Debugging

Notifications

There are several ways that R can provide feedback:

- `message`: generic notification produced by `message()` function; this does not stop execution.

- `warning`: indicates something unexpected happens; produced by `warning()` function; does not

stop execution.

- `error`: indicates a fatal problem; produced by `stop()` function, which halts execution.

- `condition`: a generic concept for indicating something unexpected may occur.

Warnings and Errors

```
> x <- c(1,2,3,4)
```

```
> y <- c(-1,0,1)
```

```
> xy <- x*y
```

```
Warning message:
```

```
In x * y : longer object length is not a multiple of shorter object length
```

```
> xy
```

```
[1] -1  0  3 -4
```

```
> xz <- x*z
```

```
Error: object 'z' not found
```

```
> xz
```

```
Error: object 'xz' not found
```

Example

Here is a simple function for checking passed values:

```
is_even <- function(x) {
```

```
    if (x %% 2) {
```

```
        print("Value is odd")
```

```
    } else {
```

```
        print("Value is even")
```

```
    }
```

```
    invisible(x)
```

```
}
```

```
> is_even(5)
```

```
[1] "Value is odd"
```

```
> is_even(NA)
```

```
Error in if (x%%2) { : argument is not interpretable as logical
```

Example

```
is_even <- function(x) {
```

```
    if(is.na(x)) print("Value is NA")
```

```
    else if(x %% 2) print("Value is odd")
```

```
    else print("Value is even")
```

```
    invisible(x)
```

```
}
```

```
> is_even(NA)
```

```
[1] "Value is NA"
```

```
> is_even(log(-1))
```

```
[1] "Value is NA"
```

```
Warning message:
```

```
In log(-1) : NaNs produced
```

Preventing Bugs

In some situations, you may be aware of conditions that would cause your code to silently fail. You

can guard against this by inserting checks in the code. For example, in our secant method code, we

need the point `xc` to be between `xa` and `xb`. The `stopifnot` function will throw an error if the

specified condition is not met:

```
golden_section <- function(f, xa, xb, xc, tol=1e-9) {
```

```
    stopifnot(xc < xb)
```

```
    stopifnot(xc > xa)
```

```
...
```

```
> golden_section(cos, 2,5,6)
```

```
Error: xc < xb is not TRUE
```

Be careful about over-using this approach. If the code would fail on its own, there is no need to add a

manual check.

Locating the Problem

There are several steps that will help you diagnose an issue:

- How did you call your function? What was the input?

- What were you expecting as a result?

- What was the actual result?

- Can you reproduce the problem?

It is useful to use a RNG seed, so that stochastic output can be exactly reproduced.

`print` Statements

A rudimentary form of debugging is to place `print` (or `cat`) statements at key locations of your code,

to verify the values of variables:

```
secant <- function(fun, x0, x1, tol=1e-9, max_iter=100) {
```

```
    # Keep track of number of interations
```

```
    iter <- 0
```

```
    # Evaluate function at initial points
```

```
    f1 <- fun(x1)
```

```
    f0 <- fun(x0)
```

```
    cat("f1:", f1, "f0:", f0, "\n")
```

```
# Loop
```

```
    while((abs(x1-x0) > tol) && (iter<max_iter)) {
```

```
        # Calculate new value
```

```
        x_new <- x1 -  f1*(x1 - x0)/(f1 - f0)
```

```
        print(sprintf("The new x value is %f", x_new))
```

```
        # Replace old value with current
```

```
        x0 <- x1
```

```
        x1 <- x_new
```

```
        f0 <- f1
```

```
        f1 <- fun(x1)
```

```
        # Increment counter
```

```
        iter <- iter + 1
```

```
    }
```

```
}
```

`sprintf` allows for fancier formatting.

R Debugging Tools

While it is possible to track down and fix bugs in R without any specialized debugging tools, several

functions are available that make debugging more interactive and effective:

- `traceback` prints out the function call stack when an error is generated.

- `debug` halts execution at the start of the next function, so that one may step through it.

- `browser` halts execution at the current line, so that one may step through the code at that point.

- `trace` inserts debugging code at specific locations in a function.

- `recover` allows one to interactively modify conditions during an error that allows execution to

continue.

traceback

By default `traceback()` prints the call stack of the last uncaught error, *i.e.*, the sequence of calls that

lead to the error. This is useful when an error occurs with an unidentifiable error message.

```
> sample(x)
```

```
Error in sample(x) : object 'x' not found
```

```
> traceback()
```

```
1: sample(x)
```

```
> lm(x~y)
```

```
Error in eval(expr, envir, enclos) : object 'x' not found
```

```
> traceback()
```

```
7: eval(expr, envir, enclos)
```

```
6: eval(predvars, data, env)
```

```
5: model.frame.default(formula = x ~ y, drop.unused.levels = TRUE)
```

```
4: model.frame(formula = x ~ y, drop.unused.levels = TRUE)
```

```
3: eval(expr, envir, enclos)
```

```
2: eval(mf, parent.frame())
```

```
1: lm(x ~ y)
```

The default display is of the stack of the last uncaught error as stored as a list of deparsed calls in

.Traceback, which `traceback` prints in a user-friendly format.

