base")
        sink(outcon, split = TRUE)
        displayWarnings(nwarnings)
        sink()
        echoOutput(transcon, outcon)
    }
}
```

Uses `displayWarnings` 12b, `echoOutput` 11a, `handleError` 10a, `handleValue` 10b, `newwarnings` 11b, `nwarnings` 11b, `outcon` 2c, `renewwarnings` 11b, `transcon` 2c, `tryCatchWithWarnings` 12a, `warningCalls` 11b, and `warningMessages` 11b.

2.3 Parsing Support Functions

The lines sent to parser are fetched by calling "`readLine`". (We can't use "`readline`" because it trims spaces and we are trying to preserve the user's layout.)

9a ⟨*support functions* 9a⟩≡
```
readLine =
    function(prompt) {
        cat(prompt)
        flush(stdout())
        readLines(n = 1)
    }
```
Defines:
  `readLine`, used in chunk 5b.
Uses `prompt` 4.

An incomplete parse is detected when the result of the parse is an error that contains the string `"unexpected end of input"`.

9b ⟨*support functions* 9a⟩+≡
```
incompleteParse =
    function(e)
    (inherits(e, "error") &&
     grepl("unexpected end of input", e$message))
```
Defines:
  `incompleteParse`, used in chunk 6b.

The most complicated support function is the one that handles the printing of error messages from parsing. Because the parse is taking place using a character vector as input, the error messages produced look rather different from those produced when the parser gets its input from the console. This function transforms the error messages into that form.

9c ⟨*support functions* 9a⟩+≡
```
handleParseError =
    function(e) {
        msg = strsplit(conditionMessage(e), "\n")[[1]]
        errortxt = msg[1]
        msg = gsub("[0-9]+: ", "", msg[-c(1, length(msg))])
        msg = msg[length(msg) - 1:0]
        if (length(msg) == 1)
            msg = paste(" in: \"", msg, "\"\n", sep = "")
        else
            msg = paste(" in:\n\"",
                paste(msg, collapse = "\n"),
                "\"\n", sep = "")
        cat("Error",
            gsub("\n.*", "",
                 gsub("<text>:[0-9]+:[0-9]+", "",
                      errortxt)),
            msg, sep = "")
    }
```
Defines:
  `handleParseError`, used in chunk 6b.

2.4 Input-Output Support

The error messages produced during evaluation are easy to process. We simply cat them to the (split) output.

10a ⟨*support functions* 9a⟩+≡
```
handleError =
    function(e) {
        cat("Error in", deparse(conditionCall(e)),
            ":", conditionMessage(e), "\n")
    }
```
Defines:
`handleError`, used in chunk 8.

Printing the values that result from evaluating expressions has one wrinkle to it. We have to check the visibility of the result and only print "visible" results.

10b ⟨*support functions* 9a⟩+≡
```
handleValue =
    function(e) {
        if (e$visible) {
            print(e$value)
        }
    }
```
Defines:
`handleValue`, used in chunk 8.

This function echos any accumulated commands to the transcript. They have already appeared on the console so there is no need to echo them there.

10c ⟨*support functions* 9a⟩+≡
```
echoCommands =
    function(cmd, transcon) {
        cat(paste(c("> ",
                    rep("+ ", max(length(cmd) - 1), 0)),
                  cmd, "\n", sep = ""), sep = "",
            file = transcon)
    }
```
Defines:
`echoCommands`, used in chunks 5 and 6.
Uses `cmd` 4 and `transcon` 2c.

The `echoOutput` function echos output to the transcript file. To do this we rewind the output collection connection and echo its contents to the transcript.

11a ⟨*support functions* 9a⟩+≡
```
echoOutput =
    function(transcon, outcon) {
        seek(outcon, 0)
        lines = readLines(outcon, warn = FALSE)
        writeLines(lines, transcon)
        seek(outcon, 0)
        truncate(outcon)
    }
```
Defines:
`echoOutput`, used in chunks 5–8.
Uses `outcon` 2c and `transcon` 2c.

2.5 Warning Support

A number of top-level closure variables are used to manage the warning messages produced by evaluation of expressions. The following variables manage the accumulation of error messages.

`warningCalls`	holds the calls that produced warnings
`warningMessages`	holds the warning messages
`nwarnings`	the number or warnings accumulated
`renewwarnings`	purge the warning list on next warning?
`newwarnings`	has the evaluation produced new warnings

The variables are initialised as follows.

11b ⟨*warning state variables* 11b⟩≡
```
warningCalls = vector("list", 50)
warningMessages = character(50)
nwarnings = 0
renewwarnings = TRUE
newwarnings = FALSE
```
Defines:
`newwarnings`, used in chunks 8 and 12a.
`nwarnings`, used in chunks 8 and 12.
`renewwarnings`, used in chunks 8 and 12a.
`warningCalls`, used in chunks 8 and 12a.
`warningMessages`, used in chunks 8 and 12a.

Warnings are trapped by the following two functions. The effect is to simply add warnings to the accumulated list of warnings and then call the built-in `muffleWarning` restart.

