Dealing with errors

January 23, 2019

Warnings and errors

As you run R code, you'll experience a variety of errors and warnings. Be default, warnings simply produce a message. Unlike errors, a warning does not halt the execution of code (though one can change settings so that warnings act like errors). A warning can be issued in your own code via the `warning` function.

In contrast to warnings, errors do stop the execution of code. These can be issued using the `stop` function. Another function that can issue errors is `stopifnot`. It takes a logical condition as its parameter and issues an error if the condition evaluates to `FALSE`.

Some errors can be classified as "recoverable." For these, you may "catch" the error, handle it in some way, and then continue the execution of your program instead of halting your program. It is possible to do this in R (see: `tryCatch`, `withCallingHandlers`, `withRestarts`), but it is beyond the scope of this course.

A common use case for issuing errors and warnings is validation of function parameters. You may want to do checks on the data type (see `inherits`), `NULL` checks, etc. to assure the data that is provided to your function meets expectations. For example, if your function expects a matrix, but a data frame is provided, you may wish to convert it to a matrix but only after issuing a warning. You may also want to do checks on number of rows/columns, missing values, etc.

Unit testing

In non-trivial programming tasks, we want to be able to verify that changes to the code are not breaking some of the elementary functions. To help mitigate this problem, we may want to deploy a number of functions that validate that some functions produce the expected predetermined results. This is what "unit testing" accomplishes.

In R we have the `testthat` package:

```
library(testthat)
```

Imagine that we have the following function that pulls the numbers from a string.

```
extract_numbers <- function(x) {
    if (is.factor(x)) {
        warning("Input is factor")
        x <- as.character(x)
    }
    if (!is.character(x)) {
        stop("Input must be character")
    }
    str <- gsub("[^0-9]", "", x)
    return(as.numeric(str))
}
```

We want to test several things. First that given a string it actually extracts all the digits and only the digits:

```
context("Extract numbers")
test_that("extract_numbers extracts the digits", {
    expect_equal(extract_numbers("a1"), 1)
    expect_equal(extract_numbers("1"), 1)
})
```

But we also want to make sure that the function behaves correctly if the input does not contain any digits:

```
test_that("extract numbers of a string with no digits produces NA", {
    expect_equal(extract_numbers("a"), NA_integer_)
})
```

Or when the input is unexpected. In particular, we want to make sure the function outputs correct warnings and errors:

```
test_that("extract numbers of factor is a warning", {
    expect_warning(extract_numbers(factor("a")), "Input is factor")
})


test_that("extract numbers of number is error", {
    expect_error(extract_numbers(1), "Input must be character")
})
```

The `testthat` package offers a few more options but they all have a similar structure: we test the behavior of the function against what we expect to happen.

Which tests to run is largely a matter of ingenuity in deciding which are the most important functions in our code and how they can fail. However, there are tools that can help us.

Debugging

One of the most useful debugging tools in R is the `traceback` function. When called it provides a stack trace for the most recent error that occured. That is, it lists the stack of function calls that led up to the point where the error was issued.

Another useful tool is R's built-in browser, which allows you to stop a program in mid-execution and enter interactive mode, where you can inspect variables and/or execute your program one line at a time. This can be initiated by a call to `browser`, which is placed at any point in your code, usually in some condition that checks for the existence of an error.

```
if(nrow(data) == 0) # Data set should not be empty.
    browser() # Let's look at variables for clues!
```

Do note that calls to `browser` should not appear in "production code." Debuggers are useful for programmers, but users of your program don't want to see it! After your program is debugged, you should either remove calls to browser or wrap them in a condition, e.g., a testing flag.

Further, there is a set of functions that allow you to initiate the browser when a particular function is called.

```
debug(myfunction)
```

A browser will be opened whenever `myfunction` is called. Debugging for that function can be turned off as follows:

```
undebug(myfunction)
```

An alternative is `debugonce`, which works like `debug` except that the browser is only opened on the next call to your function, as if an implicit `undebug` were called afterwards.

In addition to normal R commands, R's browser supports a number of commands to assist in debugging. Here are a few:

- `c`: Exit the browser and continue execution of your program

- `Q`: Exit the browser and stop execution of your program
- `f`: Finish execution of the current loop or function.
- `n`: Evaluate the next statement, stepping over function calls.
- `s`: Evaluate the next statement, stepping into function calls.
- `where`: Print a stack trace

Logging console output

Text that is normally sent to the console can be redirected using the `sink` function. This can be useful when debugging programs or creating documents. To begin logging, you can `sink` with the specified output file name, and to stop logging, you call `sink` with no arguments.

```
sink("output_log.txt")

## DO STUFF

sink()
```

** NOTE: ** R Markdown makes use of `sink`. If the browser is opened while R Markdown is being processed, you will need to call `sink` with no arguments to view variables in the console.