debug

Frequently, we might want to step through a function line-by-line to locate a bug. Calling `debug` with

a particular function as its argument *flags* that function for debugging. Then, each time it is called,

execution stops just prior to the function call, so that the user can walk through it:

```
> debug(sample)
```

```
> sample(c(1,2,3))
```

```
debugging in: sample(x)
```

```
debug: {
```

```
    if (length(x) == 1L && is.numeric(x) && x >= 1) {
```

```
        if (missing(size))
```

```
            size <- x
```

```
        .Internal(sample(x, size, replace, prob))
```

```
    }
```

```
    else {
```

```
        if (missing(size))
```

```
            size <- length(x)
```

```
        x[.Internal(sample(length(x), size, replace, prob))]
```

```
    }
```

```
}
```

```
Browse[2]>
```

debug

Typing `n` executes the current line and moves to the next line; `c` executes the rest of the function

without stopping; `Q` exits the debugger; and `where` shows you your current location, incase you get

lost:

```
Browse[2]> n
```

```
debug: if (length(x) == 1L && is.numeric(x) && x >= 1) {
```

```
    if (missing(size))
```

```
        size <- x
```

```
    .Internal(sample(x, size, replace, prob))
```

```
} else {
```

```
    if (missing(size))
```

```
        size <- length(x)
```

```
    x[.Internal(sample(length(x), size, replace, prob))]
```

```
}
```

```
Browse[2]> n
```

```
debug: if (missing(size)) size <- length(x)
```

```
Browse[2]> n
```

```
debug: size <- length(x)
```

```
Browse[2]> n
```

```
debug: x[.Internal(sample(length(x), size, replace, prob))]
```

```
Browse[2]> n
```

```
exiting from: sample(c(1, 2, 3))
```

```
[1] 3 2 1
```

```
> undebug(sample)
```

```
browser
```

A call to `browser` anywhere in your code invokes the browser when execution reaches that line, rather

than at the start of the corresponding function. So, if you know more precisely where an error may be,

this is a more direct approach:

```
secant <- function(fun, x0, x1, tol=1e-9, max_iter=100) {
```

```
    # Keep track of number of interations
```

```
    iter <- 0
```

```
    # Evaluate function at initial points
```

```
    f1 <- fun(x1)
```

```
    f0 <- fun(x0)
```

```
# Loop
```

```
    while((abs(x1-x0) > tol) && (iter<max_iter)) {
```

```
        # Calculate new value
```

```
        browser()
```

```
        x_new <- x1 -  f1*(x1 - x0)/(f1 - f0)
```

```
...
```

```
> secant(f_test, 2, 3)
```

```
Called from: secant(f_test, 2, 3)
```

```
Browse[1]> n
```

```
debug at secant.r#13: x_new <- x1 - f1 * (x1 - x0)/(f1 - f0)
```

```
browser
```

You can pass `browser` a condition argument that only invokes the browser when that condition is met.

For example, if you only want to look at the code if a variable is negative, you can inject this

statement:

```
browser(x < 0)
```

which is shorthand for:

```
if (x < 0) browser()
```

As another example, if a problem seems to arise in your code after several iterations, you can use your

counter value as a condition:

```
browser(i > 100)
```

which will trigger the browser at the 101st iteration.

`setBreakpoint`

Rather than editing your source code, `setBreakpoint` allows you to invoke the browser at a particular

line of the code:

```
setBreakpoint('secant.r', 12)
```

These breakpoints can be set *during* a debugging session, as well as before.

trace

The `trace` function allows you to temporarily add arbitrary code to a function, without permanently

changing it.

Among its arguments, `trace` accepts the name of the function to be traced (`what`), a function or

unevaluated evaluated expression to execute (`tracer`), and the line number at which to execute it

```
(at).
```

```
trace(what, tracer, exit, at, print, signature,
```

```
    where = topenv(parent.frame()), edit = FALSE)
```

For complex tracing, the `edit=TRUE` argument can be passed to `trace`. This will invoke a text editor,

allowing you to insert tracing code wherever desired.

trace

Calling `trace` on a function without an argument will print the function name whenever it is called.

For example:

```
> trace(sum)
```

```
> hist(rnorm(100))
```

```
trace: sum
```

```
trace: sum
```

```
trace: sum
```

Here, the function `hist` (plots a histogram) calls the `sum` function 3 times.

```
trace("secant", browser)
```

will start the browser, similar to placing a `browser` call at the start of `secant`.

Calling `untrace` allows you to remove trace code.

recover

`browser` allows you to browse the environment in the current function call, but not the environments

for previous function calls. In some situations, you may want to halt execution in one location, then

browse a previous function call to hunt down a bug. `recover` allows you to jump up to higher

positions in the call stack:

```
> options(error=recover)
```

```
> lm(y~x)
```

```
Error in eval(expr, envir, enclos) : object 'y' not found
```

```
Enter a frame number, or 0 to exit
```

```
1: lm(y ~ x)
```

```
2: eval(mf, parent.frame())
```

```
3: eval(expr, envir, enclos)
```

```
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
```

```
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
```

```
6: eval(predvars, data, env)
```

```
7: eval(expr, envir, enclos)
```

```
Selection:
```