Code style, line profiling, and unit testing

Tad Dallas

Reading for this week:

> Wickham, H. (2011). testthat: Get started with testing. The R Journal, 3(1), 5-10.

Style guide

As Hadley Wickham states in his Advanced R book (http://adv-r.had.co.nz/)

> Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations.

Code style may seem like something to not worry about, but by being a bit stricter with yourself in how you code (especially when it comes to variable and function naming), it will make your code more readable, and your life a bit easier when writing long analyses or looking back on code from past projects.

There are many existing packages which help you follow good coding principles. I will recommend none of them. This is not because they are not useful, but this course is on *reproducible research* and one way to enhance reproducibility is to limit dependencies. That being said, RStudio has some built-in code formatting tools, which I will also not touch on, for much of the same reason.

Naming

Naming is going to be a big deal here, because we name everything. We name `R` scripts (e.g., 'myAnalysis.R'), variables (`myVariable`), and our functions (`myFunction`).

scripts: I promote the use of a single analytical file (I really like `R markdown`, but you can use `.R` files instead). If you have lots of functions in the analysis, it may be useful to package these together as an `R` package, or to just separate them in another `.R` file which can be `source()`d in prior to the main analysis. Script names should be be informative. For instance, if you have a single analytical file, it may be alright to name it `analysis.Rmd`. However, if you have support functions in a separate file, this file should be named something like `supportFunctions.R`.

Do not use lowercase r (bad: `supportFunctions.r`). Do not include spaces (bad: `support Functions.r`).

objects: there is some wiggle room here, but it is important to select a naming scheme and to stick with it. For instance, it may be evident from above that I am a fan of lowerCamelCase. This is where the start is always lower case, and subsequent words are not separated with spaces, but the first letter of each word is capitalized. Wickham and the RStudio crowd tend to be fans of using underscore (`_`) characters and all lowercase letters.

Do not overwrite existing `R` functionality (e.g., naming a variable `c`, `t`, `ts`, `matrix`, etc.) Do not use uninformative names (e.g., `var`, `df`, etc.)

functions: In the context of code, objects are nouns and functions are verbs, so I like to start function names with an action verb, which I tend to try to conserve across functions. Two common verbs I use are `make` and `get` as prefixes. For instance, `getOccurrences` might pull species occurrence data from an online resource,

`getModel` might create some model of the resulting data, and `getHeatmap` might plot the data and store the plot somewhere.

Spacing

I like languages that are agnostic to spacing (LaTeX) over languages that are super sensitive (yaml). `R` is nice in that spacing is more stylistic, so the code will not error-out. . . it will just be ugly.

Put spaces around all operators (+, -, =, <, etc.) Do not put spaces around : or :: calls (1:10) Put spaces after commas (plot(x, y, xlab=”))

The main objective of spacing is to make code readable, a common theme of all style guides (http://google.github.io/styleguide/).

Indentation and line breaks

This is one that I have seen ignored most often, and have myself been guilty of ignoring.

When writing functions, separate the initial function definition from the function body. This is good practice whenever you use curly braces, that there should probably be a carriage return (a line break).

```
getMean <- function(x){
    return(mean(x, na.rm=TRUE))
}
```

```
if(brody == 'cat'){
    print('brody is a cat')
}
```

Some will tell you to not use tab characters and instead use spaces. This is best practice, but I like tabs. The downside is that tabs mean something different on different machines, so that some person who has tabs representing 8 spaces is going to be find your code difficult to read if you use tabs.

Also, some will tell you to put new lines on the same level as previous lines when mid-argument.

```
myList <- list(a='orange',
               b='peach',
               d='plum')
```

I am a huge non-fan of this. I think it takes up entirely too much space and is ugly. I would start a new line, as it is important to be cognizant that text editors will wrap lines over `n` characters, with `n` typically being around 80 or so. This 80 is based on typewriters, and now different language style guides use different values for the max number of desired characters per line (but pretty much all have a limit).

```
myList <- list(a='orange',
    b='peach', d='plum')
```

Assignment

Stop using `=`. Use `<-`. I know it is a bit annoying because it takes an extra keystroke, but it is supposedly good style.

Line and memory profiling

Profiling `R` code – and code in general – is incredibly useful in order to determine computational and performance bottlenecks. Writing optimized, well-styled, readable code is the goal (or at least choose 2). Line/memory profiling is a way to see how much time/memory each step of a multi-step process takes. There are `R` packages that you can use for line profiling (`lineprof`) and benchmarking (`microbenchmark`) your code. We will use base `R` functions in the `utilities` package, because we want to be as reproducible as possible, so we need to limit dependencies.

line and memory profiling

`system.time()`: this is a useful function for determining how much time a given command will take, where the command is bounded by the `system.time()` function

```
system.time(runif(10))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(runif(1000000))
```

```
##    user  system elapsed
##   0.012   0.011   0.024
```

`user time`: time charged to the CPU(s) for this expression `elapsed time`: time for you (includes data handling steps) `system time`: the difference between `elapsed time` and `user time`

In a multi-step process, you can wrap separate functions in `system.time` calls to determine which part of the pipeline takes the most time.