12a ⟨*support functions* 9a⟩+≡
```
  warningHandler = function(w) {
      newwarnings <<- TRUE
      if (renewwarnings) {
          renewwarnings <<- FALSE
          nwarnings <<- 0
      }
      n = nwarnings + 1
      if (n <= 50) {
          warningCalls[[n]] <<- conditionCall(w)
          warningMessages[n] <<- conditionMessage(w)
          nwarnings <<- n
      }
      invokeRestart("muffleWarning")
  }
  tryCatchWithWarnings =
      function(expr)
      withCallingHandlers(tryCatch(expr,
              error = function(e) e),
          warning = warningHandler)
```
Defines:
`tryCatchWithWarnings`, used in chunk 8.
Uses `newwarnings` 11b, `nwarnings` 11b, `renewwarnings` 11b, `warningCalls` 11b, and `warningMessages` 11b.

The `displayWarnings` function is used to display warnings at the end of an evaluation. If there are 10 or fewer messages they are displayed. If there are more than 10 messages, the user is told to inspect them with "`warnings()`". Only the first 50 messages are stored.

12b ⟨*support functions* 9a⟩+≡
```
  displayWarnings =
      function(n) {
          if (n <= 10)
              print(warnings())
          else if (n < 50) {
              cat("There were",
                  nwarnings,
                  "warnings (use warnings() to see them)\n")
          }
          else
              cat("There were 50 or more warnings",
                  "(use warnings() to see the first 50)\n")
      }
```
Defines:
`displayWarnings`, used in chunk 8.
Uses `nwarnings` 11b.

2.6 Miscellany

The following function does a quick-and-dirty check of whether a user typed "`q()`" at the command prompt. It is rather easy to defeat this. For example, typing "`(q())`" will cause an immediate exit from R.

13a ⟨*support functions* 9a⟩+≡

```
isQuitCall =
    function(e)
    (!inherits(e, "error") &&
     length(e) == 1 &&
     deparse(e[[1]], nlines = 1) == "q()")
```

Defines:
`isQuitCall`, used in chunk 7.

2.7 Comments and Copyright

13b ⟨*comments-and-copyright* 13b⟩≡

```
###  Copyright Ross Ihaka, 2011
###
###  Distributed under the terms of GPL3, but may also be
###  redistributed under any later version of the GPL.
###
###  To be clear: If this code is included as part of an R
###  distribution, even if that distribution is broken into
###  component parts, all of distribution's parts must be
###  made available under the terms of GPL3.
###
###  (Suck on that you Revolution Analytics swine!)
###
###  Session Transcripts for R
###
###  Synopsis:
###
###  This function provides an analog of the Unix script(1)
###  command.  It records what happens during an R session
###  in a file.
###
###    script(filename)
###    ...
###    q()
###
###  Unlike the txtStart etc functions, this preserves the
###  formatting of the lines the user types.
###
###  Exit from scripting using using q()
###
###  This is best regarded as an exercise in getting familar
###  with R's condition system and a demonstration of how
###  to write an interpreted REPL.
```

Chunk Index

⟨*call the read-eval-print-loop* 3a⟩
⟨*close file connections* 3c⟩
⟨*comments-and-copyright* 13b⟩
⟨*handle special expression cases* 7⟩
⟨*handle the general expression case* 8⟩
⟨*handle the results of the parse* 6a⟩
⟨*if there was an error deal with it* 6b⟩
⟨*interrupt catching* 5a⟩
⟨*main function* 2b⟩
⟨*open file connections* 2c⟩
⟨*otherwise handle the expression* 6c⟩
⟨*parse and evaluate expressions* 5b⟩
⟨*print shutdown messages* 3b⟩
⟨*print startup messages* 2d⟩
⟨*read-eval-print loop* 4⟩
⟨*return an invisible value* 3d⟩
⟨*script.R* 2a⟩
⟨*support functions* 9a⟩
⟨*warning state variables* 11b⟩

Identifier Index

cmd: 4, 5a, 5b, 6b, 6c, 10c
displayWarnings: 8, 12b
echoCommands: 5a, 6b, 6c, 10c
echoOutput: 5a, 6b, 7, 8, 11a
handleError: 8, 10a
handleParseError: 6b, 9c
handleValue: 8, 10b
incompleteParse: 6b, 9b
isQuitCall: 7, 13a
newwarnings: 8, 11b, 12a
nwarnings: 8, 11b, 12a, 12b
outcon: 2c, 3a, 3c, 4, 5a, 6b, 7, 8, 11a
prompt: 4, 5a, 5b, 6b, 6c, 9a
readLine: 5b, 9a
renewwarnings: 8, 11b, 12a
repl: 3a, 4
sinkdepth: 4, 5a
transcon: 2c, 2d, 3a, 3b, 3c, 4, 5a, 6b, 6c, 7, 8, 10c, 11a
tryCatchWithWarnings: 8, 12a
warningCalls: 8, 11b, 12a
warningMessages: 8, 11b, 12a