`Rprof` is a base `R` function that allows you to do both line and memory profiling, which will address not only where the biggest bottleneck in your code is time-wise, but also memory-wise. These are sometimes the same place, but not always.

```
Rprof(interval=0.0001, memory.profiling=TRUE, line.profiling=TRUE)

a <- runif(1000)

hilbert <- function(n) {
    i <- 1:n
  1 / outer(i - 1, i, "+")
}

for(i in 1:length(a)){
    hilbert(i)*a[i]
}

lm(a ~ sample(a))
```

```
##
## Call:
## lm(formula = a ~ sample(a))
##
## Coefficients:
## (Intercept)    sample(a)
##     0.50420     -0.02033
```

```
Rprof(NULL)
```

`Rprof` should give you a general idea about which parts of your code are time- or computation- intensive. To dive more into memory usage of single objects in the interactive `R` session, you can use the `object.size()` function.

```
a <- runif(10)
object.size(a)
```

```
## 176 bytes
```

```
a2 <- runif(100000)
object.size(a2)
```

```
## 800048 bytes
```

We can also see how the memory needed to store an object changes when the object changes.

```
a <- list()
for(i in 1:10){
    a[[i]] <- runif(10000)
    print(object.size(a))
}
```

```
## 80104 bytes
## 160160 bytes
## 240224 bytes
## 320272 bytes
## 400336 bytes
## 480384 bytes
## 560448 bytes
## 640496 bytes
## 720608 bytes
## 800656 bytes
```

Writing robust code

The most efficient and well-documented code will still fail if it fails to error out 'gracefully'. This section will go over two core ideas: error handling and unit testing.

Error handling

If we try to take the mean of a character vector, it will return NA and issue a warning

> Warning message: argument is not numeric or logical: returning NA

If we try to multiply a numeric vector by a string, it will produce an error

> Error: non-numeric argument to binary operator

Writing functions that handle errors gracefully is an important skill for `R` programmers. It will save you lots of headaches if you incorporate warning and error messages into your code, and is essential if you wanted to develop `R` code for distribution.

`warning()` is a function that will issue a warning, and can be wrapped in an `if` statement within a function. As an example, we will create a function that is supposed to tell if an object (`obj`) is the color red or not. It is not well-written, because it will actually only detect if the input is either 'fire truck', 'stop sign', or 'clown

nose'. A `warning` to the user is helpful in this case. This is super artificial, and warnings should actually be in informative places where functions are not so silly (e.g., when NA or NULL values might bias results, when model convergence is not reached, etc.).

```
isRed <- function(obj){
    warning('this function has really high error rates')
    if(obj %in% c('fire truck', 'stop sign', 'clown nose')){
        print('yes')
    }else{
        print('maybe?')
    }
}
```

Errors are a bigger deal, as they stop the function from proceeding. That is, warnings simply print messages to the console, while errors `break` from whatever the function is doing and return nothing.

```
isRed <- function(obj){
    warning('this function has really high error rates')
    if(is.numeric(obj)){
        stop('obj is numeric, not a string')
    }
    if(obj %in% c('fire truck', 'stop sign', 'clown nose')){
        print('yes')
    }else{
        print('maybe?')
    }
}
```

```
isRed('my pencil')
```

```
## Warning in isRed("my pencil"): this function has really high error rates
```

```
## [1] "maybe?"
```

The following code errors out, producing the error

```
obj is numeric, not a string.
```

```
isRed(1)
```

There are many related functions in base `R` that allow you to write bombproof code, which is definitely a goal and will help ensure reproducibility.

Unit testing

We do unit testing as we write code often, in a very ad hoc way. That is, we run a chunk of code on subset or artificial data, to make sure the code is doing what we want it to do. This concept underlies the idea of unit testing as we will learn it. Here, we write a set of functions to perform an analysis. We know the output of these functions in terms of their class, data structure (shape, size, etc.), and sometimes some summary statistics on them. What we want to do is to design an automated way to make sure the functions are outputting reasonable output compared to what we would expect.

This is perhaps most useful to `R` packages, as small changes in one part of the package may break something in a different function. This would not be apparent to the package maintainer, but if automated tests were in place, the developer could be alerted.

> As a side note: this is how your homework is automatically graded – as we have touched on briefly – thanks to continuous integration and Travis CI, which runs the unit tests each time you make a push to Github with edits to your homework.

For this, we will have to extend beyond base `R`, using the `testthat` package. This package, and the general idea behind testing, is laid out in detail in http://rjournal.github.io/archive/2011-1/RJournal_2011-1.pdf#page=5 (the article that we read for class).

The way the `testthat` package recommends organizing tests is to have a `tests` folder in your analysis directory, which contains a file `testthat.R` and a folder containing `R` scripts of individual tests. Each test in the `tests` folder is an `R` script which is composed of multiple `expectations`. Below is an example.

```
testthat::context("your analysis")

testthat::test_that("isRed function does stuff", {

    testthat::expect_output(isRed('orange'), 'maybe?')
    testthat::expect_type(isRed('clown nose'), 'character')
    testthat::expect_equal(length(isRed('orange')), 1)
})
```

```
## [1] "yes"
## [1] "maybe?"
```

We see three things. First, we define a `context` which allows the grouping of multiple tests. Then we define the test (`test_that`) which wraps all the `expectations` inside a single function. If tests are met, there is no output. We can see this by breaking the test condition.

```
testthat::expect_type(isRed('clown nose'), 'numeric')
```

Which errors out with the error

> Error: isRed("clown nose") has type `character`, not `numeric`.

By issuing `testthat::test_dir()`, we can run all the tests contained in a given directory

Other useful testing functions

```
expect_null()
expect_true()
expect_success()
expect_more_than()
expect_identical()
expect_equal()
expect_error()
expect_warning()
```

Hadley mentions in his 2011 article (http://rjournal.github.io/archive/2011-1/RJournal_2011-1.pdf#page=5) the need for the incorporation of code coverage. The idea is that we can test code, but we may also want to know which functions or parts of our code are lacking appropriate testing. Code coverage basically tells you the fraction of your code that is covered by tests, and points to functions or areas which could be improved. We will not go into this in any more depth in this course.